PowerPC™ e500 Core Family Reference Manual

Supports
e500v1
e500v2

E500CORERM
Rev. 1, 4/2005

freescale™
semiconductor

How to Reach Us:

Home Page:
www.freescale.com

email:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
(800) 521-6274
480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku
Tokyo 153-0064, Japan
0120 191014
+81 2666 8080
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate,
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor
Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
(800) 441-2447
303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor
@hibbertgroup.com

Document Number: E500CORERM
Rev. 1, 4/2005

freescale™
semiconductor

Contents

Paragraph Number	Title	Page Number

About This Book

Part I
e500 Core

Chapter 1
Core Complex Overview

Contents

Contents

Contents

Chapter 3
Instruction Model

Contents

Contents

Contents

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

Chapter 4
Execution Timing

Contents

Paragraph Number	Title	Page Number

Contents

Chapter 6
Power Management

Chapter 7
Performance Monitor

Contents

Contents

Paragraph Number	Title	Page Number

Part II
e500 Core Complex

Chapter 9
Timer Facilities

Chapter 10
Auxiliary Processing Units (APUs)

Contents

Paragraph Number	Title	Page Number

Chapter 11
L1 Caches

Contents

Contents

Contents

Paragraph Number	Title	Page Number

Appendix A
Programming Examples

Appendix B
Guidelines for 32-Bit Book E

Appendix C
Simplified Mnemonics for PowerPC Instructions

Contents

Paragraph Number	Title	Page Number

Appendix D
Opcode Listings

Appendix E
Revision History

Index

Figures

Figures

Figure Number	Title	Page Number

Figures

Figures

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

Tables

Tables

Tables

Tables

Tables

Tables

About This Book

The primary objective of this user's manual is to describe the functionality of the e500 embedded microprocessor core for software and hardware developers. This book is intended as a companion to the *EREF: A Reference for Freescale Book E and the e500 Core* (hereafter referred to as EREF). The e500 is a PowerPC™ processor.

Note that, while previous versions of this manual covered only the e500v1 core (and referred to it simply as the e500 core), this version includes coverage of both the e500v1 and e500v2 cores. Where the two cores diverge, the differences are clearly delineated.

Book E is a PowerPC architecture definition for embedded processors that ensures binary compatibility with the user-instruction set architecture (UISA) portion of the PowerPC architecture as it was jointly developed by Apple, IBM, and Motorola. The version of the architecture jointly developed by Apple, IBM, and Motorola is referred to as the AIM version of the PowerPC architecture.

This document distinguishes between the three levels of the architectural and implementation definition, as follows:

- The Book E architecture. Book E defines a set of user-level instructions and registers that are drawn from the user instruction set architecture (UISA) portion of the AIM definition PowerPC architecture. Book E also include numerous other supervisor-level registers and instructions as they were defined in the AIM version of the PowerPC architecture for the virtual environment architecture (VEA) and the operating environment architecture (OEA).

  Because Book E defines a much different model for operating system resources (such as the MMU and interrupts), it defines many new registers and instructions.
- Freescale Book E implementation standards. In many cases, the Book E architecture definition provides a very general framework, leaving many higher-level details up to the implementation. To ensure consistency among its Book E implementations, Freescale has defined implementation standards that provide an additional layer of architecture between Book E and the actual devices.
- e500 implementation details. Each processor typically defines instructions, registers, bits within registers, and other aspects that are more detailed than either the Book E definition or the Freescale Book E implementation standards.

  This book describes all of the instructions and registers implemented on the e500, including those defined by Book E and those that are e500-specific.

Information in this book is subject to change without notice, as described in the disclaimers on the title page of this book. As with any technical documentation, it is the readers' responsibility to be sure they are using the most recent version of the documentation.

Audience

It is assumed that the reader understands operating systems, microprocessor system design, and the basic principles of RISC processing.

Organization

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

- Chapter 1, "Core Complex Overview," provides a general description of e500 functionality.
- Chapter 2, "Register Model," is useful for software engineers who need to understand the programming model for the three programming environments and the functionality of each register.
- Chapter 3, "Instruction Model," provides an overview of the addressing modes and a description of the instructions. Instructions are organized by function.
- Chapter 4, "Execution Timing," describes how instructions are fetched, decoded, issues, executed, and completed and how instruction results are presented to the processor and memory system. Tables are provided that indicate latency and throughput for each of the instructions supported by the e500.
- Chapter 5, "Interrupts and Exceptions," describes how the e500 implements the interrupt model as it is defined by the Book E architecture.
- Chapter 6, "Power Management," describes the power management facilities as they are defined by Book E and implemented in the e500 core.
- Chapter 7, "Performance Monitor," describes the e500 implementation of the performance monitor APU that is defined by the Freescale Book E implementation standards.
- Chapter 8, "Debug Support," describes the debug facilities as they are defined by Book E and implemented in the e500 core.
- Chapter 9, "Timer Facilities," describes the Book E-defined timer facilities implemented in the e500 core. These resources include the time base (TB), decrementer (DEC), fixed-interval timer (FIT), and watchdog timer.
- Chapter 10, "Auxiliary Processing Units (APUs)," lists the extensions to the Book E–defined programming model that are supported on the e500 and describes the e500-specific branch target buffer locking APU.
- Chapter 11, "L1 Caches," provides specific hardware and software details regarding the e500 cache implementation.

- Chapter 12, "Memory Management Units," provides specific hardware and software details regarding the e500 MMU implementation.
- Chapter 13, "Core Complex Bus (CCB)," describes those aspects of the CCB that are configurable or that provide status information through the programming interface. It provides a glossary of those signals that are mentioned in other chapters to offer a clearer understanding of how the core is integrated as part of a larger device.
- Appendix A, "Programming Examples," provides example code for use of creating atomic primitives with load and store with reservation instructions and for programming multiple-precision shifts.
- Appendix B, "Guidelines for 32-Bit Book E," provides a set of guidelines for software developers. Application software written to these guidelines can be labelled 32-bit Book E applications and can expect to execute properly on all implementations of Book E, both 32-bit and 64-bit implementations.
- Appendix C, "Simplified Mnemonics for PowerPC Instructions," provides a set of simplified mnemonic examples and symbols.
- Appendix D, "Opcode Listings," lists opcodes by mnemonic and by opcode. It includes an alphabetical listing that includes simplified mnemonics and the architecturally defined instructions (with syntax) to which they map.
- Appendix E, "Revision History," contains a revision history for this manual.
- This book also includes an index.

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

- *The PowerPC Architecture: A Specification for a New Family of RISC Processors*, Second Edition, by International Business Machines, Inc.

  For updates to the specification, see http://www.austin.ibm.com/tech/ppc-chg.html
- *Computer Architecture: A Quantitative Approach*, Third Edition, by John L. Hennessy and David A. Patterson.
- *Computer Organization and Design: The Hardware/Software Interface*, Second Edition, David A. Patterson and John L. Hennessy.

Related Documentation

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

- *EREF: A Reference for Freescale Book E and the e500 Core* (EREF)—This book provides a higher-level view of the programming model as it is defined by Book E, the Freescale Book E implementation standards, and the e500 microprocessor.
- *e500 Software Optimization Guide (eSOG)* (AN2665)—This manual provides information to programmers so that they may write optimal code for the e500.
- Reference manuals—These books provide details about individual implementations and are intended for use with the *EREF*.
- 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 reference manuals.
- Hardware specifications—Hardware specifications provide specific data regarding bus timing, signal behavior, and AC, DC, and thermal characteristics, as well as other design considerations.
- Product briefs—Each device has a product brief that provides an overview of its features. This document is roughly the equivalent to the overview (Chapter 1) of an implementation's reference manual.
- Application notes—These short documents address specific design issues useful to programmers and engineers working with Freescale processors.

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

Conventions

This document uses the following notational conventions:

cleared/set	When a bit takes the value zero, it is said to be cleared; when it takes a value of one, it is said to be set.
mnemonics	Instruction mnemonics are shown in lowercase bold.
italics	Italics indicate variable command parameters, for example, **bcctr***x*.
	Book titles in text are set in italics.
	Internal signals are set in italics, for example, $\overline{\textit{qual BG}}$.
0x0	Prefix to denote hexadecimal number
0b0	Prefix to denote binary number
rA, **r**B	Instruction syntax used to identify a source GPR

rD	Instruction syntax used to identify a destination GPR
REG[FIELD]	Abbreviations for registers are shown in uppercase text. Specific bits, fields, or ranges appear in brackets. For example, MSR[LE] refers to the little-endian mode enable bit in the machine state register.
x	In some contexts, such as signal encodings, an unitalicized x indicates a don't care.
x	An italicized *x* indicates an alphanumeric variable.
n	An italicized *n* indicates an numeric variable.
¬	NOT logical operator
&	AND logical operator
|	OR logical operator
0000	Indicates reserved bits or bit fields in a register. Although these bits can be written to as ones or zeros, they are always read as zeros.

Terminology Conventions

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

Table i. Terminology Conventions

Architecture Specification	This Manual
Change bit	Changed bit
Extended mnemonics	Simplified mnemonics
Out of order memory accesses	Speculative memory accesses
Privileged mode (or privileged state)	Supervisor level
Problem mode (or problem state)	User level
Reference bit	Referenced bit
Relocation	Translation
Storage (locations)	Memory
Storage (the act of)	Access

Part I
e500 Core

Part I specifically describes the e500 core, excluding details about cache memories and MMU features. It contains chapters that apply to the entire core, as follows:

- Chapter 1, "Core Complex Overview," summarizes the e500 core. This a 32-bit implementation of the Book E PowerPC architecture, including a recognition that different processor implementations may require extensions or deviations from the architectural descriptions.
- Chapter 2, "Register Model," describes the e500 core register model as defined in Book E and the additional implementation-specific registers unique to the e500 core, including a Book E SPR model.
- Chapter 3, "Instruction Model," provides information about the Book E architecture as it relates specifically to the e500 core complex. The e500 core complex also implements several APUs, which define additional instructions, registers, and interrupts. The chapter also features operand conventions, branch prediction, memory access alignment support, and memory synchronization sections.
- Chapter 4, "Execution Timing," describes the e500 core's operations performance as defined by instructions and how it reports the results of instruction execution. It gives detailed descriptions of how the core execution units work and how these units interact with other parts of the processor, such as the instruction fetching mechanism, register files, and caches. Included are examples of instruction sequences and tables that provide information useful to assembly language programmers for optimizing performance.
- Chapter 5, "Interrupts and Exceptions," is a general description of the Book E interrupt and exception model and gives details of the additions and changes to that model that are implemented in the e500 core complex.
- Chapter 6, "Power Management," describes the hardware and software resources the system uses to minimize its power consumption. This chapter regards the power management facilities as they are defined by Book E and implemented in devices that contain the e500 core, but its scope is limited to features of the core only.
- Chapter 7, "Performance Monitor," describes the e500 implementation of the performance monitor APU that is defined by the Freescale Book E implementation standards.
- Chapter 8, "Debug Support," describes the e500 core complex internal debug capabilities and associated features. Included are important deviations to the Book E debug mode.

Chapter 1
Core Complex Overview

This chapter provides an overview of the PowerPC™ e500 microprocessor core.

References to e500 are true for both the e500v1 and e500v2.

This chapter includes the following:

- An overview of the Book E version of the PowerPC architecture features as implemented in this core and a summary of the core feature set
- A summary of the instruction pipeline and flow
- An overview of the programming model
- An overview of interrupts and exception handling
- A description of the memory management architecture
- High-level details of the e500 core memory and coherency model
- A brief description of the core complex bus (CCB)
- A summary of the Book E architecture compatibility and migration from the original version of the PowerPC architecture as it is defined by Apple, IBM, and Motorola (referred to as the AIM version of the PowerPC architecture)

The e500 core provides features that the integrated device may not implement or may implement in a more specific way.

1.1 Overview

The e500 processor core is a low-power implementation of the family of reduced instruction set computing (RISC) embedded processors that implement the Book E definition of the PowerPC architecture. The e500 is a 32-bit implementation of the Book E architecture using the lower words in the 64-bit general-purpose registers (GPRs).

Figure 1-1 is a block diagram of the processor core complex that shows how the functional units operate independently and in parallel. Note that this conceptual diagram does not attempt to show how these features are physically implemented.

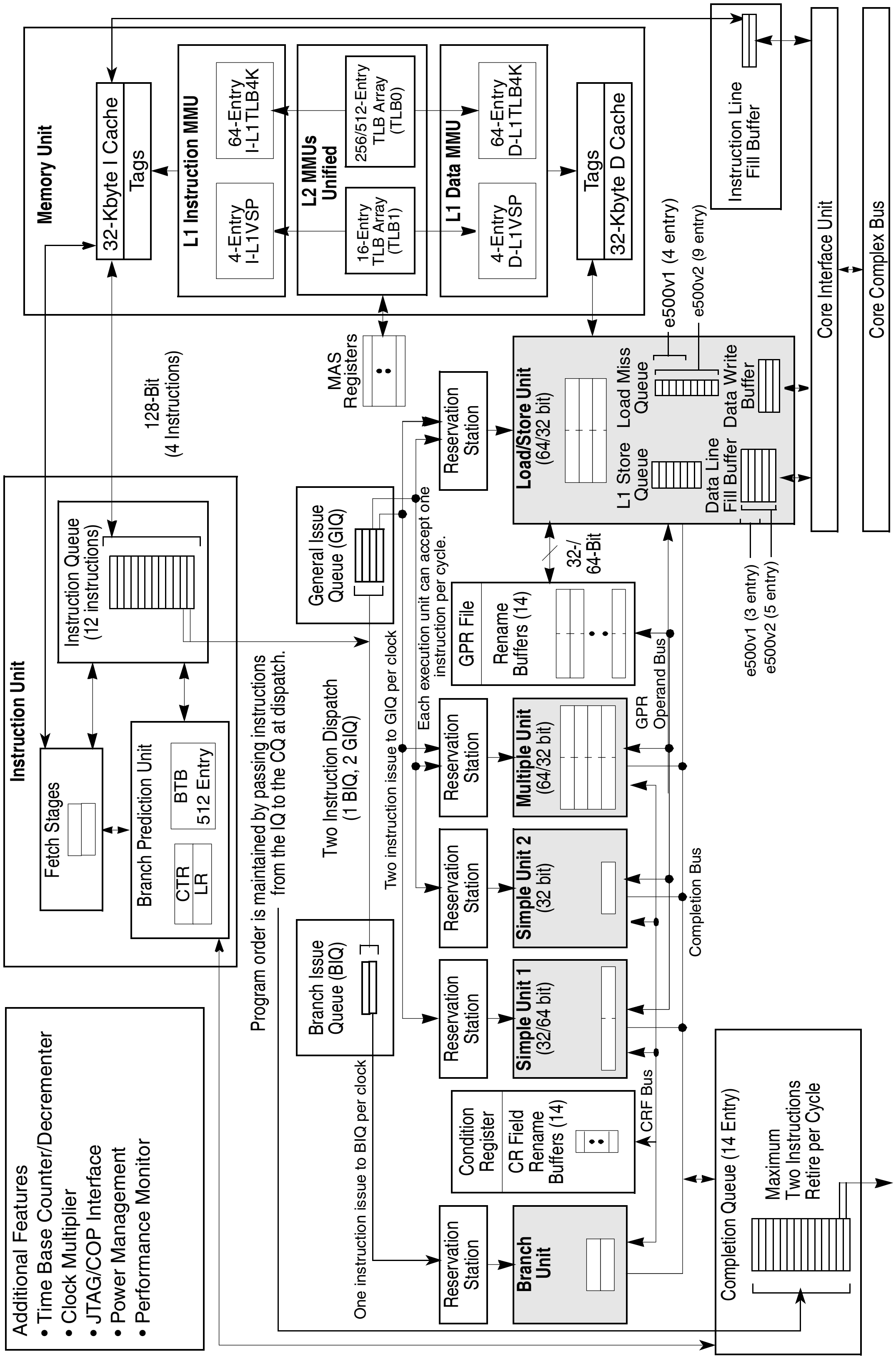


Figure 1-1. e500 Core Complex Block Diagram

Book E allows processors to provide auxiliary processing units (APUs), which are extensions to the architecture that can perform computational or system management functions. One of these on the e500 is the signal processing engine APU (SPE APU), which includes a suite of vector instructions that use the upper and lower halves of the GPRs as a single two-element operand. Most APUs implemented on the e500 are defined by the Freescale Semiconductor Book E implementation standards (EIS).

1.1.1 Upward Compatibility

The e500 provides 32-bit effective addresses and integer data types of 8, 16, and 32 bits, as defined by Book E. It also provides two-element, 64-bit data types for the SPE APU and the embedded vector floating-point APU, which include instructions that operate on operands comprised of two 32-bit elements. For detailed information regarding the e500 instruction set, see Chapter 3, “Instruction Model.”

The embedded single-precision scalar floating-point APU provides 32-bit single-precision instructions.

NOTE

> The SPE APU and embedded floating-point APU functionality is implemented in all PowerQUICC III devices. However, these instructions will not be supported in devices subsequent to PowerQUICC III. Freescale Semiconductor strongly recommends that use of these instructions be confined to libraries and device drivers. Customer software that uses SPE or embedded floating-point APU instructions at the assembly level or that uses SPE intrinsics will require rewriting for upward compatibility with next-generation PowerQUICC devices.
>
> Freescale Semiconductor offers a libmoto_e500 library that uses SPE instructions. Freescale will also provide libraries to support next-generation PowerQUICC devices.

1.1.2 Core Complex Summary

The core complex is a superscalar processor that can issue two instructions and complete two instructions per clock cycle. Instructions complete in order, but can execute out of order. Execution results are available to subsequent instructions through the rename buffers, but those results are recorded into architected registers in program order, maintaining a precise exception model. All arithmetic instructions that execute in the core operate on data in the GPRs. Although the GPRs are 64 bits wide, only SPE APU, DPFP (e500v2 only), and embedded vector floating-point instructions operate on the upper word of the GPRs; the upper 32 bits are not affected by other 32-bit instructions.

The processor core integrates two simple instruction units (SU1, SU2), a multiple-cycle instruction unit (MU), a branch unit (BU), and a load/store unit (LSU).

The LSU and SU2 support 64- and 32-bit instructions.

The ability to execute five instructions in parallel and the use of simple instructions with short execution times yield high efficiency and throughput. Most integer instructions execute in 1 clock cycle. A series of independent vector floating-point add instructions can be issued and completed with a throughput of one instruction per cycle.

The core complex includes independent on-chip, 32-Kbyte, eight-way set-associative, physically addressed caches for instructions and data. It also includes on-chip first-level instruction and data memory management units (MMUs) and an on-chip second-level unified MMU.

- The first-level MMUs contain two four-entry, fully-associative instruction and data translation lookaside buffer (TLB) arrays that provide support for demand-paged virtual memory address translation and variable-sized pages. They also contain two 64-entry, 4-way set-associative instruction and data TLB arrays that support 4-Kbyte pages. These arrays are maintained entirely by the hardware with a true least-recently-used (LRU) algorithm.
- The second-level MMU contains a 16-entry, fully-associative unified (instruction and data) TLB array that provides support for variable-sized pages. It also contains a unified TLB for 4-Kbyte page size support, as follows:
  - a 256-entry, 2-way set-associative unified TLB for the e500v1
  - a 512-entry, 4-way set-associative unified TLB for the e500v2

  These second-level TLBs are maintained completely by the software.

The core complex allows cache-line-based user-mode locks on the contents in either the instruction or data cache. 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 core complex supports a high-speed on-chip internal bus with data tagging called the core complex bus (CCB). The CCB has two general-purpose read data buses, one write data bus, data parity bits, data tag bits, an address bus, and address attribute bits. The processor core complex supports out-of-order reads, in-order writes, and one level of pipelining for addresses with address-retry responses. It can also support single-beat and burst data transfers for memory accesses and memory-mapped I/O operations.

1.2 e500 Processor and System Version Numbers

Table 1-1 matches the revision code in the processor version register (PVR) and the system version register (SVR). These registers can be accessed as SPRs through the e500 core (see Chapter 2, "Register Model") or as memory-mapped registers defined by the integrated device (see the reference manual for the device).

Table 1-1. Revision Level-to-Device Marking Cross-Reference

SoC **Revision**	**e500v2 Core Revision**	**Processor Version Register (PVR)**	**System Version Register (SVR)**
1.0	1.0	0x8020_0010	SoC-dependent value
1.1	2.0	0x8020_0020	SoC-dependent value
2.0	2.0	0x8021_0010	SoC-dependent value

1.3 Features

Key features of the e500 are summarized as follows:

- Implements Book E 32-bit architecture
- Auxiliary processing units

  The branch target buffer (BTB) locking APU is specific to the e500. The BTB locking APU gives the user the ability to lock, unlock, and invalidate BTB entries; further information is provided in Table 1-5 and Section 10.2, "Branch Target Buffer (BTB) Locking APU." The EIS defines the following APUs:

  — Integer select. This APU consists of the Integer Select instruction, **isel**, which is a conditional register move that helps eliminate conditional branches, decreases latency, and reduces the code footprint.

  — Performance monitor. The performance monitor facility provides the ability to monitor and count predefined events such as processor clocks, misses in the instruction cache or data cache, types of instructions decoded, or mispredicted branches. The count of such events can be used to trigger the performance monitor exception. Additional performance monitor registers (PMRs) similar to SPRs 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**). See Section 1.12, "Performance Monitoring."

  — Cache locking. This APU allows instructions and data to be locked into their respective caches on a cache block basis. Locking is performed by a set of touch and lock set instructions. This functionality can be enabled for user mode by setting MSR[UCLE]. The APU also provides resources for detecting and handling overlocking conditions.

  — Machine check. The machine check interrupt is treated as a separate level of interrupt. It uses its own save and restore registers (MCSRR0 and MCSRR1) and Return from

Machine Check Interrupt (**rfmci**) instruction. See Section 1.8, "Interrupts and Exception Handling."

— Single-precision embedded scalar and vector floating-point APUs. These instructions are listed in Table 1-4.

— Signal processing engine APU (SPE APU). Note that the SPE is not a separate unit; SPE computational and logical instructions are executed in the simple and multiple-cycle units used by all other computational and logical instructions, and 64-bit loads and stores are executed in the common LSU. Figure 1-1 shows how execution logic for SU1, the MU, and the LSU is replicated to support operations on the upper halves of the GPRs.

Note that the SPE APU and the two single-precision floating-point APUs were combined in the original implementation of the e500 v1, as shown in Figure 1-2.

	Vector and Floating-Point APUs	e500 v1	e500 v2
Original SPE Definition	SPE vector instructions **ev**…	√	√
Original SPE Definition	Vector single-precision floating-point **evfs**…	√	√
Original SPE Definition	Scalar single-precision floating-point **efs**…	√	√
	Scalar double-precision floating-point **efd**…		√

Figure 1-2. Vector and Floating-Point APUs

The e500 register set is modified as follows:

- GPRs are widened to 64 bits to support 64-bit load, store, and merge operations. Note that the upper 32 bits are affected only by 64-bit instructions.
- A 64-bit accumulator (ACC) has been added.
- The signal processing and embedded floating-point status and control register (SPEFSCR) provides interrupt control and status for SPE and embedded floating-point instructions.

These registers are shown in Figure 1-7. SPE instructions are grouped as follows:

- Single-cycle integer add and subtract with the same latencies for SPE APU operations as for the 32-bit equivalent
- Single-cycle logical operations
- Single-cycle shift and rotates
- Four-cycle integer pipelined multiplies
- 4-, 11-, 19-, and 35-cycle integer divides
- If **r**A or **r**B is zero, a floating-point divide takes 4 cycles; all other cases take 29 cycles.
- 4-cycle SIMD pipelined multiply-accumulate (MAC)
- 64-bit accumulator for no-stall MAC operations

  - 64-bit loads and stores
  - 64-bit merge instructions
- Cache structure—Separate 32-Kbyte, 32-byte line, 8-way set-associative level 1 instruction and data caches
  - 1.5-cycle cache array access, 3-cycle load-to-use latency
  - Pseudo-LRU (PLRU) replacement algorithm
  - Copy-back data cache that can function as a write-through cache on a page-by-page basis
  - Supports all Book E memory coherency modes
  - Supports EIS-defined cache-locking instructions, as listed in Table 1-3
- Dual-issue superscalar control
  - Two-instructions-per-clock peak issue rate
  - Precise exception handling
- Decode unit
  - 12-entry instruction queue (IQ)
  - Full hardware detection of interlocks
  - Decodes as many as two instructions per cycle
  - Decode serialization control
  - Register dependency resolution and renaming
- Branch prediction unit (BPU)
  - Dynamic branch prediction using a 512-entry, 4-way set-associative branch target buffer (BTB) supported by the e500 BTB instructions listed in Table 1-5.
  - Branch prediction is handled in the fetch stages.
- Completion unit
  - As many as 14 instructions allowed in 14-entry completion queue (CQ)
  - In-order retirement of as many as two instructions per cycle
  - Completion and refetch serialization control
  - Synchronization for all instruction flow changes—interrupts, mispredicted branches, and context-synchronizing instructions
- Issue queues
  - Two-entry branch instruction issue queue (BIQ)
  - Four-entry general instruction issue queue (GIQ)
- Branch unit—The branch unit (BU) is an execution unit and is distinct from the BPU. It executes (resolves) all branch and CR logical instructions.

- Two simple units (SU1 and SU2)
  — Add and subtract
  — Shift and rotate
  — Logical operations
  — Support for 64-bit SPE APU instructions in SU1
- Multiple-cycle unit (MU)—The MU is shown in Figure 1-3.

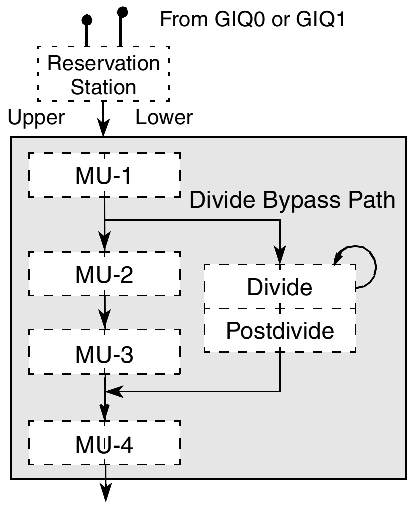


Figure 1-3. Four-Stage MU Pipeline, Showing Divide Bypass

The MU has the following features:

— Four-cycle latency for all multiplication, including SPE integer and fractional multiply instructions and embedded scalar and vector floating-point multiply instructions

— Variable-latency divide: 4, 11, 19, and 35 cycles for all integer divide instructions. If **r**A or **r**B is zero, floating-point divide instructions take 4 cycles; all others take 29. Note that although most divide instructions take more than 4 cycles to execute, the MU allows subsequent multiply instructions to execute through all four MU stages in parallel with the divide.

— 4-cycle floating-point add and subtract

- The load/store unit (LSU) is shown in Figure 1-4.

The LSU has the following features:

— 3-cycle load latency

— Fully pipelined

— Load miss queue allows up to four load misses before stalling (up to nine load misses in the e500v2).

— Load hits can continue to be serviced when the load miss queue is full.

— The seven-entry L1 store queue allows full pipelining of stores.

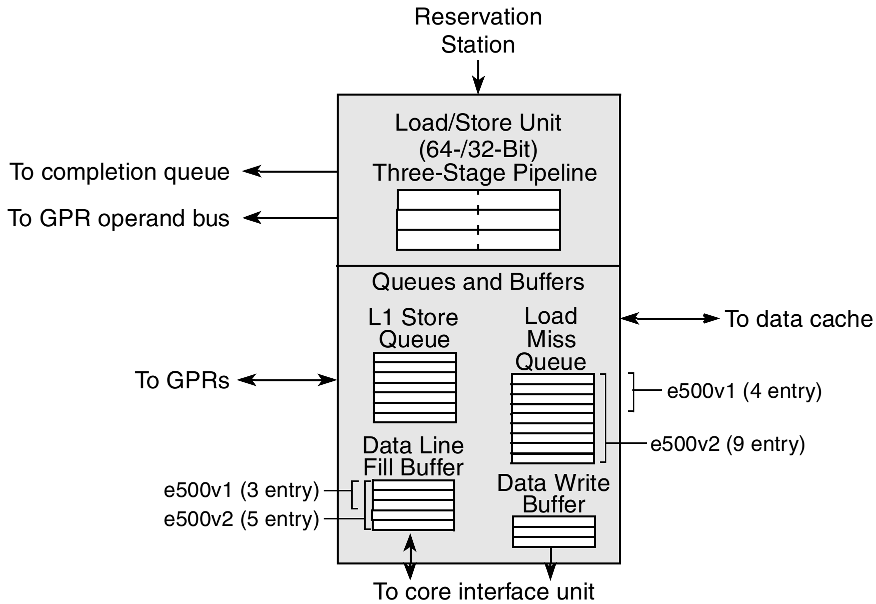


Figure 1-4. Three-Stage Load/Store Unit

- — The three-entry data line fill buffer (five-entry on the e500v2) is used for loads and cacheable stores. Stores are allocated here so loads can access data from the store immediately.
- — The data write buffer contains three entries: one dedicated for snoop pushes, one dedicated for castouts, and one that can be used for snoop pushes or cast outs.

- Cache coherency
  - — Supports four-state cache coherency: modified-exclusive, exclusive, shared, and invalid (MESI). Note, however that shared state may not be accessible in some implementations.
  - — Bus support for hardware-enforced coherency (bus snooping)
- Core complex bus (CCB)—internal bus
  - — High-speed, on-chip local bus with data tagging
  - — 32-bit address bus
  - — Address protocol with address pipelining and retry/copyback derived from bus used by previous generations of PowerPC processors (referred to as the 60x bus)
  - — Two general-purpose read data buses and one write data bus
- Extended exception handling
  - — Supports Book E interrupt model
    - – Less than 10-cycle interrupt latency
    - – Interrupt vector prefix register (IVPR)

  - Interrupt vector offset registers (IVORs) 0–15 as defined in Book E, plus e500-defined IVORs 32–35
  - Exception syndrome register (ESR)
  - Book E-defined preempting critical interrupt, including critical interrupt status registers (CSRR0 and CSRR1) and an **rfci** instruction
- — e500-specific interrupts not defined in Book E architecture
  - Machine-check APU
  - SPE APU unavailable exception
  - Floating-point data exception
  - Floating-point round exception
  - Performance monitor
- Memory management unit (MMU)
  - — 32-bit effective address translated to 32-bit real address (using a 41-bit interim virtual address) for the e500v1core and 36-bit real addressing for the e500v2 core
  - — TLB entries for variable- (4-Kbyte–256-Mbyte pages for the e500v1 and 4-Kbyte–4-Gbyte pages for the e500v2) and fixed-size (4-Kbyte) pages
  - — Data L1 MMU
    - 4-entry, fully associative TLB array for variable-sized pages
    - 64-entry, 4-way set-associative TLB for 4-Kbyte pages
  - — Instruction L1 MMU
    - 4-entry, fully associative TLB array for variable-sized pages
    - 64-entry, 4-way set-associative TLB for 4-Kbyte pages
  - — Unified L2 MMU
    - 16-entry, fully associative TLB array for variable-sized pages
    - e500v1—A 256-entry, 2-way set-associative unified (for instruction and data accesses) L2 TLB array (TLB0) supports only 4-Kbyte pages
    - e500v2—A 512-entry, 4-way set-associative unified (for instruction and data accesses) L2 TLB array (TLB0) supports only 4-Kbyte pages
  - — Software reload for TLBs
  - — Virtual memory support for as much as 4 Gbytes (2^{32}) of effective address space
  - — Real memory support for as much as 4 Gbytes (2^{32}) of physical memory on the e500v1 and 64 Gbytes (2^{36}) on the e500v2
  - — Support for big-endian and true little-endian memory on a per-page basis
- Power management

— Low-power design
— Power-saving modes: core-halted and core-stopped
— Internal clock multipliers ranging from 1 to 8 times the bus clock, including integer and half-mode multipliers.
— Dynamic power management of execution units, caches, and MMUs
— NAP, DOZE, and SLEEP bits in HID0 can be used to assert *nap*, *doze*, and *sleep* output signals to initiate power-saving modes at the integrated device level.

- Testability
  — LSSD scan design
  — JTAG interface
  — ESP support
  — Nexus debug support
- Reliability and serviceability
  — Parity checking on caches
  — Parity checking on e500 local bus

1.3.1 e500v2 Differences

The e500v2 provides the following additional features not supported by the e500v1:

- The e500v2 uses 36-bit physical addressing, which is supported by the following:
  — MMU assist register 7 (MAS7)
  — HID0[EN_MAS7_UPDATE]
  — Programmable jumper options to specify the upper bits of the reset vector.
- The e500v2 has a 512-entry, 4-way set-associative unified TLB for TLB1.
- The maximum variable page size is extended to 4 Gbytes.
- Embedded double-precision floating-point APU has been added. These instructions use the 64-bit GPRs as single, 64-bit double-precision operands. This APU is enabled through MSR[SPE].
- Slightly different functionality of HID1[RFXE] bit.
- The data line fill buffer in the LSU is expanded from three to five entries.
- The load miss queue in the LSU is expanded from four to nine entries.
- TBSEL and TBEE bits have been added to the performance monitor global control register 0 (PMGC0) to support monitoring of time base events.
- Minor modifications to the SPE APU.

- Data cache flush assist capability, supported through HID0[DCFA]. When DCFA is set, the cache miss replacement algorithm ignores invalid entries and follows the replacement sequence defined by the PLRU bits. This reduces the series of uniquely addressed load or **dcbz** instructions required to flush the cache.

Detailed descriptions of these differences are provided in their respective chapters.

NOTE

Unless otherwise indicated, references to e500 apply to both e500v1 and e500v2.

1.4 Instruction Set

The e500 implements the following instructions:

- The Book E instruction set for 32-bit implementations. This is composed primarily of the user-level instructions defined by the PowerPC user instruction set architecture (UISA). The e500 does not include Book E floating-point, load string, or store string instructions.
- The e500 supports the following implementation-specific instructions:
  — Integer select APU. This APU consists of the Integer Select instruction (**isel**), which functions as an if-then-else statement that selects between two source registers by comparison to a CR bit. This instruction eliminates conditional branches, decreases latency, and reduces the code footprint.

  — Performance monitor APU. Table 1-2 lists performance monitor APU instructions.

Table 1-2. Performance Monitor APU Instructions

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

  — Cache locking APU. This APU consists of the instructions described in Table 1-3.

Table 1-3. Cache Locking APU Instructions

Name	Mnemonic	Syntax
Data Cache Block Lock Clear	**dcblc**	CT, **r**A, **r**B
Data Cache Block Touch and Lock Set	**dcbtls**	CT, **r**A, **r**B
Data Cache Block Touch for Store and Lock Set	**dcbtstls**	CT, **r**A, **r**B
Instruction Cache Block Lock Clear	**icblc**	CT, **r**A, **r**B
Instruction Cache Block Touch and Lock Set	**icbtls**	CT, **r**A, **r**B

  — Machine check APU. This APU defines the Return from Machine Check Interrupt instruction (**rfmci**).

— SPE APU vector instructions. Vector instructions are defined that view the 64-bit GPRs as composed of a vector of two 32-bit elements (some instructions also read or write 16-bit elements). Some scalar instructions produce a 64-bit scalar result. Section 3.8.1.3, "SPE APU Instructions," lists SPE APU vector instructions.

— The embedded floating-point APUs provide scalar and vector floating-point instructions. Scalar single-precision floating-point instructions use only the lower 32 bits of the GPRs; double-precision operands (e500v2 only) use all 64 bits. Table 1-4 lists embedded floating-point instructions.

Table 1-4. Scalar and Vector Embedded Floating-Point APU Instructions

Instruction	Mnemonic			Syntax
	Scalar SP	Scalar DP	Vector	
Convert Floating-Point Single- from Double-Precision	—	**efscfd**	—	**r**D,**r**B
Convert Floating-Point Double- from Single-Precision	—	**efdcfs**	—	**r**D,**r**B
Convert Floating-Point from Signed Fraction	**efscfsf**	**efdcfsf**	**evfscfsf**	**r**D,**r**B
Convert Floating-Point from Signed Fraction	**efscfsf**	**efdcfsf**	**evfscfsf**	**r**D,**r**B
Convert Floating-Point from Signed Integer	**efscfsi**	**efdcfsi**	**evfscfsi**	**r**D,**r**B
Convert Floating-Point from Unsigned Fraction	**efscfuf**	**efdcfuf**	**evfscfuf**	**r**D,**r**B
Convert Floating-Point from Unsigned Integer	**efscfui**	**efdcfui**	**evfscfui**	**r**D,**r**B
Convert Floating-Point to Signed Fraction	**efsctsf**	**efdctsf**	**evfsctsf**	**r**D,**r**B
Convert Floating-Point to Signed Integer	**efsctsi**	**efdctsi**	**evfsctsi**	**r**D,**r**B
Convert Floating-Point to Signed Integer with Round toward Zero	**efsctsiz**	**efdctsiz**	**evfsctsiz**	**r**D,**r**B
Convert Floating-Point to Unsigned Fraction	**efsctuf**	**efdctuf**	**evfsctuf**	**r**D,**r**B
Convert Floating-Point to Unsigned Integer	**efsctui**	**efdctui**	**evfsctui**	**r**D,**r**B
Convert Floating-Point to Unsigned Integer with Round toward Zero	**efsctuiz**	**efdctuiz**	**evfsctuiz**	**r**D,**r**B
Floating-Point Absolute Value	**efsabs**	**efdabs**	**evfsabs**	**r**D,**r**A
Floating-Point Add	**efsadd**	**efdadd**	**evfsadd**	**r**D,**r**A,**r**B
Floating-Point Compare Equal	**efscmpeq**	**efdcmpeq**	**evfscmpeq**	**cr**D,**r**A,**r**B
Floating-Point Compare Greater Than	**efscmpgt**	**efdcmpgt**	**evfscmpgt**	**cr**D,**r**A,**r**B
Floating-Point Compare Less Than	**efscmplt**	**efdcmplt**	**evfscmplt**	**cr**D,**r**A,**r**B
Floating-Point Divide	**efsdiv**	**efddiv**	**evfsdiv**	**r**D,**r**A,**r**B
Floating-Point Multiply	**efsmul**	**efdmul**	**evfsmul**	**r**D,**r**A,**r**B
Floating-Point Negate	**efsneg**	**efdneg**	**evfsneg**	**r**D,**r**A
Floating-Point Negative Absolute Value	**efsnabs**	**efdnabs**	**evfsnabs**	**r**D,**r**A
Floating-Point Subtract	**efssub**	**efdsub**	**evfssub**	**r**D,**r**A,**r**B
Floating-Point Test Equal	**efststeq**	**efdtsteq**	**evfststeq**	**cr**D,**r**A,**r**B
Floating-Point Test Greater Than	**efststgt**	**efdtstgt**	**evfststgt**	**cr**D,**r**A,**r**B
Floating-Point Test Less Than	**efststlt**	**efdtstlt**	**evfststlt**	**cr**D,**r**A,**r**B

— BTB locking APU instructions. The core complex provides a 512-entry BTB for efficient processing of branch instructions. The BTB is a branch target address cache, organized as 128 rows with 4-way set associativity, that holds the address and target instruction of the 512 most-recently taken branches. Table 1-5 lists BTB instructions.

Table 1-5. BTB Locking APU Instructions

Name	Mnemonic	Syntax
Branch Buffer Load Entry and Lock Set	**bblels**	—
Branch Buffer Entry Lock Reset	**bbelr**	—

1.5 Instruction Flow

The e500 core is a pipelined, superscalar processor with parallel execution units that allow instructions to execute out of order but record their results in order. Pipelining breaks instruction processing into discrete stages, so multiple instructions in an instruction sequence can occupy the successive stages: as an instruction completes one stage, it passes to the next, leaving the previous stage available to a subsequent instruction. So, even though it may take multiple cycles for an instruction to pass through all of the pipeline stages, once a pipeline is full, instruction throughput is much shorter than the latency.

A superscalar processor is one that issues multiple independent instructions into separate execution units, allowing parallel execution. The e500 core has five execution units, one each for branch (BU), load/store (LSU), and multiple-cycle operations (MU), and two for simple arithmetic operations (SU1 and SU2). The MU and SU1 arithmetic execution units also execute 64-bit SPE vector instructions, using both the lower and upper halves of the 64-bit GPRs.

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 SU after an integer divide is issued to the MU should finish executing before the higher latency divide instruction. The add instruction can make its results available to a subsequent instruction, but it cannot update the architected GPR specified as its target operand ahead of the multiple-cycle divide instruction.

1.5.1 Initial Instruction Fetch

The e500 core begins execution at fixed virtual address 0xFFFF_FFFC. The MMU has a default page translation which maps this to the identical physical address. So, the instruction at physical address 0xFFFF_FFFC must be a branch to another address within the 4-Kbyte boot page.

1.5.2 Branch Detection and Prediction

To improve branch performance, the e500 provides implementation-specific dynamic branch prediction using the BTB to resolve branch instructions and improve the accuracy of branch predictions. Each of the 512 entries in the 4-way set associative address cache of branch target addresses includes a 2-bit saturating branch history counter, whose value is incremented or decremented depending on whether the branch was taken. These bits can take on four values indicating strongly taken, weakly taken, weakly not taken, and strongly not taken. The BTB is used not only to predict branches, but to detect branches during the fetch stage, offering an efficient way to access instruction streams for branches predicted as taken.

In the e500, all branch instructions are assigned positions in the completion queue at dispatch. Speculative instructions in branch target streams are allowed to execute and proceed through the completion queue, although they can complete only after the branch prediction is resolved as correct and after the branch instruction itself completes.

If a branch resolves as correct, instructions in the target stream are marked nonspeculative and are allowed to complete. If the branch history bits in the BTB indicated weakly taken or weakly not taken, the prediction is upgraded to strongly taken or strongly not taken.

If a branch resolves as incorrect, instructions in the target stream are flushed from the execution pipeline, the branch history bits are updated in the BTB entry, and nonspeculative fetching begins from the correct path.

1.5.3 e500 Execution Pipeline

The seven stages of the e500 execution pipeline—fetch1, fetch2/predecode, decode/dispatch, issue, execute, complete, and write back—are highlighted in grey in Figure 1-5.

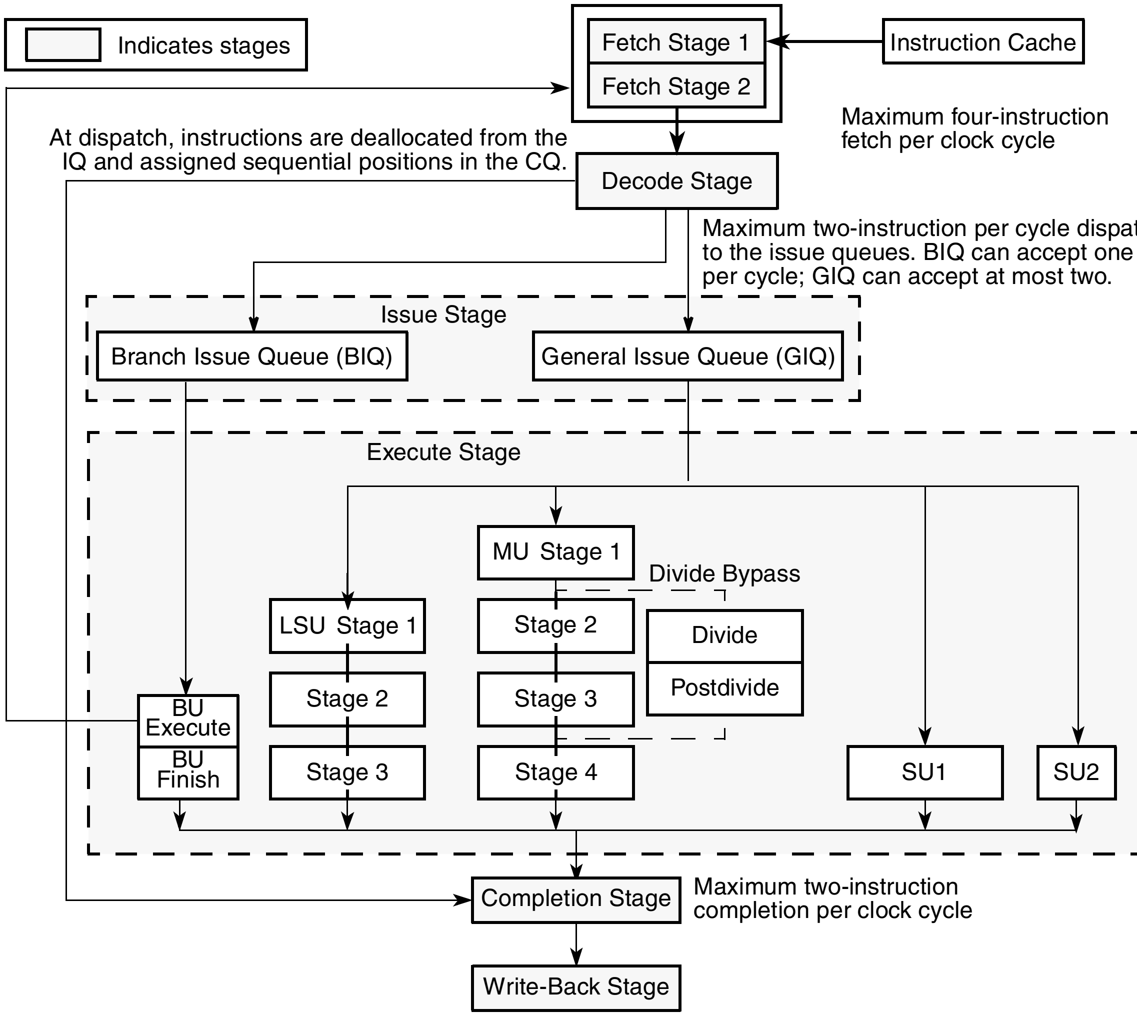


Figure 1-5. Instruction Pipeline Flow

The common pipeline stages are as follows:

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

  Instruction fetch timing depends on many variables, such as whether an instruction is in the on-chip instruction cache or an L2 cache (if implemented). Those factors increase when it is necessary to fetch instructions from system memory and include the processor-to-bus clock ratio, the amount of bus traffic, and whether any cache coherency operations are required.

Because there are so many variables, unless otherwise specified, the instruction timing examples in this chapter assume optimal performance and show the portion of the fetch stage in which the instruction is in the instruction queue. The fetch1 and fetch2 stages are primarily involved in retrieving instructions.

- The decode/dispatch stage fully decodes each instruction; most instructions are dispatched to the issue queues (however, **isync**, **rfi**, **sc**, **nop**s, and some other instructions do not go to issue queues).
- The two issue queues, BIQ and GIQ, can accept as many as one and two instructions, respectively, in a cycle. The behavior of instruction dispatch is covered in significant detail in the *e500 Software Optimization Guide*. 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.
  - — Space must be available in the CQ for an instruction to decode and dispatch (this includes instructions that are assigned a space in the CQ but not in an issue queue).

Dispatch is treated as an event at the end of the decode stage. The issue stage reads source operands from rename registers and register files and determines when instructions are latched into the execution unit reservation stations. Note that the e500 has 14 rename registers, one for each completion queue entry, so instructions cannot stall because of a shortage of rename registers.

The general behavior of the two issue queues is described as follows:

— The GIQ accepts as many as two instructions from the dispatch unit per cycle. SU1, SU2, MU, and all LSU instructions (including 64-bit loads and stores) are dispatched to the GIQ, shown in Figure 1-6.

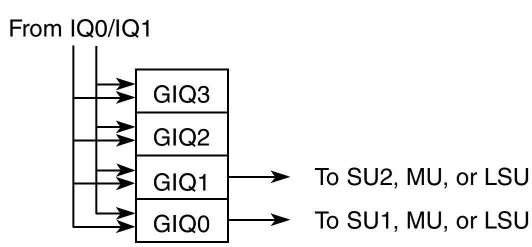


Figure 1-6. GPR Issue Queue (GIQ)

Instructions can be issued out-of-order from the bottom two GIQ entries (GIQ1–GIQ0). GIQ0 can issue to SU1, MU, and LSU. GIQ1 can issue to SU2, MU, and LSU.

Note that SU2 executes a subset of the instructions that can be executed in SU1. The ability to identify and dispatch instructions to SU2 increases the availability of SU1 to execute more computational-intensive instructions.

An instruction in GIQ1 destined for SU2 or the LSU need not wait for an MU instruction in GIQ0 that is stalled behind a long-latency divide.

- The execute stage accepts instructions from its issue queue when the appropriate reservation stations are not busy. In this stage, the operands assigned to the execution stage from the issue stage are latched.

  The execution unit executes the instruction (perhaps over multiple cycles), writes results on its result bus, and notifies the CQ when the instruction finishes. The execution unit reports any exceptions to the completion stage. Instruction-generated exceptions are not taken until the excepting instruction is next to retire.

  Most integer instructions have a 1-cycle latency, so results of these instructions are available 1 clock cycle after an instruction enters the execution unit. The MU and LSU are pipelined, as shown in Figure 1-5.

  Branches resolve in execute stage. If a branch is mispredicted, it takes 5 cycles for the next instruction to reach the execute stage.
- The complete and write-back stages maintain the correct architectural machine state and commit results to the architecture-defined registers in the proper order. If completion logic detects an instruction containing an exception status or a mispredicted branch, all following instructions are cancelled, their execution results in rename registers are discarded, and the correct instruction stream is fetched.

  The complete stage ends when the instruction is retired. Two instructions can be retired per clock cycle. If no dependencies exist, as many as two instructions are retired in program order. Section 4.7.4, "Completion Unit Resource Requirements," describes completion dependencies.

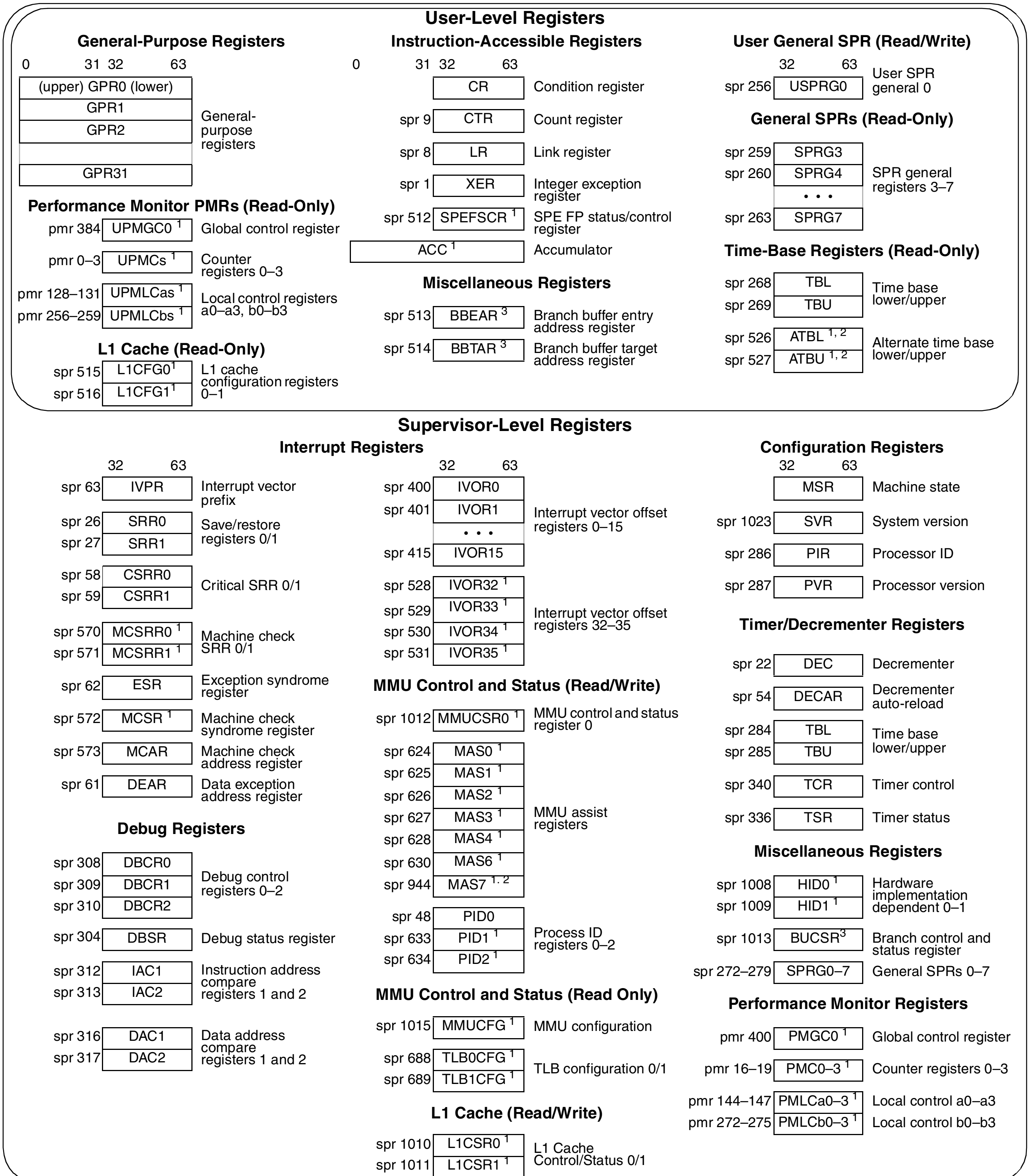
  The write-back stage occurs in the clock cycle after the instruction is retired.

The e500 core also provides new instructions that perform single-instruction, multiple-data (SIMD) operations. These signal processing instructions consist of parallel operations on both the upper and lower 32 bits of two 64-bit GPR values and produce two 32-bit results written to a 64-bit GPR.

As shown in Figure 1-5, the LSU, MU, and SU1 replicate logic to support 64-bit operations. Although a vector instruction generates separate, discrete results in the upper and lower halves of the target GPR, latency and throughput for vector instructions are the same as those for their scalar equivalents.

1.6 Programming Model

The following section describes the e500 core registers defined in Book E, the Freescale Semiconductor Book E implementation standards (EIS), and registers that are specific to the e500. Figure 1-7 shows the e500 register set.



1 These registers are defined by the EIS
2 e500v2 only
3 These registers are e500-specific

Figure 1-7. e500 Core Programming Model

1.7 On-Chip Cache Implementation

The core complex contains separate 32-Kbyte, eight-way set-associative, level 1 (L1) instruction and data caches to give rapid access to instructions and data.

The data cache supports four-state MESI memory coherency protocol. The core complex broadcasts all cache management functions based on the setting of the address broadcast enable bit, HID1[ABE], allowing management of other caches in the system.

The caches implement a pseudo-least-recently-used (PLRU) replacement algorithm.

Parity generation and checking may be enabled for both caches, and each cache can be independently invalidated through L1CSR1 and L1CSR0. Additionally, instructions are provided to perform cache locking and unlocking on both data and instruction caches on a cache-block granularity. These are listed in Section 1.10.3, “Cache Control Instructions.”

Individual instruction cache blocks and data cache blocks can be invalidated using the **icbi** and **dcbi** instructions, respectively. The entire data cache can be invalidated by setting L1CSR0[CFI]; the entire instruction cache can be invalidated by setting L1CSR1[ICFI].

1.8 Interrupts and Exception Handling

The e500 core supports an extended exception handling model, with nested interrupt capability and extensive interrupt vector programmability. The following sections define the exception model, including an overview of exception handling as implemented on the e500 core, a brief description of the exception classes, and an overview of the registers involved in the processes.

1.8.1 Exception Handling

In general, interrupt processing begins with an exception that occurs due to external conditions, errors, or program execution problems. When the exception occurs, the processor checks to verify interrupt processing is enabled for that particular exception. If enabled, the interrupt causes the state of the processor to be saved in the appropriate registers and prepares to begin execution of the handler located at the associated vector address for that particular exception.

Once the handler is executing, the implementation may need to check one or more bits in the exception syndrome register (ESR) or the SPEFSCR, depending on the exception, to verify the specific cause of the exception and take appropriate action.

The core complex provides the interrupts described in Section 1.8.5, “Interrupt Registers.”

1.8.2 Interrupt Classes

All interrupts may be categorized as asynchronous/synchronous and critical/noncritical.

- Asynchronous interrupts (such as machine check, critical input, and external interrupts) are caused by events that are independent of instruction execution. For asynchronous interrupts, the address reported in a save/restore register is the address of the instruction that would have executed next had the asynchronous interrupt not occurred.
- Synchronous interrupts are those that are caused directly by the execution or attempted execution of instructions. Synchronous inputs may be either precise or imprecise, which are described as follows:
  - Synchronous precise interrupts are those that precisely indicate the address of the instruction causing the exception that generated the interrupt or, in some cases, the address of the immediately following instruction. The interrupt type and status bits indicate which instruction is addressed in the appropriate save/restore register.
  - Synchronous imprecise interrupts are those that may indicate the address of the instruction causing the exception that generated the interrupt or some instruction after the instruction causing the interrupt. If the interrupt was caused by either the context synchronizing mechanism or the execution synchronizing mechanism, the address in the appropriate save/restore register is the address of the interrupt forcing instruction. If the interrupt was not caused by either of those mechanisms, the address in the save/restore register is the last instruction to start execution and may not have completed. No instruction following the instruction in the save/restore register has executed.

1.8.3 Interrupt Types

The e500 core processes all interrupts as either machine check, critical, or noncritical types. Separate control and status register sets are provided for each interrupt type. The core handles interrupts from these three types in the following priority order:

1. Machine check interrupt (highest priority)—The e500 defines a separate set of resources for the machine check interrupt. They use the machine check save and restore registers (MCSRR0/MCSRR1) to save state when they are taken, and they use the **rfmci** instruction to restore state. These interrupts can be masked by the machine check enable bit, MSR[ME].
2. Noncritical interrupts—First-level interrupts that allow the processor to change program flow to handle conditions generated by external signals, errors, or unusual conditions arising from program execution or from programmable timer-related events. These interrupts are largely identical to those previously defined by the OEA portion of the Power PC architecture. They use save and restore registers (SRR0/SRR1) to save state when they are taken and they use the **rfi** instruction to restore state. Asynchronous noncritical interrupts can be masked by the external interrupt enable bit, MSR[EE].

3. Critical interrupts—Critical interrupts can be taken during a noncritical interrupt or during regular program flow. They use the critical save and restore registers (CSRR0/CSRR1) to save state when they are taken and they use the **rfci** instruction to restore state. These interrupts can be masked by the critical enable bit, MSR[CE]. Book E defines the critical input, watchdog timer, and machine check interrupts as critical interrupts, but the e500 defines a third set of resources for the machine check interrupt, as described in Table 1-6.

All interrupts except machine check are ordered within the two categories of noncritical and critical, such that only one interrupt of each category is reported, and when it is processed (taken), no program state is lost. Because save/restore register pairs are serially reusable, program state may be lost when an unordered interrupt is taken (see Section 5.10, "Interrupt Ordering and Masking").

1.8.4 Upper Bound on Interrupt Latencies

Core complex interrupt latency is defined as the number of core clocks between the sampling of the interrupt signal as asserted and the initiation of the IVOR fetch (that is, the fetch of the first instruction in the handler). Core complex interrupt latency is determinate unless a guarded load or a cache-inhibited **stwcx.** is being executed, in which case the latency is indeterminate. The minimum latency is 3 core clocks and the maximum is 8, not including the 2 bus clock cycles required to synchronize the interrupt signal from the pad.

When an interrupt is taken, all instructions in the IQ are thrown away unless the oldest instruction is a load/store instruction. That is, if an asynchronous interrupt is being serviced and the oldest instruction is not a load/store instruction, the core complex goes straight from sampling the interrupt to ensuring a recoverable state and issuing an exception. If a load/store instruction is oldest, the core complex waits 4 clocks before ensuring a recoverable state. During this time, any instruction finished by the LSU is deallocated.

1.8.5 Interrupt Registers

The registers associated with interrupt and exception handling are described in Table 1-6.

Table 1-6. Interrupt Registers

Register	Description
Noncritical Interrupt Registers	
SRR0	Save/restore register 0—Holds the address of the instruction causing the exception or the address of the instruction that will execute after the **rfi** instruction.
SRR1	Save/restore register 1—Holds machine state on noncritical interrupts and restores machine state after an **rfi** instruction is executed.
Critical Interrupt Registers	
CSRR0	Critical save/restore register 0—On critical interrupts, holds either the address of the instruction causing the exception or the address of the instruction that will execute after the **rfci** instruction.
CSRR1	Critical save/restore register 1—Holds machine state on critical interrupts and restores machine state after an **rfci** instruction is executed.

Table 1-6. Interrupt Registers (continued)

Register	Description
Machine Check Interrupt Registers	
MCSRR0	Machine check save/restore register 0—Used to store the address of the instruction that will execute after an **rfmci** instruction is executed.
MCSRR1	Machine check save/restore register 1—Holds machine state on machine check interrupts and restores machine state (if recoverable) after an **rfmci** instruction is executed.
MCAR	Machine check address register—Holds the address of the data or instruction that caused the machine check interrupt. MCAR contents are not meaningful if a signal triggered the machine check interrupt.
Syndrome Registers	
MCSR	Machine check syndrome register—Holds machine state information on machine check interrupts and restores machine state after an **rfmci** instruction is executed.
ESR	Exception syndrome register—Provides a syndrome to differentiate between the different kinds of exceptions that generate the same interrupt type. Upon generation of a specific exception type, the associated bit is set and all other bits are cleared.
SPE APU Interrupt Registers	
SPEFSCR	Signal processing and embedded floating-point status and control register—Provides interrupt control and status as well as various condition bits associated with the operations performed by the SPE APU.
Other Interrupt Registers	
DEAR	Data exception address register—Holds the address that was referenced by a load, store, or cache management instruction that caused an alignment, data TLB miss, or data storage interrupt.
IVPR IVORs	Together, IVPR[32–47] || IVOR*n* [48–59] || 0b0000 define the address of an interrupt-processing routine. See Table 1-7 and the EREF for more information.

Each interrupt has an associated interrupt vector address, obtained by concatenating the IVPR value with the address index in the associated IVOR (that is, IVPR[32–47] || IVOR*n*[48–59] || 0b0000). The resulting address is that of the instruction to be executed when that interrupt occurs. IVPR and IVOR values are indeterminate on reset, and must be initialized by the system software using **mtspr**. Table 1-7 lists IVOR registers implemented on the e500 and the associated interrupts. For more information, see Chapter 5, “Interrupts and Exceptions.”

Table 1-7. Interrupt Vector Registers and Exception Conditions

Register	Interrupt
Book E–Defined IVORs	
IVOR0	Critical input
IVOR1	Machine check interrupt offset
IVOR2	Data storage interrupt offset
IVOR3	Instruction storage interrupt offset
IVOR4	External input interrupt offset
IVOR5	Alignment interrupt offset
IVOR6	Program interrupt offset
IVOR7	Floating-point unavailable interrupt offset (not supported on the e500)
IVOR8	System call interrupt offset

Table 1-7. Interrupt Vector Registers and Exception Conditions (continued)

Register	Interrupt
IVOR9	Auxiliary processor unavailable interrupt offset (not supported on the e500)
IVOR10	Decrementer interrupt offset
IVOR11	Fixed-interval timer interrupt offset
IVOR12	Watchdog timer interrupt offset
IVOR13	Data TLB error interrupt offset
IVOR14	Instruction TLB error interrupt offset
IVOR15	Debug interrupt offset
e500-Specific IVORs	
IVOR32	SPE APU unavailable interrupt offset
IVOR33	SPE floating-point data exception interrupt offset
IVOR34	SPE floating-point round exception interrupt offset
IVOR35	Performance monitor

1.9 Memory Management

The e500 core complex supports demand-paged virtual memory as well other memory management schemes that depend on precise control of effective-to-physical address translation and flexible memory protection as defined by Book E. The mapping mechanism consists of software-managed TLBs that support variable-sized pages with per-page properties and permissions. The following properties can be configured for each TLB:

- User-mode page execute access
- User-mode page read access
- User-mode page write access
- Supervisor-mode page execute access
- Supervisor-mode page read access
- Supervisor-mode page write access
- Write-through required (W)
- Caching inhibited (I)
- Memory coherency required (M)
- Guarded (G)
- Endianness (E)
- User-definable (U0–U3), a 4-bit implementation-specific field

The core complex employs a two-level memory management unit (MMU) architecture. There are separate instruction and data level-1 (L1) MMUs backed up by a unified level-2 (L2) MMU, This two-level structure is shown in Figure 1-8.

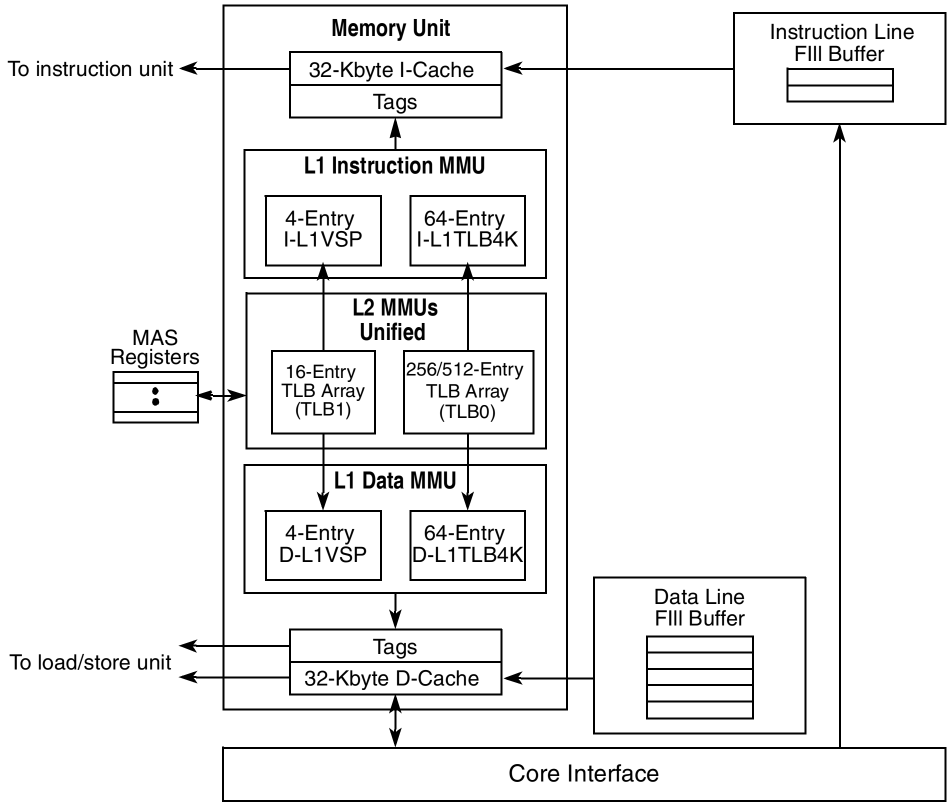


Figure 1-8. MMU Structure

Level-1 MMUs have the following features:

- Four-entry, fully associative TLB array that supports all nine page sizes
- 64-entry, 4-way set-associative TLB 4-Kbyte array that supports 4-Kbyte pages only
- Hardware partially managed by L2 MMU
- Supports snooping of TLBs by both internal and external **tlbivax** instructions

The level-2 MMU has the following features:

- A 16-entry, fully associative L2 TLB array (TLB1) that supports all nine variable page sizes
- TLB array (TLB0) that supports only 4-Kbyte pages, as follows:
  — e500v1—256-entry, 2-way set-associative TLB array
  — e500v2—512-entry, 4-way set-associative TLB array
- Hardware assist for TLB miss exceptions
- Software managed by **tlbre**, **tlbwe**, **tlbsx**, **tlbsync**, **tlbivax**, and **mtspr** instructions
- Supports snooping of TLB by both internal and external **tlbivax** instructions

1.9.1 Address Translation

The core complex fetch and load/store units generate 32-bit effective addresses. The MMU translates these addresses to real addresses (32-bit real addresses for the e500v1 core, 36-bit for the e500v2) (which are used for memory bus accesses) using an interim 41-bit virtual address.

Figure 1-9 shows the translation flow for the e500v1 core.

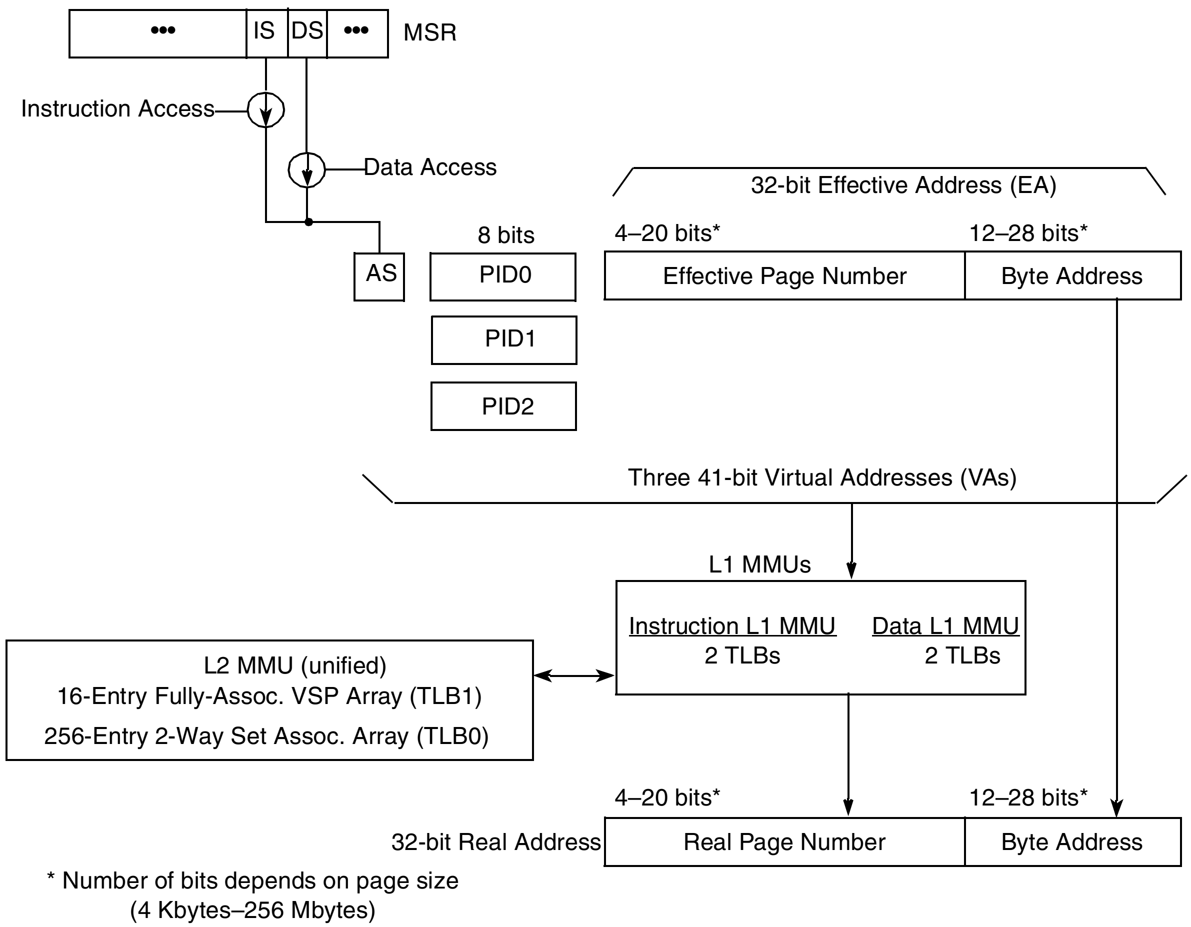


Figure 1-9. Effective-to-Real Address Translation Flow

Figure 1-10 shows the same translation flow for the e500v2 core.

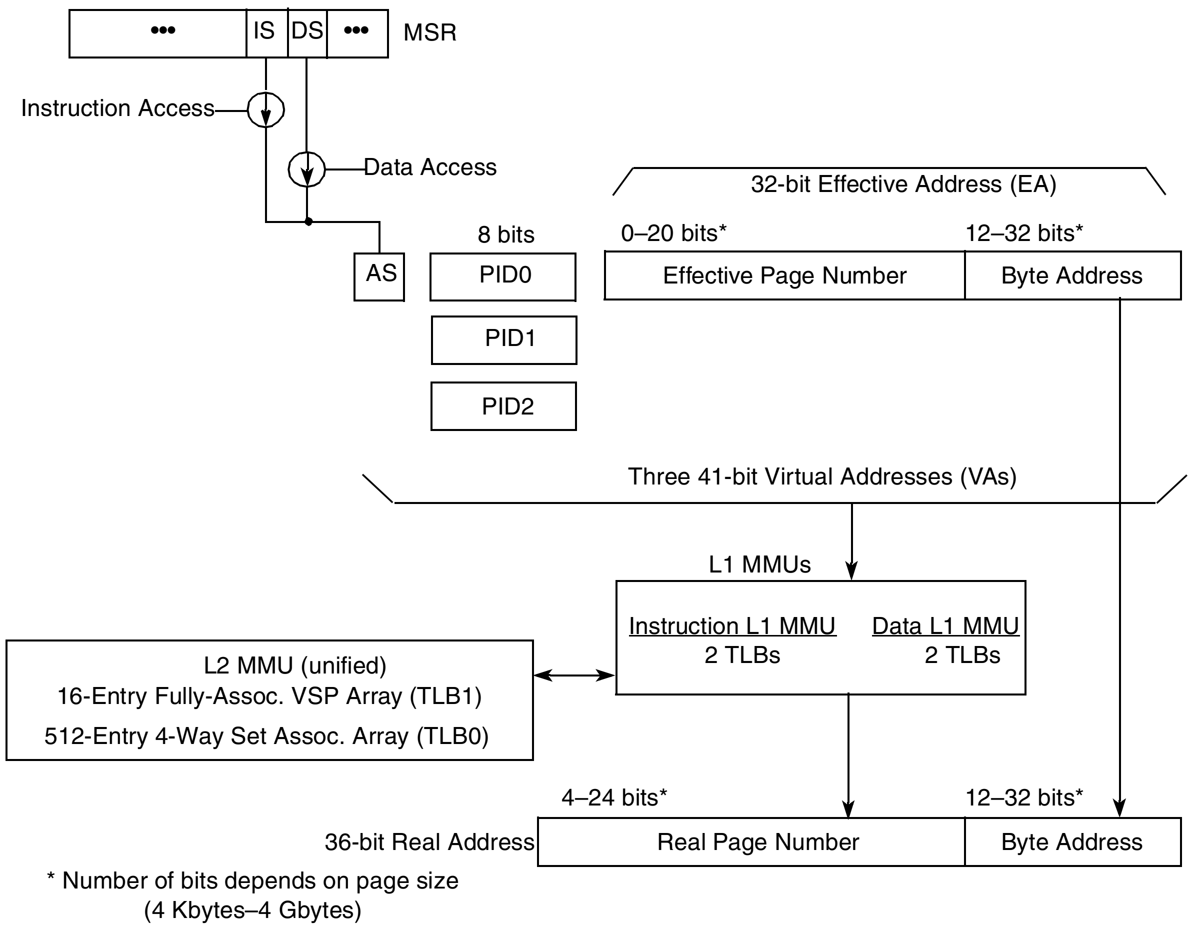


Figure 1-10. Effective-to-Real Address Translation Flow (e500v2)

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 byte address to obtain the physical address of the requested access. On misses, the L1 TLB entries are replaced from their L2 TLB counterparts using a true LRU algorithm.

1.9.2 MMU Assist Registers (MAS0–MAS4 and MAS6–MAS7)

Book E defines SPR numbers for the MMU assist registers, which are used to hold values either read from or to be written to the TLBs and information required to identify the TLB to be accessed. To ensure consistency among Freescale Semiconductor Book E processors, certain aspects of the implementation are defined by the Freescale Semiconductor Book E standard, whereas more specific details are left to individual implementations. MAS3 implements the real page number

(RPN), the user attribute bits (U0–U3), and permission bits (UX, SX, UW, SW, UR, SR) that specify user and supervisor read, write, and execute permissions.

The e500 does not implement MAS5.

MAS registers are affected by the following instructions (see Section 12.4, “TLB Instructions—Implementation,” for more detailed information):

- MAS registers are accessed with the **mtspr** and **mfspr** instructions.
- The TLB Read Entry instruction (**tlbre**) causes the contents of a single TLB entry from the L2 MMU to be placed in defined locations in MAS0–MAS3 (and optionally MAS7 on the e500v2). The TLB entry to be extracted is determined by information written to MAS0 and MAS2 before the **tlbre** instruction is executed.
- The TLB Write Entry instruction (**tlbwe**) causes the information stored in certain locations of MAS0–MAS3 (and MAS7 on the e500v2) to be written to the TLB specified in MAS0.
- The TLB Search Indexed instruction (**tlbsx**) updates MAS registers conditionally, based on success or failure of a lookup in the L2 MMU. The lookup is specified by the instruction encoding and specific search fields in MAS6. The values placed in the MAS registers may differ, depending on a successful or unsuccessful search.

For TLB miss and certain MMU-related DSI/ISI exceptions, MAS4 provides default values for updating MAS0–MAS2.

1.9.3 Process ID Registers (PID0–PID2)

The e500 core complex also implements three process ID (PID) registers that hold the values used to construct the three virtual addresses for each access. These process IDs provide an extended page sharing capability. Which of these three virtual addresses is used is controlled by the TID field of a matching TLB entry, and when TID = 0x00 (identifying a page as globally shared), the PID values are ignored.

A hit to multiple TLB entries in the L1 MMU (even if they are in separate arrays) or a hit to multiple entries in the L2 MMU is considered to be a programming error.

1.9.4 TLB Coherency

The core complex provides the ability to invalidate a TLB entry, as defined in the Book E architecture. The **tlbivax** instruction invalidates a matching local TLB entry. Execution of this instruction is also broadcast on the core complex bus (CCB) if HID1[ABE] is set. The core complex also snoops TLB invalidate transactions on the CCB from other bus masters.

1.10 Memory Coherency

The core complex supports four-state memory coherency. Memory coherency is hardware-supported on the system bus through bus snooping and the retry/copyback bus protocol, and through broadcasting of cache management instructions. Translation coherency is also hardware-supported through broadcasting and bus snooping of TLB invalidate transactions. The four-state MESI protocol supports efficient large-scale real-time data sharing between multiple caching bus masters.

1.10.1 Atomic Update Memory References

The e500 core supports atomic update memory references for both aligned word forms of data using the load and reserve and store conditional instruction pair, **lwarx** and **stwcx.**. Typically, a load and reserve instruction establishes a reservation and is paired with a store conditional instruction to achieve the atomic operation. However, there are restrictions and requirements for this functionality. The processor revokes reservations during a context switch, so the programmer must reacquire the reservation after a context switch occurs.

1.10.2 Memory Access Ordering

The core complex supports weakly ordered references to memory. Thus the e500 manages the order and synchronization of instructions to ensure proper execution when memory is shared between multiple processes or programs. The cache and data memory control attributes, along with **msync** and **mbar**, provide the required access control; **msync** and **mbar** are also broadcast on the CCB to provide the appropriate control in the case of multiprocessor or shared memory systems.

1.10.3 Cache Control Instructions

The core complex supports Book E instructions for performing a full range of cache control functions, including cache locking by line. The core complex supports broadcasting and snooping of these cache control instructions on the CCB. The e500 core also supports the following e500-specific cache locking instructions:

- Data Cache Block Lock Clear (**dcblc**)
- Data Cache Block Touch and Lock Set (**dcbtls**)
- Data Cache Block Touch for Store and Lock Set (**dcbtstls**)
- Instruction Cache Block Lock Clear (**icblc**)
- Instruction Cache Block Touch and Lock Set (**icbtls**)

1.10.4 Programmable Page Characteristics

Cache and memory attributes are programmable on a per-page basis. In addition to the write-through, caching-inhibited, memory coherency enforced, and guarded characteristics defined by the WIMG bits, Book E defines an endianness bit, E, that allows selection of big- or little-endian byte ordering on a per-page basis.

In addition to the WIMGE bits, the Book E MMU model defines user-definable page attribute bits (U0–U3).

1.11 Core Complex Bus (CCB)

The core complex defines a versatile local bus interface that allows a wide range of system performance and system-complexity trade-offs. The interface defines the following buses.

- An address-out bus for mastering bus transactions
- An address-in bus for snooping internal resources
- Three tagged data buses

Two of the data buses are general-purpose data-in buses for reads, and the third is a data-out bus for writes. The two data-in buses feature support for out-of-order read transactions from two different sources simultaneously, and all three data buses may be operated concurrently. The address-in bus supports snooping for external management of the L1 caches and TLBs by other bus masters. The core complex broadcasts and snoops the cache and TLB management instructions accordingly. It is envisioned that a wide range of system implementations can be constructed from the defined interface.

1.12 Performance Monitoring

The e500 core provides a performance monitoring capability that allows counting of events such as processor clocks, instruction cache misses, data cache misses, mispredicted branches, and others. The count of these events may be configured to trigger a performance monitor exception following the e500 interrupt model. This interrupt is assigned to vector offset register IVOR35.

The register set associated with the performance monitoring function consists of counter registers, a global control register, and local control registers. These registers are read/write from supervisor mode, and each register is reflected to a corresponding read-only register for user mode. Two instructions, **mtpmr** and **mfpmr**, are provided for moving data to and from these registers. An overview of the performance monitoring registers is provided in the following sections.

1.12.1 Global Control Register

The PMGC0 register provides global control of the performance monitoring facility from supervisor mode. From this register all counters may be frozen, unfrozen, or configured to freeze on an enabled condition or event. Additionally, the performance monitoring facility may be disabled or enabled from this register. The contents of PMGC0 are reflected to UPMGC0, which may be read from user mode using the **mfpmr** instruction.

1.12.2 Performance Monitor Counter Registers

There are four counter registers (PCM0–PCM3) provided in the performance monitoring facility. These 32-bit registers hold the current count for software-selectable events and can be programmed to generate an exception on overflow. These registers may be written or read from supervisor mode using the **mtpmr** and **mfpmr** instructions. The contents of these registers are reflected to UPCM0–UPCM3, which can be read from user mode with **mfpmr**.

Performance monitor exceptions occur only if all of the following conditions are met:

- A counter is in the overflow state.
- The counter's overflow signaling is enabled.
- Overflow exception generation is enabled in PMGC0.
- MSR[EE] is set.

1.12.3 Local Control Registers

For each of the counter registers, there are two corresponding local control registers. These two registers specify which of the 128 available events is to be counted, what specific action is to be taken on overflow, and various options for freezing a counter value under given modes or conditions.

- PMLCa0–PMLCa3 provide fields that allow freezing of the corresponding counter in user mode, supervisor mode, or under software control. Additionally, the overflow condition may be enabled or disabled from this register. The contents of these registers are reflected to UPMLCa0–UPMLCa3, which can be read from user mode with **mfpmr**.
- PMLCb0–PMLCb3 provide count scaling for each counter register using configurable threshold and multiplier values. The threshold is a 6-bit value and the multiplier is a 3-bit encoded value, allowing eight multiplier values in the range of 1 to 128. Any counter may be configured to increment only when an event occurs more than [threshold × multiplier] times. The contents of these registers are reflected to UPMLCb0–UPMLCb3, which can be read from user mode with **mfpmr**.

1.13 Legacy Support of PowerPC Architecture

This section provides an overview of the architectural differences and compatibilities of the e500 core compared with the AIM PowerPC architecture. The two levels of the e500 programming environment are as follows:

- User level—This defines the base user-level instruction set, user-level registers, data types, memory conventions, and the memory and programming models seen by application programmers.
- Supervisor level—This defines supervisor-level resources typically required by an operating system, the memory management model, supervisor level registers, and the exception model.

In general, the e500 core supports the user-level architecture from the existing AIM architecture. The following subsections are intended to highlight the main differences. For specific implementation details refer to the relevant chapter.

1.13.1 Instruction Set Compatibility

The following sections generally describe the user and supervisor instruction sets.

1.13.1.1 User Instruction Set

The e500 core executes legacy user-mode binaries and object files except for the following:

- The e500 supports vector and scalar single-precision floating-point operations as APUs. The e500v2 supports scalar double-precision floating-point instructions. These instructions have different encoding than the AIM definition of the PowerPC architecture. Additionally, the e500 core uses GPRs for floating-point operations, rather than the FPRs defined by the UISA. Most porting of floating-point operations can be handled by recompiling.
- String instructions are not implemented on the e500; therefore, trap emulation must be provided to ensure backward compatibility.

1.13.1.2 Supervisor Instruction Set

The supervisor mode instruction set defined by the AIM version of the PowerPC architecture is compatible with the e500 with the following exceptions:

- The MMU architecture is different, so some TLB manipulation instructions have different semantics.
- Instructions that support the BATs and segment registers are not implemented.

1.13.2 Memory Subsystem

Both Book E and the AIM version of the PowerPC architecture provide separate instruction and data memory resources. The e500 provides additional cache control features, including cache locking.

1.13.3 Exception Handling

Exception handling is generally the same as that defined in the AIM version of the PowerPC architecture for the e500, with the following differences:

- Book E defines a new critical interrupt, providing an extra level of interrupt nesting. The critical interrupt includes external critical and watchdog timer time-out inputs.
- The machine check exception differs from the Book E and from the AIM definition. It defines the Return from Machine Check Interrupt instruction, **rfmci**, and two machine check save/restore registers, MCSRR0 and MCSRR1.
- Book E processors can use IVPR and IVORs to set exception vectors individually, but they can be set to the address offsets defined in the OEA to provide compatibility.
- Unlike the AIM version of the PowerPC architecture, Book E does not define a reset vector; execution begins at a fixed virtual address, 0xFFFF_FFFC.
- Some Book E and e500-specific SPRs are different from those defined in the AIM version of the PowerPC architecture, particularly those related to the MMU functions. Much of this information has been moved to a new exception syndrome register (ESR).
- Timer services are generally compatible, although Book E defines a new decrementer auto reload feature, the fixed-interval timer critical interrupt, and the watchdog timer interrupt, which are implemented in the e500 core.

An overview of the interrupt and exception handling capabilities of the e500 core can be found in Section 1.8, “Interrupts and Exception Handling.”

1.13.4 Memory Management

The e500 core implements a straightforward virtual address space that complies with the Book E MMU definition, which eliminates segment registers and block address translation resources. Book E defines resources for fixed 4-Kbyte pages and multiple, variable page sizes that can be configured in a single implementation. TLB management is provided with new instructions and SPRs.

1.13.5 Reset

Book E–compliant cores do not share a common reset vector with the AIM version of the PowerPC architecture. Instead, at reset fetching begins at address 0xFFFF_FFFC. In addition to the Book E reset definition, the EIS and the e500 define specific aspects of the MMU page translation and protection mechanisms. Unlike the AIM version of the PowerPC core, as soon as instruction fetching begins, the e500 core is in virtual mode with a hardware-initialized TLB entry.

EIS–defined aspects of the MMU are described in the EREF. Specific details of how the e500 is initialized are provided in Section 12.6, "TLB States after Reset."

1.13.6 Little-Endian Mode

Unlike the AIM version of the PowerPC architecture, where little-endian mode is controlled on a system basis, Book E allows control of byte ordering on a memory page basis. In addition, the little-endian mode used in Book E is true little endian.

Chapter 2
Register Model

This chapter describes implementation-specific details of the register model as it is implemented on the e500 core processors. It identifies all registers that are implemented on the e500 cores, 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 Book E architecture and by the Freescale Book E implementation standards (EIS). A full description of these registers is provided in the *EREF: A Reference for Freescale Book E and the e500 Core* (EREF).

It is important to note that a device that integrates the e500 core may not implement all of the fields and registers that are defined here, and may interpret some fields more specifically than can be defined here. For specific details, refer to the "Register Summary" chapter in the reference manual for the device that incorporates the e500 core. The register summary chapter fully describes all registers and register fields as they are implemented on the device.

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 most operations. The e500 implements 64-bit GPRs. Book E–defined instructions access only the lower word; SPE vector instructions and embedded vector single-precision and double-precision floating-point APUs (e500v2 only) use all 64 bits. See Section 2.3.1, "General-Purpose Registers (GPRs)."
- Special-purpose registers (SPRs)—Accessed by using the Book E–defined Move to Special-Purpose Register (**mtspr**) and Move from Special-Purpose Register (**mfspr**) instructions. Section 2.2.1, "Special-Purpose Registers (SPRs)," lists SPRs.
- System-level registers that are not SPRs. These are as follows:
  — Machine state register (MSR). MSR is accessed with the Move to Machine State Register (**mtmsr**) and Move from Machine State Register (**mfmsr**) instructions. See Section 2.5.1, "Machine State Register (MSR)."
  — Condition register (CR) bits are grouped into eight 4-bit fields, CR0–CR7, which are set as follows:
    – Specified CR fields can be set by a move to the CR from a GPR (**mtcrf**).
    – A specified CR field can be set by a move to the CR from another CR field (**mcrf**), or from the XER (**mcrxr**).

    – CR0 can be set as the implicit result of an integer instruction.
    – A specified CR field can be set as the result of an integer or floating-point compare instruction (including SPE and SPFP compare instructions).

    See Section 2.4.1, "Condition Register (CR)."
  — The EIS-defined accumulator, used by the SPE APU. See Section 2.14.2, "Accumulator (ACC)."
- Performance monitor registers (PMRs). Similar to SPRs, PMRs are accessed by using the EIS-defined Move to Performance Monitor Register (**mtpmr**) and Move from Performance Monitor Register (**mfspr**) instructions. See Section 2.15, "Performance Monitor Registers (PMRs)."

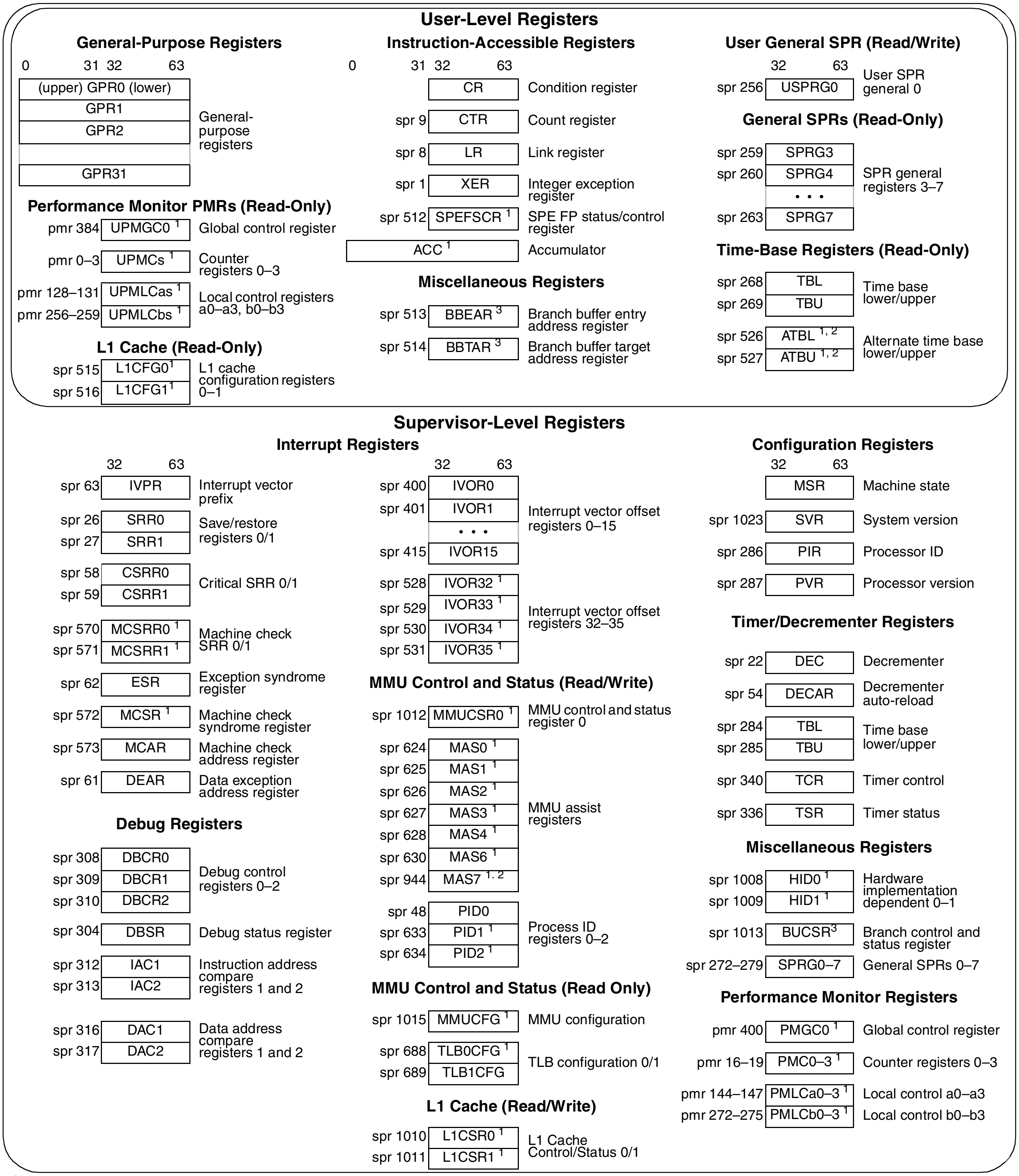
2.2 e500 Register Model

The following sections describe the e500 core register model as defined in Book E and the additional implementation-specific registers unique to the e500 core. Figure 2-1 shows the e500 register set and identifies which are defined by Book E, which are defined by the EIS, and which are e500-specific.

Book E processors implement the following types of software-accessible registers:

- Book E–defined registers that are accessed as part of instruction execution. These include the following:
  — Registers used for integer operations:
    – General-purpose registers (GPRs)—Book E defines a set of 32 GPRs used to hold source and destination operands for load, store, arithmetic, and computational instructions, and to read and write to other registers.
    – Integer exception register (XER)—Bits in this register are set based on the operation of an instruction considered as a whole, not on intermediate results. (For example, the Subtract from Carrying instruction (**subfc**), the result of which is specified as the sum of three values, sets bits in the XER based on the entire operation, not on an intermediate sum.)

  These registers are described in Section 2.3, "Registers for Integer Operations."
  — Condition register (CR)—Used to record conditions such as overflows and carries that occur as a result of executing arithmetic instructions (including those implemented by the SPE and SPFP APUs). The CR is described in Section 2.4, "Registers for Branch Operations."
  — Machine state register (MSR)—Used by the operating system to configure parameters such as user/supervisor mode, address space, and enabling of asynchronous interrupts. MSR is described in Section 2.5.1, "Machine State Register (MSR)."



[1] These registers are defined by the EIS
[2] e500v2 only
[3] These registers are e500-specific

Figure 2-1. e500 Register Model

- Book E–defined special-purpose registers (SPRs) that are accessed explicitly using **mtspr** and **mfspr** instructions. These registers are listed in Table 2-1 in Section 2.2.1, "Special-Purpose Registers (SPRs)."
- Freescale EIS–defined SPRs and e500-defined SPRs that are accessed explicitly using the **mtspr** and **mfspr** instructions. These registers are listed in Table 2-2 in Section 2.2.1, "Special-Purpose Registers (SPRs)."
- Freescale EIS–defined performance monitor registers (PMRs). These registers are similar to SPRs, but are accessed with EIS–defined move to and move from PMR instructions (**mtpmr** and **mfpmr**).

Book E– and e500-defined SPRs are grouped by function as follows:

- Section 2.4, "Registers for Branch Operations." This section includes descriptions of the count register (CTR) and the link register (LR).
- Section 2.5, "Processor Control Registers"
- Section 2.6, "Timer Registers"
- Section 2.7, "Interrupt Registers"
- Section 2.8, "Software-Use SPRs (SPRG0–SPRG7 and USPRG0)"
- Section 2.9, "Branch Target Buffer (BTB) Registers"
- Section 2.10, "Hardware Implementation-Dependent Registers"
- Section 2.11, "L1 Cache Configuration Registers"
- Section 2.12, "MMU Registers"
- Section 2.13, "Debug Registers"
- Section 2.14, "SPE and SPFP APU Registers"

Book E defines 32- and 64-bit registers. All 32-bit registers are supported as defined in Book E. However, except for the 64-bit FPRs, which are not implemented on the e500, only bits 32–63 of Book E's 64-bit registers (such as LR, CTR, the GPRs, SRR0, and CSRR0) are required to be implemented in hardware in a 32-bit Book E implementation. The e500 implements 64-bit GPRs, the upper 32 bits of which are used only with the e500-specific signal processing engine (SPE) APU, embedded vector single-precision floating-point APU, and the e500v2 embedded scalar double-precision floating-point APU instructions.

Likewise, all Book E integer instructions defined to return a 64-bit result return only bits 32–63 of the result on a 32-bit Book E implementation. SPE APU vector instructions return 64-bit values, as do DPFP APU instructions on the e500v2; SPFP APU instructions return single-precision 32-bit values.

NOTE

The SPE APU and embedded floating-point APU functionality is implemented in all PowerQUICC III devices. However, these instructions will not be supported in devices subsequent to PowerQUICC III. Freescale Semiconductor strongly recommends that use of these instructions be confined to libraries and device drivers. Customer software that uses SPE or embedded floating-point APU instructions at the assembly level or that uses SPE intrinsics will require rewriting for upward compatibility with next-generation PowerQUICC devices.

Freescale Semiconductor offers a libmoto_e500 library that uses SPE instructions. Freescale will also provide libraries to support next-generation PowerQUICC devices.

This chapter describes how the e500 implements registers defined by Book E. As with the instruction set and other aspects of the architecture, Book E defines some features very specifically, for example, resources that ensure compatibility with implementations of the PowerPC ISA. However, because a principal goal of the Book E architecture is to offer flexibility among embedded processors and families of embedded processors, some resources are either defined as optional or are defined in a very general way, leaving specific details up to the implementation.

2.2.1 Special-Purpose Registers (SPRs)

SPRs are on-chip registers that are architecturally part of the processor core. They control the use of the debug facilities, timers, interrupts, memory management unit, and other architected processor resources and are accessed with the **mtspr** and **mfspr** instructions. Unlisted encodings are reserved for future use.

Table 2-1 summarizes SPRs defined in Book E. The SPR numbers are used in the instruction mnemonics. Bit 5 in an SPR number indicates whether an SPR is accessible from user or supervisor software. An **mtspr** or **mfspr** instruction that specifies an unsupported SPR number is considered an invalid instruction. The e500 treats such invalid instructions as follows:

- If the invalid SPR falls within the range specified as user mode (SPR[5] = 0), an illegal exception is taken.
- If supervisor software attempts to access an invalid supervisor-level SPR (SPR[5] = 1), results are undefined.
- If user software attempts to access an invalid supervisor-level SPR, a privilege exception is taken.

Table 2-1. Book E Special-Purpose Registers (by SPR Abbreviation)

SPR Abbreviation	Name	Defined SPR Number		Access	Supervisor Only	Section/ Page
		Decimal	Binary			
ATBL	Alternate time base register lower	526	10000 01110	Read-only	No	2.6.6/2-16
ATBU	Alternate time base register upper	527	10000 01111	Read-only	No	2.6.6/2-16
CSRR0	Critical save/restore register 0	58	00001 11010	Read/Write	Yes	2.7.1.1/2-18
CSRR1	Critical save/restore register 1	59	00001 11011	Read/Write	Yes	2.7.1.1/2-18
CTR	Count register	9	00000 01001	Read/Write	No	2.4.3/2-10
DAC1	Data address compare 1	316	01001 11100	Read/Write	Yes	2.13.4/2-48
DAC2	Data address compare 2	317	01001 11101	Read/Write	Yes	2.13.4/2-48
DBCR0	Debug control register 0 [1]	308	01001 10100	Read/Write	Yes	2.13.1/2-46
DBCR1	Debug control register 1 [1]	309	01001 10101	Read/Write	Yes	2.13.1/2-46
DBCR2	Debug control register 2 [1]	310	01001 10110	Read/Write	Yes	2.13.1/2-46
DBSR	Debug status register	304	01001 10000	Read/Clear[2]	Yes	2.13.2/2-47
DEAR	Data exception address register	61	00001 11101	Read/Write	Yes	2.6.5/2-16
DEC	Decrementer	22	00000 10110	Read/Write	Yes	2.6.4/2-16
DECAR	Decrementer auto-reload	54	00001 10110	Write-only	Yes	2.6.4/2-16
ESR	Exception syndrome register	62	00001 11110	Read/Write	Yes	2.7.1.6/2-20
IAC1	Instruction address compare 1	312	01001 11000	Read/Write	Yes	2.13.3/2-48
IAC2	Instruction address compare 2	313	01001 11001	Read/Write	Yes	2.13.3/2-48
IVOR0	Critical input	400	01100 10000	Read/Write	Yes	2.7.1.5/2-19
IVOR1	Machine check interrupt offset	401	01100 10001	Read/Write	Yes	2.7.1.5/2-19
IVOR2	Data storage interrupt offset	402	01100 10010	Read/Write	Yes	2.7.1.5/2-19
IVOR3	Instruction storage interrupt offset	403	01100 10011	Read/Write	Yes	2.7.1.5/2-19
IVOR4	External input interrupt offset	404	01100 10100	Read/Write	Yes	2.7.1.5/2-19
IVOR5	Alignment interrupt offset	405	01100 10101	Read/Write	Yes	2.7.1.5/2-19
IVOR6	Program interrupt offset	406	01100 10110	Read/Write	Yes	2.7.1.5/2-19
IVOR8	System call interrupt offset	408	01100 11000	Read/Write	Yes	2.7.1.5/2-19
IVOR10	Decrementer interrupt offset	410	01100 11010	Read/Write	Yes	2.7.1.5/2-19
IVOR11	Fixed-interval timer interrupt offset	411	01100 11011	Read/Write	Yes	2.7.1.5/2-19
IVOR12	Watchdog timer interrupt offset	412	01100 11100	Read/Write	Yes	2.7.1.5/2-19
IVOR13	Data TLB error interrupt offset	413	01100 11101	Read/Write	Yes	2.7.1.5/2-19
IVOR14	Instruction TLB error interrupt offset	414	01100 11110	Read/Write	Yes	2.7.1.5/2-19
IVOR15	Debug interrupt offset	415	01100 11111	Read/Write	Yes	2.7.1.5/2-19
IVPR	Interrupt vector	63	00001 11111	Read/Write	Yes	2.7.1.4/2-19
LR	Link register	8	00000 01000	Read/Write	No	2.4.2/2-10

Table 2-1. Book E Special-Purpose Registers (by SPR Abbreviation) (continued)

SPR Abbreviation	Name	Defined SPR Number		Access	Supervisor Only	Section/ Page
		Decimal	Binary			
PID	Process ID register [3]	48	00001 10000	Read/Write	Yes	2.12.1/2-36
PIR	Processor ID register	286	01000 11110	Read-only	Yes	2.5.2/2-12
PVR	Processor version register	287	01000 11111	Read-only	Yes	2.5.3/2-13
SPRG0	SPR general 0	272	01000 10000	Read/Write	Yes	2.8/2-24
SPRG1	SPR general 1	273	01000 10001	Read/Write	Yes	2.8/2-24
SPRG2	SPR general 2	274	01000 10010	Read/Write	Yes	2.8/2-24
SPRG3	SPR general 3	259	01000 00011	Read-only	No[4]	2.8/2-24
		275	01000 10011	Read/Write	Yes	
SPRG4	SPR general 4	260	01000 00100	Read-only	No	2.8/2-24
		276	01000 10100	Read/Write	Yes	
SPRG5	SPR general 5	261	01000 00101	Read-only	No	2.8/2-24
		277	01000 10101	Read/Write	Yes	
SPRG6	SPR general 6	262	01000 00110	Read-only	No	2.8/2-24
		278	01000 10110	Read/Write	Yes	
SPRG7	SPR general 7	263	01000 00111	Read-only	No	2.8/2-24
		279	01000 10111	Read/Write	Yes	
SRR0	Save/restore register 0	26	00000 11010	Read/Write	Yes	2.7.1.1/2-18
SRR1	Save/restore register 1	27	00000 11011	Read/Write	Yes	2.7.1.1/2-18
TBL	Time base lower	268	01000 01100	Read-only	No	2.6.3/2-16
		284	01000 11100	Write-only	Yes	2.6.3/2-16
TBU	Time base upper	269	01000 01101	Read-only	No	2.6.3/2-16
		285	01000 11101	Write-only	Yes	2.6.3/2-16
TCR	Timer control register	340	01010 10100	Read/Write	Yes	2.6.1/2-15
TSR	Timer status register	336	01010 10000	Read/Clear[5]	Yes	2.6.2/2-16
USPRG0	User SPR general 0[6]	256	01000 00000	Read/Write	No	2.8/2-24
XER	Integer exception register	1	00000 00001	Read/Write	No	2.3.2/2-9

1 Writing to these registers requires synchronization, as described in Section 2.16, "Synchronization Requirements for SPRs."

2 The DBSR is read using **mfspr**. It cannot be directly written to. Instead, DBSR bits corresponding to 1 bits in the GPR can be cleared using **mtspr**.

3 Implementations may support more than one PID. The e500 implements the Book E–defined PID as PID0.

4 User-mode read access to SPRG3 is implementation-dependent.

5 The TSR is read using **mfspr**. It cannot be directly written to. Instead, TSR bits corresponding to 1 bits in the GPR can be cleared using **mtspr**.

6 USPRG0 is a separate physical register from SPRG0.

Table 2-2 describes the implementation-specific SPRs of the core complex. Compilers should recognize the mnemonic name given in Table 2-2 when parsing instructions.

Table 2-2. Implementation-Specific SPRs (by SPR Abbreviation)

SPR Abbreviation	Name	SPR Number	Access	Supervisor Only	Section/Page
BBEAR	Branch buffer entry address register [1]	513	Read/Write	No	2.9.1/2-25
BBTAR	Branch buffer target address register [1]	514	Read/Write	No	2.9.2/2-25
BUCSR	Branch unit control and status register [1]	1013	Read/Write	Yes	2.9.3/2-26
HID0	Hardware implementation dependent register 0 [1]	1008	Read/Write	Yes	2.10.1/2-27
HID1	Hardware implementation dependent register 1 [1]	1009	Read/Write	Yes	2.10.1/2-27
IVOR32	SPE/embedded floating-point APU unavailable interrupt offset	528	Read/Write	Yes	2.7.1.5/2-19
IVOR33	Embedded floating-point data exception interrupt offset	529	Read/Write	Yes	2.7.1.5/2-19
IVOR34	Embedded floating-point round exception interrupt offset	530	Read/Write	Yes	2.7.1.5/2-19
IVOR35	Performance monitor	531	Read/Write	Yes	2.7.1.5/2-19
L1CFG0	L1 cache configuration register 0	515	Read-only	No	2.11.3/2-34
L1CFG1	L1 cache configuration register 1	516	Read-only	No	2.11.4/2-35
L1CSR0	L1 cache control and status register 0 [1]	1010	Read/Write	Yes	2.11.1/2-31
L1CSR1	L1 cache control and status register 1 [1]	1011	Read/Write	Yes	2.11.2/2-33
MAS0	MMU assist register 0 [1]	624	Read/Write	Yes	2.12.5.1/2-40
MAS1	MMU assist register 1 [1]	625	Read/Write	Yes	2.12.5.2/2-41
MAS2	MMU assist register 2 [1]	626	Read/Write	Yes	2.12.5.3/2-42
MAS3	MMU assist register 3 [1]	627	Read/Write	Yes	2.12.5.4/2-43
MAS4	MMU assist register 4 [1]	628	Read/Write	Yes	2.12.5.5/2-43
MAS6	MMU assist register 6 [1]	630	Read/Write	Yes	2.12.5.6/2-44
MAS7	MMU assist register 7 [1]	944	Read/Write	Yes	2.12.5.7/2-45
MCAR	Machine check address register	573	Read-only	Yes	2.7.2.3/2-22
MCSR	Machine check syndrome register	572	Read/Write	Yes	2.7.2.4/2-23
MCSRR0	Machine-check save/restore register 0	570	Read/Write	Yes	2.7.2.1/2-22
MCSRR1	Machine-check save/restore register 1	571	Read/Write	Yes	2.7.2.2/2-22
MMUCFG	MMU configuration register	1015	Read-only	Yes	2.12.3/2-37
MMUCSR0	MMU control and status register 0 [1]	1012	Read/Write	Yes	2.12.2/2-36
PID0	Process ID register 0. Book E defines only this PID register and refers to as PID, not PID0. [1]	48	Read/Write	Yes	2.12.1/2-36
PID1	Process ID register 1 [1]	633	Read/Write	Yes	2.12.1/2-36
PID2	Process ID register 2 [1]	634	Read/Write	Yes	2.12.1/2-36
SPEFSCR	Signal processing and embedded floating-point status and control register [1]	512	Read/Write	No	2.14.1/2-49

Table 2-2. Implementation-Specific SPRs (by SPR Abbreviation) (continued)

SPR Abbreviation	Name	SPR Number	Access	Supervisor Only	Section/Page
SVR	System version register	1023	Read-only	Yes	2.5.4/2-13
TLB0CFG	TLB configuration register 0	688	Read-only	Yes	2.12.4/2-37
TLB1CFG	TLB configuration register 1	689	Read-only	Yes	2.12.4.2/2-39

[1] Writing to these registers requires synchronization, as described in Section 2.16, "Synchronization Requirements for SPRs."

2.3 Registers for Integer Operations

The following sections describe registers defined for integer computational instructions.

2.3.1 General-Purpose Registers (GPRs)

Book E implementations provide 32 GPRs (GPR0–GPR31) for integer operations. The instruction formats provide 5-bit fields for specifying the GPRs to be used in the execution of the instruction. Each GPR is a 64-bit register and can be used to contain address and integer data, although all instructions except SPE APU instructions, double-precision embedded floating-point instructions (e500v2 only), and single-precision embedded vector floating-point instructions use and return 32-bit values in GPR bits 32–63.

2.3.2 Integer Exception Register (XER)

Bits in the integer exception register (XER) are set based on the operation of an instruction considered as a whole, not on intermediate results. (For example, the Subtract from Carrying instruction (**subfc**), the result of which is specified as the sum of three values, sets bits in the XER based on the entire operation, not on an intermediate sum.)

The e500 implements the XER as it is defined by Book E.

2.4 Registers for Branch Operations

This section describes registers used by Book E branch and CR operations.

2.4.1 Condition Register (CR)

The e500 implements the CR as it is defined by Book E for integer instructions. Note that the embedded floating-point instructions do not use the CR.

2.4.2 Link Register (LR)

The e500 implements the LR as it is defined by Book E.

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

2.4.3 Count Register (CTR)

The e500 implements the CTR as it is defined by Book E. The 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 CTR value is 0 before being decremented, it is –1 afterward. The entire CTR can be used to hold the branch target address for a Branch Conditional to CTR (**bcctr***x*) instruction.

2.5 Processor Control Registers

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

2.5.1 Machine State Register (MSR)

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

Access: Supervisor-only

	32 36	37	38	39 44	45	46	47	48	49	50	51	52	53	54	55 57	58	59	60	61	62 63
R W	—	UCLE	SPE	—	WE	CE	—	EE	PR	FP	ME	—	UBLE	DE	—	IS	DS	—	PMM	—
Reset	All zeros																			

Figure 2-2. Machine State Register (MSR)

MSR contents are automatically saved, altered, and restored by the interrupt-handling mechanism. If a non-critical interrupt is taken, MSR contents are automatically copied into SRR1. If a critical interrupt is taken, MSR contents are automatically copied into CSRR1. When an **rfi** or **rfci** is executed, MSR contents are restored from SRR1 or CSRR1. The e500 implements the machine check interrupt differently than it is defined in Book E. When a machine check interrupt is taken, MCSRR0 and MCSRR1 hold the return address and MSR information. The EIS defines the Return from Machine Check Interrupt instruction, **rfmci**, which restores MSR contents from MCSRR1 when it is executed.

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.

Table 2-3 describes e500-specific MSR fields. Note that other registers in this chapter describe only fields that are either e500-specific or that differ from the Book E definition.

Table 2-3. MSR Field Descriptions

Bits	Name	Description
32–36	—	Reserved, should be cleared.[1]
37	UCLE	User-mode cache lock enable. (e500-specific). Used to restrict user-mode cache-line locking by the operating system. 0 Any cache lock instruction executed in user-mode takes a cache-locking DSI exception and sets either ESR[DLK] or ESR[ILK]. This allows the operating system to manage and track the locking/unlocking of cache blocks by user-mode tasks. 1 Cache-locking instructions can be executed in user-mode and they do not take a DSI for cache-locking. (They may still take a DSI for access violations, though.)
38	SPE	SPE enable. (e500-specific). 0 If software attempts to execute an instruction that accesses the upper word of a GPR, the SPE APU unavailable exception is taken. 1 Software can execute the following instructions: On the e500v1, these instructions include the SPE instructions and both vector and scalar single-precision floating-point instructions. On the e500v2, these instructions include the SPE instructions, embedded double-precision, and single-precision vector floating-point instructions. (That is, all instructions that access the upper half of the 64-bit GPRs.)
39–44	—	Reserved, should be cleared. [1]
45	WE	Wait state enable. On the e500, this allows the core complex to signal a request for power management, according to the states of HID0[DOZE], HID0[NAP], and HID0[SLEEP]. 0 The processor is not in wait state and continues processing. On the e500, no power management request is signaled to external logic. 1 The processor enters wait state by ceasing to execute instructions and entering low-power mode. Details of how wait state is entered and exited and how the processor behaves in the wait state are implementation dependent. On the e500, MSR[WE] gates the DOZE, NAP, and SLEEP outputs from the core complex; as a result, these outputs negate to the external power management logic on entry to the interrupt and then return to their previous state on return from the interrupt. WE is cleared on entry to any interrupt and restored to its previous state upon return.
46	CE	Critical enable. Book E defines this bit as an enable for the critical input, watchdog timer, and machine check interrupts. On the e500, this bit does not affect machine check interrupts. 0 Critical input and watchdog timer interrupts are disabled. 1 Critical input and watchdog timer interrupts are enabled.
47	—	Reserved, should be cleared.
48	EE	External enable 0 External input, decrementer, fixed-interval timer, and performance monitor interrupts are disabled. 1 External input, decrementer, fixed-interval timer, and performance monitor interrupts are enabled.
49	PR	User mode (problem state) 0 The processor is in supervisor mode, can execute any instruction, and can access any resource (for example, GPRs, SPRs, and the MSR). 1 The processor is in user mode, cannot execute any privileged instruction, and cannot access any privileged resource. PR also affects memory access control.

Table 2-3. MSR Field Descriptions (continued)

Bits	Name	Description
50	FP	Floating-point available. Book E defines the operation of FP as follows: 0 The processor cannot execute floating-point instructions, including floating-point loads, stores, and moves. 1 The processor can execute floating-point instructions. On the e500, this bit is reserved and permanently cleared, indicating that it does not implement a Book E floating-point unit (FPU). Setting it has no effect.
51	ME	Machine check enable. 0 Machine check interrupts are disabled. On e500 cores, a machine check condition causes a checkstop. 1 Machine check interrupts are enabled.
52	FE0	Floating-point exception mode 0. On the e500, this bit is reserved and permanently cleared, indicating that the e500 does not implement a Book E FPU. Setting it has no effect.
53	UBLE	Allocated for implementation-dependent use. On the e500, it is the user BTB lock enable bit. 0 Execution of the BTB lock instructions for user mode is disabled; a privileged instruction exception is taken instead. 1 Execution of the BTB lock instructions for user mode is enabled.
54	DE	Debug interrupt enable 0 Debug interrupts are disabled. 1 Debug interrupts are enabled if DBCR0[IDM] = 1. For the e500, see the description of the DBSR[UDE] in Section 2.13.2, “Debug Status Register (DBSR).”
55	FE1	Floating-point exception mode 1. On the e500, this bit is reserved and permanently cleared, indicating that the e500 does not implement a Book E FPU. Setting it has no effect.
56–57	—	Reserved, should be cleared. [1]
58	IS	Instruction address space 0 The processor directs all instruction fetches to address space 0 (TS = 0 in the relevant TLB entry). 1 The processor directs all instruction fetches to address space 1 (TS = 1 in the relevant TLB entry).
59	DS	Data address space 0 The processor directs data memory accesses to address space 0 (TS = 0 in the relevant TLB entry). 1 The processor directs data memory accesses to address space 1 (TS = 1 in the relevant TLB entry).
60	—	Reserved, should be cleared. [1]
61	PMM	Performance monitor mark bit. System software can set PMM when a marked process is running to enable statistics to be gathered only during the execution of the marked process. MSR[PR] and MSR[PMM] together define a state that the processor (supervisor or user) and the process (marked or unmarked) may be in at any tim e. If this state matches an individual state specified in the PMLCa*n*, the state for which monitoring is enabled, counting is enabled.
62–63	—	Reserved, should be cleared. [1]

[1] An MSR bit that is reserved may be altered by return from interrupt instructions.

2.5.2 Processor ID Register (PIR)

The e500 implements the processor ID register (PIR) as defined by the Book E architecture. The PIR contains a value that can be used to distinguish the processor from other processors in the system.

2.5.3 Processor Version Register (PVR)

The e500 implements the processor version register (PVR) as defined by the Book E architecture. The read-only PVR, shown in Figure 2-3, contains a value identifying the version and revision level of the processor. The PVR distinguishes between processors that differ in attributes that may affect software.

SPR 287 Access: Supervisor read-only

	32–47	48–63
R	Version	Revision
W		
Reset(e500v1)	1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
Reset (e500v2)	1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1	0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0

Figure 2-3. Processor Version Register (PVR)

Table 2-4 describes the PVR fields.

Table 2-4. PVR Field Descriptions

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

2.5.4 System Version Register (SVR)

The system version register (SVR), shown in Figure 2-4, contains a read-only SoC-dependent value; consult the documentation for the implementation.

SPR 1023 Access: Supervisor read-only

	32–63
R	System version
W	
Reset	SoC-dependent value (determined by *svr*[0:31]. See Section 13.2, “Signal Summary.”)

Figure 2-4. System Version Register (SVR)

2.6 Timer Registers

The time base (TB), decrementer (DEC), fixed-interval timer (FIT), and watchdog timer provide timing functions for the system. The e500 provides the ability to select any of the TB bits to trigger watchdog and fixed-interval timer events, as shown in Figure 2-5.

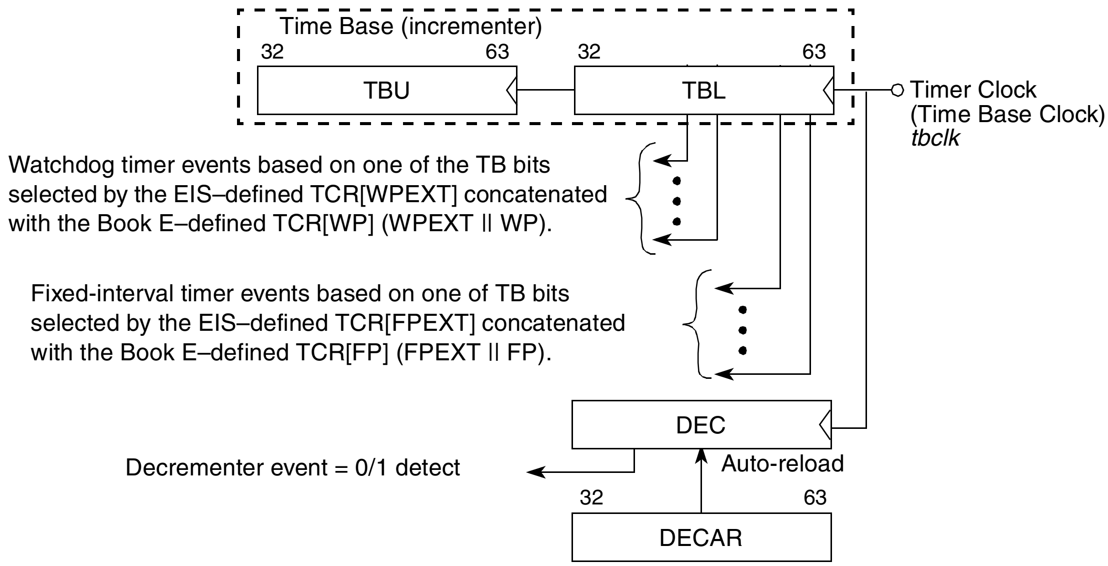


Figure 2-5. Relationship of Timer Facilities to the Time Base

e500 registers involved in timing are described as follows:

- The TB is a long-period counter driven at an implementation-dependent frequency.
- The decrementer, updated at the same rate as the TB, provides a way to signal an exception after a specified period unless one of the following occurs:
  - — DEC is altered by software in the interim.
  - — The TB update frequency changes.

  DEC is typically used as a general-purpose software timer.
- The time base for the TB and DEC is selected by the time base enable (TBEN) and select time base clock (SEL_TBCLK) bits in HID0, as follows:
  - — If HID0[TBEN] = 1 and HID0[SEL_TBCLK] = 0, the time base is updated every 8 bus clocks.
  - — If HID0[TBEN] = 1 and HID0[SEL_TBCLK] = 1, the time base is updated on the rising edge of *tbclk* (or an implementation-specific clock input).
- Software can select one from of four 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. Bits that may be selected are implementation-dependent.

- The watchdog timer, also a selected TB bit, provides a way to signal a critical exception when the selected bit transitions from 0 to 1. It is typically used for system error recovery. If software does not respond in time to the initial interrupt by clearing the associated status bits in the TSR before the next expiration of the watchdog timer interval, a watchdog timer-generated processor reset may result, if so enabled.

All timer facilities must be initialized during start-up.

2.6.1 Timer Control Register (TCR)

The e500 implements the TCR, shown in Figure 2-6, as defined by the Book E architecture except as follows:

- TCR[WPEXT] and TCR[FPEXT], not specified in Book E, are concatenated with TCR[WP] and TCR[FP] to select a bit that triggers the watchpoint timer and fixed-interval timer events.
- The value programmed into WRC is reflected on the e500 *wrs* signals.

SPR 340 Access: Supervisor-only

	32 33	34 35	36	37	38 39	40	41	42	43 46	47 50	51 63
R W	WP	WRC	WIE	DIE	FP	FIE	ARE	—	WPEXT	FPEXT	—
Reset	All zeros										

Figure 2-6. Timer Control Register (TCR)

Table 2-5 describes the e500 TCR fields that differ from the Book E definition.

Table 2-5. TCR Implementation-Specific Field Descriptions

Bits	Name	Description
32–33	WP	Watchdog timer period. When concatenated with WPEXT, specifies one of 64-bit locations of the time base used to signal a watchdog timer exception on a transition from 0 to 1. WPEXT,WP = 0000_00 selects TBU[32] (the msb of the TB) WPEXT,WP = 1111_11 selects TBL[63] (the lsb of the TB)
34–35	WRC	Watchdog timer reset control. When a watchdog reset event occurs, the value programmed into WRC is reflected on *wrs* and into TSR[WRS], but the WRC bits are reset to 00. At this point, software can reprogram WRC. Although WRC can be set by software, it cannot be cleared by software (except by a software-induced reset). Once written to a non-zero value, WRC may no longer be altered by software. 00 No watchdog timer reset will occur. 01 Force processor checkstop on second timeout of watchdog timer 10 Assert processor reset output (*p_resetout_b*) on second timeout of watchdog timer 11 Reserved
38–39	FP	Fixed interval timer period. When concatenated with FPEXT, FP 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. FPEXT || FP = 0000_00 selects TBU[32] (the msb of the TB) FPEXT || FP = 1111_11 selects TBL[63] (the lsb of the TB)

Table 2-5. TCR Implementation-Specific Field Descriptions (continued)

Bits	Name	Description
43–46	WPEXT	Watchdog timer period extension (see the description for WP)
47–50	FPEXT	Fixed-interval timer period extension (see the description for FP)

2.6.2 Timer Status Register (TSR)

The e500 implements the TSR as it is defined by the Book E architecture. The 32-bit TSR contains status on timer events and the most recent watchdog timer-initiated processor reset. All TSR bits function as write-1-to-clear.

2.6.3 Time Base (TBU and TBL)

The e500 implements the time base registers as they are defined by the Book E architecture. The time base (TB) is composed of two 32-bit registers, the time base upper (TBU) concatenated on the right with the time base lower (TBL). TB provides timing functions for the system. TB is a volatile resource and must be initialized during start-up.

2.6.4 Decrementer Register (DEC)

The e500 implements the DEC as it is defined by the Book E architecture. DEC is a 32-bit decrementing counter that is updated at the same rate as the TB. It provides a way to signal a decrementer interrupt after a specified period unless one of the following occurs:

- DEC is altered by software in the interim.
- The TB update frequency changes.

DEC is typically used as a general-purpose software timer. The decrementer auto-reload register is used to automatically reload a programmed value into DEC, as described in Section 2.6.5, "Decrementer Auto-Reload Register (DECAR)."

2.6.5 Decrementer Auto-Reload Register (DECAR)

The e500 implements the DECAR as it is defined by the Book E architecture. If the auto-reload function is enabled (TCR[ARE] = 1), the auto-reload value in DECAR is written to DEC when DEC decrements from 0x0000_0001 to 0x0000_0000. Note that writing DEC with zeros by using an **mtspr[DEC]** does not automatically generate a decrementer exception.

2.6.6 Alternate Time Base Registers (ATBL and ATBU)

The alternate time base counter (ATB), shown in Figure 2-7, is formed by concatenating the upper and lower alternate time base registers (ATBU and ATBL). ATBL (SPR 526) provides read-only access to the 64-bit alternate time base counter, which is incremented at an implementation-defined frequency. On the e500v2, this frequency is the core frequency. The ATB register is accessible in both user and supervisor mode.

Like the TB implementation, the ATBL register is an aliased name for ATB.

SPR 526 Access: User read-only

	32–63
R	ATBLU
W	
Reset	All zeros

Figure 2-7. Alternate Time Base Register Lower (ATBL)

Table 2-6 describes the ATB fields.

Table 2-6. ATBL Field Descriptions

Bits	Name	Description
32–63	ATBCL	Alternate time base counter lower Lower 32 bits of the alternate time base counter

2.6.6.1 Alternate Time Base Upper (ATBU)

The ATBU register, shown in Figure 2-8, provides read-only access to the upper 32 bits of the alternate time base counter. It is accessible in both user and supervisor mode.

SPR 527 Access: User read-only

	32–63
R	ATBCU
W	
Reset	All zeros

Figure 2-8. Alternate Time Base Register Upper (ATBU)

Table 2-7 describes the ATBU fields.

Table 2-7. ATBU Field Descriptions

Bits	Name	Description
32–63	ATBCU	Alternate time base counter upper Upper 32 bits of the alternate time base counter

2.7 Interrupt Registers

Section 2.7.1, “Interrupt Registers Defined by Book E,” and Section 2.7.2, “e500-Specific Interrupt Registers,” describe registers used for interrupt handling.

2.7.1 Interrupt Registers Defined by Book E

This section describes the following register bits and their fields:

- Section 2.7.1.1, "Save/Restore Register 0/1 (SRR0 and SRR1)"
- Section 2.7.1.2, "Critical Save/Restore Register 0/1 (CSRR0 and CSRR1)"
- Section 2.7.1.3, "Data Exception Address Register (DEAR)"
- Section 2.7.1.4, "Interrupt Vector Prefix Register (IVPR)"
- Section 2.7.1.5, "Interrupt Vector Offset Registers (IVORs)"
- Section 2.7.1.6, "Exception Syndrome Register (ESR)"

2.7.1.1 Save/Restore Register 0/1 (SRR0 and SRR1)

The e500 implements SRR0 and SRR1 as they are defined by the Book E architecture. On a noncritical interrupt, SRR0 holds the address of the instruction where the interrupted process should resume. The instruction is interrupt-specific, although for instruction-caused exceptions, it is typically the address of the instruction that caused the interrupt. When **rfi** executes, instruction execution continues at the address in SRR0.

SRR1 is provided to save and restore machine state on noncritical interrupts. When a noncritical interrupt is taken, MSR contents are placed in SRR1. When **rfi** executes, SRR1 contents are placed into MSR. SRR1 bits that correspond to reserved MSR bits are also reserved. These registers are not affected by **rfci** or **rfmci**. Reserved MSR bits may be altered by **rfi**, **rfci**, or **rfmci**.

2.7.1.2 Critical Save/Restore Register 0/1 (CSRR0 and CSRR1)

The e500 implements CSRR0 and CSRR1 as they are defined by the Book E architecture. On a critical interrupt, CSRR0 holds the address of the instruction where the interrupted process should resume. The instruction is interrupt-specific, although for instruction-caused exceptions, it is typically the address of the instruction that caused the interrupt. When **rfci** executes, instruction execution continues at the address in CSRR0.

CSRR1 is provided to save and restore machine state on critical interrupts. When a critical interrupt is taken, MSR contents are placed in CSRR1. When **rfci** executes, SRR1 contents are placed into MSR. CSRR1 bits that correspond to reserved MSR bits are also reserved. These registers are not affected by **rfi** or **rfmci**. Reserved MSR bits may be altered by **rfi**, **rfci**, or **rfmci**.

2.7.1.3 Data Exception Address Register (DEAR)

The e500 implements DEAR as it is defined by the Book E architecture. DEAR is loaded with the effective address of a data access (caused by a load, store, or cache management instruction) that results in an alignment, data TLB miss, or DSI exception.

2.7.1.4 Interrupt Vector Prefix Register (IVPR)

The e500 implements IVPR as it is defined by the Book E architecture. It is used with IVORs to determine the vector address. IVPR[32–47] provides the high-order 16 bits of the address of the exception processing routines. The 16-bit vector offsets are concatenated to the right of IVPR[32–47] to form the address of the exception processing routine.

2.7.1.5 Interrupt Vector Offset Registers (IVORs)

The e500 implements the IVORs as defined by the Book E architecture, but use only IVOR*n*[48–59], as shown in Figure 2-9, to hold the quad-word index from the base address provided by the IVPR for each interrupt type.

SPR (See Table 2-8.) Access: Supervisor-only

	32–46	47–59	60–63
R/W	—	Interrupt vector offset	—
Reset	All zeros		

Figure 2-9. Interrupt Vector Offset Registers (IVORs)

Table 2-8 shows the IVORs implemented on the e500. IVOR0–IVOR15 are defined by the architecture. (Note that the e500 does not implement IVOR7 and IVOR9.) In addition, IVOR32–IVOR35 (SPR 528–531) are used by the e500 APUs.

Table 2-8. IVOR Assignments

IVOR Number	SPR	Interrupt Type
IVOR0	400	Critical input
IVOR1	401	Machine check
IVOR2	402	Data storage
IVOR3	403	Instruction storage
IVOR4	404	External input
IVOR5	405	Alignment
IVOR6	406	Program
IVOR7	407	Floating-point unavailable (Not supported on the e500)
IVOR8	408	System call
IVOR9	409	Auxiliary processor unavailable (Not supported on the e500)
IVOR10	410	Decrementer
IVOR11	411	Fixed-interval timer interrupt
IVOR12	412	Watchdog timer interrupt
IVOR13	413	Data TLB error
IVOR14	414	Instruction TLB error
IVOR15	415	Debug

Table 2-8. IVOR Assignments (continued)

IVOR Number	SPR	Interrupt Type
IVOR16–IVOR31	—	Reserved for future architectural use
IVOR32	528	(e500-specific) SPE APU unavailable
IVOR33	529	(e500-specific) Embedded floating-point data exception
IVOR34	530	(e500-specific) Embedded floating-point round exception
IVOR35	531	(e500-specific) Performance monitor
IVOR36–IVOR63	—	Allocated for implementation-dependent use

2.7.1.6 Exception Syndrome Register (ESR)

Figure 2-10 shows the ESR as it is defined on the e500.

The ESR provides a way to differentiate among exceptions that can generate an interrupt type. When an interrupt is generated, bits corresponding to the specific exception that generated the interrupt are set and all other ESR bits are cleared. Other interrupt types do not affect ESR contents. The ESR does not need to be cleared by software. Table 2-9 shows ESR bit definitions. The e500 defines ESR[SPE] as the SPE/embedded floating-point exception bit. It is set whenever the processor takes an exception related to the execution of SPE or SPFP instructions. Note that the e500 does not use the ESR for machine check interrupts, but instead uses the machine check syndrome register, MCSR, described in Section 2.7.2.4, "Machine Check Syndrome Register (MCSR)." The ESR is defined in Book E but differs in the following respects:

- The e500 defines ESR[DLK0] (bit 42) as ESR[DLK].
- The e500 defines ESR[DLK1] (bit 43) as ESR[ILK].
- The e500 defines ESR[SPE] (bit 56).
- The e500 does not implement FP, AP, PIE, or PUO.

SPR 62 Access: Supervisor-only

	32–35	36	37	38	39	40	41	42	43	44–45	46	47–55	56	57–63
R/W	—	PIL	PPR	PTR	—	ST	—	DLK	ILK	—	BO	—	SPE	—
Reset	All zeros													

Figure 2-10. Exception Syndrome Register (ESR)

Table 2-9 describes the ESR fields, showing the associated interrupts. Note that an implementation may implement additional ESR bits to identify implementation-specific or architected interrupt types.

NOTE

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

Table 2-9. ESR Field Descriptions

Bits	Name	Syndrome	Interrupt Types
32–35	—	Reserved, should be cleared. (Defined by Book E as allocated.)	—
36	PIL	Illegal instruction exception	Program
37	PPR	Privileged instruction exception	Program
38	PTR	Trap exception	Program
39	—	Not supported on the e500. Defined by Book E as FP (floating-point operations). On the e500, this bit is reserved and permanently cleared, indicating that the e500 does not implement a Book E FPU. Setting it has no effect.	—
40	ST	Store operation	Alignment, DSI, DTLB error
41	—	Reserved, should be cleared.	—
42	DLK	Data cache locking (defined by Book E as DLK0). Settings are implementation dependent. 0 Default 1 On the e500, DLK is 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. (Book E defines this bit as DLK1.) Set when a DSI occurs because **icbtl** or **icblc** is executed in user mode (MSR[PR] = 1 and MSR[UCLE] = 0)	DSI
44	—	Not supported on the e500. Defined by Book E as AP (auxiliary processor operation).	—
45	—	Not supported on the e500. Unimplemented operation exception. On the e500, unimplemented instructions are handled as illegal instructions.	Program
46	BO	Byte-ordering exception	DSI, ISI
47	—	Not supported on the e500. Defined by Book E as PIE, Imprecise exception.	—
48–55	—	Reserved, should be cleared.	—
56	SPE	SPE/embedded floating-point exception bit (e500-specific) 0 Default 1 Any exception caused by an SPE or and SPFP instruction occurred.	
57–63	—	Reserved, should be cleared (defined by Book E as allocated).	—

2.7.2 e500-Specific Interrupt Registers

This section describes machine check save/store and syndrome registers.

2.7.2.1 Machine Check Save/Restore Register 0 (MCSRR0)

When a machine check interrupt is taken, MCSRR0, shown in Figure 2-11, is set to the address of the instruction where the interrupted process should resume. The instruction is interrupt-specific, although typically MCSRR0 holds the address of the instruction that caused the interrupt. After **rfmci** executes, instruction execution continues at this address.

SPR 570 Access: Supervisor-only

	32 — 63
R W	Next instruction address
Reset	All zeros

Figure 2-11. Machine Check Save/Restore Register 0 (MCSRR0)

2.7.2.2 Machine Check Save/Restore Register 1 (MCSRR1)

MCSRR1 is used to save and restore machine state on machine check interrupts. When a machine check interrupt is taken, MSR contents are placed into MCSRR1, shown in Figure 2-12. When **rfmci** executes, MCSRR1 contents are restored to MSR. MCSRR1 bits that correspond to reserved MSR bits are also reserved; reserved MSR bits may be altered.

SPR 571 Access: Supervisor-only

	32 — 63
R W	MSR state information
Reset	All zeros

Figure 2-12. Machine Check Save/Restore Register 1 (MCSRR1)

2.7.2.3 Machine Check Address Register (MCAR)

When the core complex takes a machine check interrupt, it updates MCAR (Figure 2-13) to indicate the address of the data associated with the machine check. Note that if a machine check interrupt is caused by a signal, the contents of MCAR are not meaningful.

SPR 573 Access: Supervisor-only

	32 — 63
R W	Machine check address
Reset	All zeros

Figure 2-13. Machine Check Address Register (MCAR)

2.7.2.4 Machine Check Syndrome Register (MCSR)

When the core complex takes a machine check interrupt, it updates MCSR to differentiate between machine check conditions. The MCSR indicates whether a machine check condition is recoverable. When a condition bit is set, the core complex asserts MCP_OUT for system information. ABIST status is logged in MCSR[48–54]. These bits do not initiate machine check (or any other exception). An ABIST bit being set indicates an error being detected in the corresponding module. The MCSR is shown in Figure 2-14.

SPR 572 Access: Supervisor-only

	32	33	34	35	36–39
R/W	MCP	ICPERR	DCP_PERR	DCPERR	—
Reset	All zeros				

	40–47
R/W	—
Reset	All zeros

	48–55
R/W	—
Reset	All zeros

	56	57	58	59	60	61	62	63
R/W	BUS_IAERR	BUS_RAERR	BUS_WAERR	BUS_IBERR	BUS_RBERR	BUS_WBERR	BUS_IPERR	BUS_RPERR
Reset	All zeros							

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

Table 2-10 describes the MCSR fields.

Table 2-10. MCSR Field Descriptions

Bit	Name	Description
32	MCP	Machine check input to core $\overline{mcp}$
33	ICPERR	Instruction cache parity error
34	DCP_PERR	Data cache push parity error
35	DCPERR	Data cache parity error
36–55	—	Reserved, should be cleared.
56	BUS_IAERR	Bus instruction address error
57	BUS_RAERR	Bus read address error
58	BUS_WAERR	Bus write address error

Table 2-10. MCSR Field Descriptions (continued)

Bit	Name	Description
59	BUS_IBERR	Bus instruction data bus error
60	BUS_RBERR	Bus read data bus error
61	BUS_WBERR	Bus write bus error
62	BUS_IPERR	Bus instruction parity error
63	BUS_RPERR	Bus read parity error

2.8 Software-Use SPRs (SPRG0–SPRG7 and USPRG0)

The e500 implements the software-use SPRs (SPRG0–SPRG7 and USPRG0) as defined by the Book E architecture. They have no defined functionality and are accessed as follows:

- SPRG0–SPRG2—These registers can be accessed only in supervisor mode.
- SPRG3—This register can be written only in supervisor mode. It is readable in supervisor mode, but whether it can be read in user mode is implementation-dependent. It is readable in user mode on the e500.
- SPRG4–SPRG7—These registers can be written only in supervisor mode. They are readable in supervisor or user mode.
- USPRG0—This register can be accessed in supervisor or user mode.

2.9 Branch Target Buffer (BTB) Registers

SPRs are defined in the core complex for enabling the locking and unlocking of entries in the BTB. These are called the branch buffer entry address register (BBEAR), the branch buffer target address register (BBTAR), and branch unit control and status register (BUCSR). The user branch locking enable bit, MSR[UBLE], is defined to allow user-mode programs to lock or unlock BTB entries.

See Section 3.9.1, "Branch Target Buffer (BTB) Locking Instructions," for more information about BTB locking. Section 2.5.1, "Machine State Register (MSR)," describes MSR bits that support the BTB.

2.9.1 Branch Buffer Entry Address Register (BBEAR)

BBEAR is shown in Figure 2-15. Writing to BBEAR requires synchronization, as described in Section 2.16, “Synchronization Requirements for SPRs.”

SPR 513 Access: Supervisor/user

	32–61	62–63
R/W	Branch buffer entry address	IAB[0–1]
Reset	All zeros	

Figure 2-15. Branch Buffer Entry Address Register (BBEAR)

Table 2-12 describes the BBEAR fields.

Table 2-11. BBEAR Field Descriptions

Bits	Name	Description
32–61	Branch buffer entry address	Branch buffer effective entry address bits 0–29
62–63	IAB[0–1]	Instruction after branch (with BBTAR[62]). 3-bit pointer that points to the instruction in the cache block after the branch. If the branch is the last instruction in the cache block, IAB = 000, to indicate the next sequential instruction, which resides in the zeroth position of the next cache block.

2.9.2 Branch Buffer Target Address Register (BBTAR)

Figure 2-16 shows the BBTAR. Writing to BBTAR requires synchronization, as described in Section 2.16, “Synchronization Requirements for SPRs.”

SPR 513 Access: Supervisor/user

	32–61	62	63
R/W	Branch buffer target address	IAB2	BDIRPR
Reset	All zeros		

Figure 2-16. Branch Buffer Target Address Register (BBTAR)

Table 2-12 describes BBTAR fields.

Table 2-12. BBTAR Field Descriptions

Bits	Name	Description
32–61	Branch buffer target address	Branch buffer target address bits 0–29
62	IAB2	Instruction after branch bit 2 (with BBEAR[62–63]). IAB is a 3-bit pointer that points to the instruction in the cache block after the branch. See the **bblels** instruction description.
63	BDIRPR	Branch direction prediction. The user can pick the direction of the predicted branch. 0 The locked address is always predicted as not taken. 1 The locked address is always predicted as taken.

2.9.3 Branch Unit Control and Status Register (BUCSR)

The BUCSR, shown in Figure 2-17, is used for general control and status of the branch target buffer (BTB). Writing to BUCSR requires synchronization, as described in Section 2.16, "Synchronization Requirements for SPRs."

SPR 1013 Access: Supervisor-only

	32–53	54	55	56	57	58–62	63
R/W	—	BBFI	BBLO	BBUL	BBLFC	—	BPEN
Reset	All zeros						

Figure 2-17. Branch Unit Control and Status Register (BUCSR)

BUCSR provides control of BTB locking, including the following:

- Enable or disable BTB locking
- Invalidate all BTB entries at once (flash invalidate)
- Unlock all BTB entries at once (flash lock clear)

Table 2-13 describes the BUCSR fields.

Table 2-13. BUCSR Field Descriptions

Bits	Name	Description
32–53	—	Reserved, should be cleared.
54	BBFI	Branch buffer flash invalidate. Clearing and then setting BBFI flash clears the valid bit of all entries in the branch buffer; clearing occurs independently from the value of the enable bit (BPEN). BBFI is always read as 0.
55	BBLO	Branch buffer lock overflow status 0 Indicates a lock overflow condition was not encountered in the branch buffer 1 Indicates a lock overflow condition was encountered in the branch buffer This sticky bit is set by hardware and is cleared by writing 0 to this bit location.
56	BBUL	Branch buffer unable to lock 0 Indicates a lock overflow condition in the branch buffer 1 Indicates a lock set instruction failed in the branch buffer, for example, if the BTB is disabled This sticky bit is set by hardware and is cleared by writing 0 to this bit location.
57	BBLFC	Branch buffer lock bits flash clear. Clearing and then setting BBLFC flash clears the lock bit of all entries in the branch buffer; clearing occurs independently from the value of the enable bit (BPEN). BBLFC is always read as 0.
58–62	—	Reserved, should be cleared.
63	BPEN	Branch prediction enable 0 Branch prediction disabled 1 Branch prediction enabled (enables BTB to predict branches)

2.10 Hardware Implementation-Dependent Registers

This section describes the e500-specific HID0 and HID1 registers.

2.10.1 Hardware Implementation-Dependent Register 0 (HID0)

This section describes the HID0 register, shown in Figure 2-18, as it is defined by the e500 core.

NOTE

Note that some HID fields may not be implemented in a device that incorporates the e500 core and that some fields may be defined more specifically by the incorporating device. For specific details it is important to refer to the "Register Summary" chapter in the device's reference manual.

HID0 is used for configuration and control. Writing to HID0 requires synchronization, as described in Section 2.16, "Synchronization Requirements for SPRs."

SPR 1008 — Access: Supervisor-only

	32	33–39	40	41	42	43–47
R/W	EMCP	—	DOZE	NAP	SLEEP	—
Reset	All zeros					

	48	49	50	51–55	56	57	58–62	63
R/W	—	TBEN	SEL_TBCLK	—	EN_MAS7_UPDATE	DCFA	—	NOPTI
Reset	All zeros							

Figure 2-18. Hardware Implementation-Dependent Register 0 (HID0)

Table 2-14 describes the HID0 fields.

Table 2-14. HID0 Field Descriptions

Bits	Name	Description
32	EMCP	Enable machine check signal, $\overline{\text{mcp}}$. Used to mask out further machine check exceptions caused by asserting the internal $\overline{\text{mcp}}$ signal. 0 $\overline{\text{mcp}}$ is disabled. 1 $\overline{\text{mcp}}$ is enabled. If MSE[ME] = 0, asserting $\overline{\text{mcp}}$ causes checkstop. If MSR[ME] = 1, asserting $\overline{\text{mcp}}$ causes a machine check exception.
33–39	—	Reserved, should be cleared.
40	DOZE	Doze power management mode. If MSR[WE] is set, this bit controls the *doze* output signal. Interpretation of this bit is handled by integrated system logic. 0 *doze* is not asserted. 1 *doze* is asserted.

Table 2-14. HID0 Field Descriptions (continued)

Bits	Name	Description
41	NAP	Nap power management mode. If MSR[WE] is set, this bit controls the *nap* output signal. Interpretation of this bit is handled by integrated system logic. 0 *nap* is not asserted. 1 *nap* is asserted.
42	SLEEP	Configure for sleep power management mode. If MSR[WE] is set, this bit controls the *sleep* output signal. Interpretation of this bit is handled by integrated system logic. 0 *sleep* is not asserted 1 *sleep* is asserted
43–48	—	Reserved, should be cleared.
49	TBEN	Time base and decrementer enable 0 Time base disabled 1 Time base enabled
50	SEL_TBCLK	Select time base clock 0 Time base is based on the processor clock 1 Time base is based on TBCLK input
51–55	—	Reserved, should be cleared.
56	EN_MAS7_UPDATE	Enable MAS7 update (e500v2 only). Enables updating MAS7 by **tlbre** and **tlbsx**. 0 MAS7 is not updated by a **tlbre** or **tlbsx**. 1 MAS7 is updated by a **tlbre** or **tlbsx**.
57	DCFA	Data cache flush assist (e500v2 only). Force data cache to ignore invalid sets on miss replacement selection. 0 The data cache flush assist facility is disabled 1 The miss replacement algorithm ignores invalid entries and follows the replacement sequence defined by the PLRU bits. This reduces the series of uniquely addressed load or **dcbz** instructions to eight per set. The bit should be set just before beginning a cache flush routine and should be cleared when the series of instructions is complete.
58–62	—	Reserved, should be cleared.
63	NOPTI	No-op the data and instruction cache touch instructions. 0 **dcbt**, **dcbtst**, and **icbt** are enabled, as defined by the EIS. Note that on the e500, if CT = 0, **icbt** is always a no-op, regardless of the value of NOPTI. If CT = 1, **icbt** does a touch load to the L2 cache. 1 **dcbt**, **dcbtst**, and **icbt** are treated as no-ops; **dcblc** and **dcbtls** are not treated as no-ops.

2.10.2 Hardware Implementation-Dependent Register 1 (HID1)

This section describes the HID1 register, shown in Figure 2-19, as it is defined by the e500 core.

SPR 1009 | Access: Supervisor-only

	32–37	38–45	46	47
R	PLL_CFG	—	RFXE	—
W				
Reset	All zeros			

	48	49	50	51	52	53	54–55	56	57–59	60–63
R	R1DPE	R2DPE	ASTME	ABE	—	MPXTT	—	ATS	—	MID
W										
Reset	All zeros									

Figure 2-19. Hardware Implementation-Dependent Register 1 (HID1)

NOTE

Note that some HID fields may not be implemented in a device that incorporates the e500 core and that some fields may be defined more specifically by the incorporating device. For specific details it is important to refer to the “Register Summary” chapter in the device’s reference manual.

HID1 is used for bus configuration and control. Writing to HID1 requires synchronization, as described in Section 2.16, “Synchronization Requirements for SPRs.”

Table 2-15 describes the HID1 fields.

Table 2-15. HID1 Field Descriptions

Bits	Name	Description
32–37	PLL_CFG	Reflected directly from the PLL_CFG input pins (read-only)
38–46	—	Reserved, should be cleared.

Table 2-15. HID1 Field Descriptions (continued)

Bits	Name	Description
46	RFXE	Read fault exception enable. With MSR[ME], controls whether assertion of *core_fault_in* causes a machine check interrupt. The assertion of *core_fault_in* can result from an L2 multibit ECC error. It can also occur for a system error if logic on the integrated device signals a fault for a transaction initiated by the core (for example, a master abort of a PCI transaction). See **Section 13.8, "Proper Reporting of Bus Faults,"** for more information. 0 Normal operation. Assertion of *core_fault_in* does not cause a machine check. In normal operation RFXE should be left clear and an interrupt should be reported by the integrated device (possibly through *int* or *cint*) for *core_fault_in* conditions. If RFXE = 0, it is important that the integrated device be configured to generate an interrupt when *core_fault_in* is asserted. 1 A machine check can occur due to assertion of *core_fault_in*. If MSR[ME] = 1 and a fault is signaled, a machine check interrupt occurs. If MSR[ME] = 0 and a fault is signaled, a checkstop occurs. Caveat for the e500v1. CCB transactions that result in *core_fault_in* being asserted may contain bad data. On the e500v1, such transactions may complete and the core could continue executing with bad data. Note that even if the peripheral blocks are set up to signal an interrupt to the core for all possible causes of *core_fault_in*, there is some delay between the completion of the CCB transaction (with potentially bad data) and the processing of the peripheral block interrupt.Therefore, for the e500v1, if software requires that code execution stop immediately when a bus fault occurs, RFXE must be set to 1 so that at a minimum, a machine check exception is taken immediately and processing does not continue with potentially bad data. However, setting RFXE when a peripheral block is configured to also signal an interrupt for a *core_fault_in* case results in both a machine check interrupt (if MSR[ME] = 0) and potentially an external interrupt occuring when a bus fault is detected by that peripheral.þIn this case, the machine check interrupt handler can re-enable external interrupts and wait for the interrupt from the peripheral block, and handle the condition, before returning from the machine check exception, therefore protecting the system from using potentially bad data. Note that on the e500v2, the core never completes a CCB transaction for which *core_fault_in is* asserted, so the above precautions regarding execution with bad data do not apply. RFXE should always be 0 for normal operation for the e500v2; it should be set only if it is necessary that the assertion of *core_fault_in* generate a machine check or a checkstop because peripherals are not properly configured to report bus faults. This would typically occur only during software or firmware development. Note that the L2 cache detects any assertion of *core_fault_in* and ensures that the L2 cache is not corrupted when data is dropped for this type of transaction. Machine check generation for bus parity errors is not affected by this bit.
47	—	Reserved, should be cleared.
48	R1DPE	R1 data bus parity enable. The R1 and R2 data buses are described in Chapter 13, "Core Complex Bus (CCB)." 0 R1 data bus parity checking disabled 1 R1 data bus parity checking enabled
49	R2DPE	R2 data bus parity enable. The R1 and R2 data buses are described in Chapter 13, "Core Complex Bus (CCB)." 0 R2 data bus parity checking disabled 1 R2 data bus parity checking enabled
50	ASTME	Address bus streaming mode enable 0 Address bus streaming mode disabled 1 Address bus streaming mode enabled

Table 2-15. HID1 Field Descriptions (continued)

Bits	Name	Description
51	ABE	Address broadcast enable. The e500 broadcasts cache management instructions (**dcbst**, **dcblc** (CT = 1), **icblc** (CT = 1), **dcbf**, **mbar**, **msync**, **tlbivax**, **tlbsync**, **icbi**) based on ABE. On some implementations, ABE must be set to allow management of external L2 caches. 0 Address broadcasting disabled 1 Address broadcasting enabled
52	—	Reserved, should be cleared.
53	MPXTT	MPX re-map transfer type 0 TTx codes are not remapped. 1 Certain TTx codes are remapped for MPX bus compatibility. See the integrated device documentation.
54–55	—	Reserved. should be cleared.
56	ATS	Atomic status (read-only). Indicates state of atomic status bit in bus unit.
57–59	—	Reserved, should be cleared.
60–63	MID	Reflected directly from the MID input pins (read-only)

2.11 L1 Cache Configuration Registers

The Freescale Book E standards define registers that provide control and configuration and status information for the L1 cache implementation.

2.11.1 L1 Cache Control and Status Register 0 (L1CSR0)

The L1CSR0 register, shown in Figure 2-20, is defined by the EIS. It is used for general control and status of the L1 data cache. Writing to L1CSR0 requires synchronization, as described in Section 2.16, “Synchronization Requirements for SPRs.”

SPR 1010 Access: Supervisor-only

	32–46	47	48	49–51	52	53	54	55	56–61	62	63
					Line Locking APU Bits						
R W	—	CPE	CPI	—	CSLC	CUL	CLO	CLFR	—	CFI	CE
Reset	All zeros										

Figure 2-20. L1 Cache Control and Status Register 0 (L1CSR0)

Table 2-16 describes the L1CSR0 fields.

Table 2-16. L1CSR0 Field Descriptions

Bits	Name	Description
32–46	—	Reserved, should be cleared.
47	CPE	(Data) Cache parity enable. See Section 5.7.2, "Machine Check Interrupt." 0 Parity checking of the cache disabled 1 Parity checking of the cache enabled Note that if the programmer attempts to set L1CSR0[CPI] (using **mtspr**) without setting L1CSR0[CPE], L1CSR0[CPI] will not be set (enforced by hardware).
48	CPI	(Data) Parity error injection enable. See Section 5.7.2.2, "Cache Parity Error Injection." 0 Parity error injection disabled 1 Parity error injection enabled. Cache parity must also be enabled (CPE = 1) when this bit is set. Note that if the programmer attempts to set L1CSR0[CPI] (using **mtspr**) without setting L1CSR0[CPE], L1CSR0[CPI] will not be set (enforced by hardware).
49–51	—	Reserved, should be cleared.
52	CSLC	(Data) Cache snoop lock clear. Sticky bit set by hardware if a **dcbi** snoop (either internally or externally generated) invalidated a locked cache block. Note that the lock bit for that line is cleared whenever the line is invalidated. This bit can be cleared only by software. 0 The cache has not encountered a **dcbi** snoop that invalidated a locked line. 1 The cache has encountered a **dcbi** snoop that invalidated a locked line.
53	CUL	(Data) Cache unable to lock. Sticky bit set by hardware and cleared by writing 0 to this bit location. 0 Indicates a lock set instruction was effective in the cache 1 Indicates a lock set instruction was not effective in the cache
54	CLO	(Data) Cache lock overflow. Sticky bit set by hardware and cleared by writing 0 to this bit location. 0 Indicates a lock overflow condition was not encountered in the cache 1 Indicates a lock overflow condition was encountered in the cache
55	CLFR	(Data) Cache lock bits flash reset. Writing a 1 during a flash clear operation causes an undefined operation. Writing a 0 during a flash clear operation is ignored. Clearing occurs regardless of the enable (CE) value. 0 Default 1 Hardware initiates a cache lock bits flash clear operation. CLFR resets to 0 when the operation completes.
56–61	—	Reserved, should be cleared.
62	CFI	(Data) Cache flash invalidate. (Invalidation occurs regardless of the enable (CE) value.) 0 No cache invalidate. Writing a 0 to CFI during an invalidation operation is ignored. 1 Cache invalidation operation. A cache invalidation operation is initiated by hardware. Once complete, CFI is cleared. Writing a 1 during an invalidation causes an undefined operation.
63	CE	(Data) Cache enable 0 The cache is neither accessed or updated. 1 Enables cache operation

2.11.2 L1 Cache Control and Status Register 1 (L1CSR1)

The L1CSR1 register, defined as part of the EIS, is shown in Figure 2-21. It is used for general control and status of the L1 instruction cache. Writing to L1CSR1 requires synchronization, as described in Section 2.16, "Synchronization Requirements for SPRs."

SPR 1011 Access: Supervisor-only

	32–46	47	48	49–51	52	53	54	55	56–61	62	63
					Line Locking APU Bits						
R/W	—	ICPE	ICPI	—	ICSLC	ICUL	ICLO	ICLFR	—	ICFI	ICE
Reset	All zeros										

Figure 2-21. L1 Cache Control and Status Register 1 (L1CSR1)

Table 2-17 describes the L1CSR1 fields.

Table 2-17. L1CSR1 Field Descriptions

Bits	Name	Description
32–46	—	Reserved, should be cleared.
47	ICPE	Instruction cache parity enable. See Section 5.7.2, "Machine Check Interrupt." 0 Parity checking of the instruction cache disabled 1 Parity checking of the instruction cache enabled Note that if the programmer attempts to set L1CSR1[ICPI] (using **mtspr**) without setting L1CSR1[ICPE], L1CSR1[ICPI] will not be set (enforced by hardware).
48	ICPI	Instruction parity error injection enable. See Section 5.7.2.2, "Cache Parity Error Injection." 0 Parity error injection into instruction cache disabled 1 Parity error injection into instruction cache enabled.Instruction cache parity must also be enabled (ICPE = 1) when this bit is set. Note that if the programmer attempts to set L1CSR1[ICPI] (using **mtspr**) without setting L1CSR1[ICPE], L1CSR1[ICPI] will not be set (enforced by hardware).
49–51	—	Reserved, should be cleared.
52	ICSLC	Instruction cache snoop lock clear. Sticky bit set by hardware if an **icbi** snoop (either internally or externally generated) invalidated a locked line in the instruction cache. Note that the lock bit for that line is cleared whenever the line is invalidated. This bit can only be cleared by software. 0 The instruction cache has not encountered an **icbi** snoop that invalidated a locked line. 1 The instruction cache has encountered an **icbi** snoop that invalidated a locked line.
53	ICUL	Instruction cache unable to lock. Sticky bit set by hardware and cleared by writing 0 to this bit location. 0 Indicates a lock set instruction was effective in the instruction cache 1 Indicates a lock set instruction was not effective in the instruction cache
54	ICLO	Instruction cache lock overflow. Sticky bit set by hardware and cleared by writing 0 to this bit location. 0 Indicates a lock overflow condition was not encountered in the instruction cache 1 Indicates a lock overflow condition was encountered in the instruction cache
55	ICLFR	Instruction cache lock bits flash reset. Writing 0 and then 1 flash clears the lock bit of all entries in the instruction cache; clearing occurs independently from the value of the enable bit (ICE). ICLFR is always read as 0.
56–61	—	Reserved, should be cleared.

Table 2-17. L1CSR1 Field Descriptions (continued)

Bits	Name	Description
62	ICFI	Instruction cache flash invalidate. Write to 0 and then write to 1 to flash clear the valid bit of all entries in the instruction cache; operates independently from the value of the enable bit (ICE). ICFI is always read as 0.
63	ICE	Instruction cache enable 0 The instruction cache is neither accessed or updated. 1 Enables instruction cache operation.

2.11.3 L1 Cache Configuration Register 0 (L1CFG0)

The L1CFG0 register, shown in Figure 2-22, is defined by the EIS to provide configuration information for the L1 data cache supplied with this version of the e500 core complex.

SPR 515 — Access: Supervisor read-only

	32–33	34–38	39–40	41–42	43	44	45–49	50–52	53–55	56–63
R	CARCH	—	CBSIZE	CREPL	CLA	CPA	—	CNWAY	—	CSIZE
W										
Reset	0 0	0 0 0 0 0	0 0	0 1	1	1	0 0 0 0 0	1 1 1	0 0 0	0 0 1 0 0 0 0 0

Figure 2-22. L1 Cache Configuration Register 0 (L1CFG0)

Table 2-18 describes the L1CFG0 fields.

Table 2-18. L1CFG0 Field Descriptions

Bits	Name	Description
32–33	CARCH	Cache architecture 00 Harvard 01 Unified
34–38	—	Reserved, should be cleared.
39–40	CBSIZE	Cache block size 0 32 bytes 1 64 bytes
41–42	CREPL	Cache replacement policy 0 True LRU 1 Pseudo LRU
43	CLA	Cache locking APU available 0 Unavailable 1 Available
44	CPA	Cache parity available 0 Unavailable 1 Available
45–49	—	Reserved, should be cleared.
50–52	CNWAY	Cache number of ways. 111 indicates eight ways

Table 2-18. L1CFG0 Field Descriptions (continued)

Bits	Name	Description
53–55	—	Reserved, should be cleared.
56–63	CSIZE	Cache size. 0x20 indicates 32 Kbytes.

2.11.4 L1 Cache Configuration Register 1 (L1CFG1)

The L1CFG1 register, shown in Figure 2-23, provides configuration information for the particular L1 instruction cache supplied with this version of the e500 core complex.

SPR 516 — Access: Supervisor read-only

	32–38	39–40	41–42	43	44	45–52	53–63
R	—	ICBSIZE	ICREPL	ICLA	ICPA	ICNWAY	ICSIZE
W							
Reset	0 0 0 0 0 0 0	0 0	0 1	1	1	0 0 0 0 0 1 1 1	0 0 0 0 0 1 0 0 0 0 0

Figure 2-23. L1 Cache Configuration Register 1 (L1CFG1)

Table 2-19 describes the L1CFG1 fields.

Table 2-19. L1CFG1 Field Descriptions

Bits	Name	Description
32–38	—	Reserved, should be cleared.
39–40	ICBSIZ	Instruction cache block size. 00 indicates block size of 32 bytes
41–42	ICREPL	Instruction cache replacement policy. 01 indicates pseudo-LRU policy.
43	ICLA	Instruction cache locking available. 1 indicates available.
44	ICPA	Instruction cache parity available. 1 indicates available.
45–52	ICNWAY	Instruction cache number of ways. 111 indicates eight ways.
53–63	ICSIZE	Instruction cache size. 0x20 indicates 32 Kbytes.

2.12 MMU Registers

This section describes the following MMU registers and their fields:

- Process ID registers (PID0–PID2)
- MMU control and status register 0 (MMUCSR0)
- MMU configuration register (MMUCFG)
- TLB configuration registers (TLB*n*CFG)
- MMU assist registers (MAS0–MAS4, MAS6–MAS7)

2.12.1 Process ID Registers (PID0–PID2)

The Book E architecture specifies that a process ID (PID) value be associated with each effective address (instruction or data) generated by the processor. Book E defines one PID register that holds the PID value for the current process. The e500 implements two additional PID registers, PID1 and PID2, shown in Figure 2-24. The number of PIDs implemented is indicated by the value of MMUCFG[NPIDS]. PID values are used to construct virtual addresses for accessing memory. The e500 implements only PID[54–63] for the process ID. Writing to PIDs requires synchronization, as described in Section 2.16, "Synchronization Requirements for SPRs."

SPR 48 (PID0: PID in Book E);
SPR 633 (PID1: e500-specific);
SPR 634 (PID2: e500-specific)

Access: Supervisor-only

	32–53	54–63
R/W	—	Process ID
Reset	All zeros	

Figure 2-24. Process ID Registers (PID0–PID2)

2.12.2 MMU Control and Status Register 0 (MMUCSR0)

The MMUCSR0 register (Figure 2-25) is used for general control of the L1 and L2 MMUs. Writing to MMUCSR0 requires synchronization, as described in Section 2.16, "Synchronization Requirements for SPRs."

SPR 1012

Access: Supervisor-only

	32–60	61	62	63
R/W	—	L2TLB0_FI	L2TLB1_FI	—
Reset	All zeros			

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

Table 2-20 describes the MMUCSR0 fields.

Table 2-20. MMUCSR0 Field Descriptions

Bits	Name	Description
32–60	—	Reserved, should be cleared.
61	L2TLB0_FI	TLB0 flash invalidate (write 1 to invalidate)
62	L2TLB1_FI	TLB1 flash invalidate (write 1 to invalidate) 0 No flash invalidate. Writing a 0 to this bit during an invalidation operation is ignored. 1 TLB1 invalidation operation. Hardware initiates a TLB1 invalidation operation. When this operation is complete, this bit is cleared. Writing a 1 during an invalidation operation causes an undefined operation. This invalidation typically takes 1 cycle.
63	—	Reserved, should be cleared.

2.12.3 MMU Configuration Register (MMUCFG)

The MMUCFG register, shown in Figure 2-26, provides configuration information about the e500 MMU.

SPR 1015 Access: Supervisor read-only

	32–48	49–52	53–57	58–59	60–61	62–63
R	—	NPIDS	PIDSIZE	—	NTLBS	MAVN
W						
Reset	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 1 1 0	0 0 1 1 1	0 0	0 1	0 0

Figure 2-26. MMU Configuration Register (MMUCFG)

Table 2-21 describes the MMUCFG fields.

Table 2-21. MMUCFG Field Descriptions

Bits	Name	Description
32–48	—	Reserved, should be cleared.
49–52	NPIDS	Number of PID registers. A 4-bit field that indicates the number of PID registers provided by the processor. The e500 implements three PIDs.
53–57	PIDSIZE	PID register size. The 5-bit value of 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 registers. 00111 Indicates 8-bit registers. This is the value presented by the e500.
58–59	—	Reserved, should be cleared.
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].) 00 1 TLB 01 2 TLBs. This is the value presented by the e500. 10 3 TLBs 11 4 TLBs
62–63	MAVN	MMU architecture version number. Indicates the version number of the architecture of the MMU implemented by the processor. 0b00 indicates version 1.0.

2.12.4 TLB Configuration Registers (TLB*n*CFG)

The TLB*n*CFG read-only registers provide information about each specific TLB that is visible to the programming model.

2.12.4.1 TLB0 Configuration Register (TLB0CFG)

TLB0CFG, shown in Figure 2-27, provides configuration information for TLB0 of the L2 MMU supplied with this version of the e500 core complex.

SPR 688 — Access: Supervisor read-only

	32–39	40–43	44–47	48	49	50–51	52–63
R	ASSOC	MINSIZE	MAXSIZE	IPROT	AVAIL	—	NENTRY
W							
Reset (e500v1)	0 0 0 0 0 0 1 0	0 0 0 1	0 0 0 1	0	0	0 0	0 0 0 1 0 0 0 0 0 0 0 0
Reset (e500v2)	0 0 0 0 0 1 0 0	0 0 0 1	0 0 0 1	0	0	0 0	0 0 1 0 0 0 0 0 0 0 0 0

Figure 2-27. TLB Configuration Register 0 (TLB0CFG)

Table 2-22 describes the TLB0CFG fields and shows the values for the e500.

Table 2-22. TLB0CFG Field Descriptions

Bits	Name	Description
32–39	ASSOC	Associativity of TLB0 0x02 Indicates associativity is 2-way set associative (e500v1 only) 0x04 Indicates associativity is 4-way set associative (e500v2 only)
40–43	MINSIZE	Minimum page size of TLB0 0x1 Indicates smallest page size is 4 Kbytes
44–47	MAXSIZE	Maximum page size of TLB0 0x1 Indicates maximum page size is 4 Kbytes
48	IPROT	Invalidate protect capability of TLB0 0 Indicates invalidate protection capability not supported
49	AVAIL	Page size availability of TLB0 0 No variable-sized pages available (MINSIZE = MAXSIZE)
50–51	—	Reserved, should be cleared.
52–63	NENTRY	Number of entries in TLB0 0x100 TLB0 contains 256 entries (e500v1 only) 0x200 TLB0 contains 512 entries (e500v2 only)

2.12.4.2 TLB1 Configuration Register 1 (TLB1CFG)

The TLB1CFG register, shown in Figure 2-28, provides configuration information for TLB1 of the L2 MMU supplied with this version of the e500 core complex.

SPR 689 Access: Supervisor read-only

	32–39	40–43	44–47	48	49	50–51	52–63
R	ASSOC	MINSIZE	MAXSIZE	IPROT	AVAIL	—	NENTRY
W							
Reset(e500v1)	0 0 0 1 0 0 0 0	0 0 0 1	1 0 0 1	1	1	0 0	0 0 0 0 0 0 0 1 0 0 0 0
Reset (e500v2)	0 0 0 1 0 0 0 0	0 0 0 1	1 0 1 1	1	1	0 0	0 0 0 0 0 0 0 1 0 0 0 0

Figure 2-28. TLB Configuration Register 1 (TLB1CFG)

Table 2-23 describes the TLB1CFG fields.

Table 2-23. TLB1CFG Field Descriptions

Bits	Name	Description
32–39	ASSOC	Associativity of TLB1 0x10 Indicates associativity is 16
40–43	MINSIZE	Minimum page size of TLB1 0x1 Indicates smallest page size is 4 Kbytes
44–47	MAXSIZE	Maximum page size of TLB1 0x9 Indicates maximum page size is 256 Mbytes (e500v1) 0xB Indicates maximum page size is 4 Gbytes (e500v2)
48	IPROT	Invalidate protect capability of TLB1 1 Indicates that TLB1 supports invalidate protection capability
49	AVAIL	Page size availability of TLB1 1 Indicates all page sizes between MINSIZE and MAXSIZE supported
50–51	—	Reserved, should be cleared.
52–63	NENTRY	Number of entries in TLB1 0x010 TLB1 contains 16 entries

2.12.5 MMU Assist Registers (MAS0–MAS4, MAS6–MAS7)

MMU assist registers, MAS*n* are implementation-defined SPRs used by the MMU to manage pages and TLBs. They, along with MAS5 (which is not implemented in the e500), are defined by the Freescale implementation standard. Note that some fields in these registers are redefined on the e500.

2.12.5.1 MAS Register 0 (MAS0)

Figure 2-29 shows MAS0 as it is implemented on the e500. For the e500, TLB0 is two-way set associative, so bits 45–51 of the effective address are used to index into TLB0. ESEL then identifies which of the two indexed entries is to be referenced by the TLB operation (ESEL selects the way). Writing to MAS0 requires synchronization, as described in Section 2.16, "Synchronization Requirements for SPRs."

SPR 624 — Access: Supervisor-only

	32–34	35	36–43	44–47	48–61	62–63
R/W	—	TLBSEL	—	ESEL	—	NV
Reset	All zeros					

Figure 2-29. MAS Register 0 (MAS0)

The MAS0 fields are described in Table 2-24.

Table 2-24. MAS0 Field Descriptions—MMU Read/Write and Replacement Control

Bit	Name	Comments or Function when Set
32–34	—	Reserved, should be cleared.
35	TLBSEL	Selects TLB for access. 0 TLB0 1 TLB1
36–43	—	Reserved, should be cleared.
44–47	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. For the e500, ESEL serves as the way select for the corresponding TLB as follows: When TLBSEL = 00 (TLB0 selected), bits 46–47 are used (and bits 44–45 should be cleared). This field selects between way 0, 1, 2, or 3 of TLB0. EA bits 45–51 from MAS2[EPN] are used to index into the TLB to further select the entry for the operation. Note that for the e500v1, bit 47 selects either way 0 or way 1, and bit 46 should remain cleared. When TLBSEL = 01 (TLB1 selected), all four bits are used to select one of 16 entries in the array.
48–61	—	Reserved, should be cleared.
62–63	NV	Next victim. (Note that the Freescale standard allows NV to be as large as 12-bits on other implementations.) 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. If the TLB selected by MAS0[TLBSEL] does not support the NV field, then this field is undefined. The specific meaning of this field is implementation-dependent. For the e500, 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 12.3.2.2, "Replacement Algorithms for L2 MMU"). Note that for the e500v1, bit 62 should remain cleared and only bit 63 has significance. Note that this field is not defined for operations that specify TLB1 (when TLBSEL = 01).

2.12.5.2 MAS Register 1 (MAS1)

Figure 2-30 describes the format of MAS1. Note that while the Freescale Book E allows for a TID field of 12 bits, the TID field on the core complex is implemented as only 8 bits. Writing to MAS1 requires synchronization, as described in Section 2.16, “Synchronization Requirements for SPRs.”

SPR 625 Access: Supervisor-only

	32	33	34 39	40 47	48 50	51	52 55	56 63
R W	V	IPROT	—	TID	—	TS	TSIZE	—
Reset	All zeros							

Figure 2-30. MAS Register 1 (MAS1)

The MAS1 fields are described in Table 2-25.

Table 2-25. 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 due the execution of **tlbivax** (TLB1 only). Note that not all TLB arrays are necessarily protected from invalidation with IPROT. Arrays that support invalidate protection are denoted as such in the TLB configuration registers. 0 Entry is not protected from invalidation. 1 Entry is protected from invalidation.
34–39	—	Reserved, should be cleared.
40–47	TID	Translation identity. Defines the process ID for this TLB entry. TID is compared with the current process IDs of the three effective address to be translated. A TID value of 0 defines an entry as global and matches with all process IDs.
48–50	—	Reserved, should be cleared.
51	TS	Translation space. Compared with the IS or DS fields of the MSR (depending on the type of access) to determine if this TLB entry may be used for translation.
52–55	TSIZE	Translation size. Defines the page size of the TLB entry. For TLB arrays that contain fixed-size TLB entries, this field is ignored. For variable page size TLB arrays, the page size is 4^{TSIZE} Kbytes. Note that although the Freescale Book E standard supports all 16 page sizes defined in Book E, the e500 only supports the following page sizes: 0001 4 Kbyte 0010 16 Kbyte 0011 64 Kbyte 0100 256 Kbyte 0101 1 Mbyte 0110 4 Mbyte 0111 16 Mbyte 1000 64 Mbyte 1001 256 Mbyte 1010 1 Gbyte 1011 4 Gbyte
56–63	—	Reserved, should be cleared.

2.12.5.3 MAS Register 2 (MAS2)

Figure 2-31 shows the format of MAS2. Writing to MAS2 requires synchronization, as described in Section 2.16, “Synchronization Requirements for SPRs.”

SPR 626 Access: Supervisor-only

	32–51	52–56	57	58	59	60	61	62	63
R W	EPN	—	X0	X1	W	I	M	G	E
Reset	All zeros								

Figure 2-31. MAS Register 2 (MAS2)

The MAS2 fields are described in Table 2-26.

Table 2-26. MAS2 Field Descriptions—EPN and Page Attributes

Bits	Name	Description
32–51	EPN	Effective page number. Depending on page size, only the bits associated with a page boundary are valid. Bits that represent offsets within a page are ignored and should be zero.
52–56	—	Reserved, should be cleared
57	X0	Implementation-dependent page attribute
58	X1	Implementation-dependent page attribute
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.
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).
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 optionally available in previous devices that implement the PowerPC architecture. 0 The page is accessed in big-endian byte order. 1 The page is accessed in true little-endian byte order.

2.12.5.4 MAS Register 3 (MAS3)

Figure 2-32 shows the format of MAS3. Writing to MAS3 requires synchronization, as described in Section 2.16, "Synchronization Requirements for SPRs."The core complex uses the same bit definitions as the Freescale Book E standard for MAS3 for 32-bit implementations.

SPR 627 — Access: Supervisor-only

	32–51	52 53	54–57	58	59	60	61	62	63
R/W	RPN	—	U0–U3	UX	SX	UW	SW	UR	SR
Reset	All zeros								

Figure 2-32. MAS Register 3 (MAS3)

The MAS3 fields are described in Table 2-27.

Table 2-27. 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. Note that, on the e500v2, additional bits of the RPN are contained in MAS7. See Section 2.12.5.7, "MAS Register 7 (MAS7)—e500v2 Only," for more information.
52–53	—	Reserved, should be cleared.
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	PERMIS	Permission bits (UX, SX, UW, SW, UR, SR). User and supervisor read, write, and execute permission bits. See the EREF for more information on the page permission bits as they are defined by Book E.

2.12.5.5 MAS Register 4 (MAS4)

Figure 2-33 shows the format of MAS4. Writing to MAS4 requires synchronization, as described in Section 2.16, "Synchronization Requirements for SPRs."

SPR 628 — Access: Supervisor-only

	32 33	34 35	36–45	46 47	48	55	56	57	58	59	60	61	62	63
R/W	—	TLBSELD	—	TIDSELD	—	TSIZED	—	X0D	X1D	WD	ID	MD	GD	ED
Reset	All zeros													

Figure 2-33. MAS Register 4 (MAS4)

The MAS4 fields are described in Table 2-28.

Table 2-28. MAS4 Field Descriptions—Hardware Replacement Assist Configuration

Bits	Name	Description
32–33	—	Reserved, should be cleared.
34–35	TLBSELD	TLBSEL default value. 2-bit field that specifies the default value to be loaded in MAS0[TLBSEL] on a TLB miss exception.
36–45	—	Reserved, should be cleared.
46–47	TIDSELD	TID default selection value. Defined by the EIS as a 4-bit field that specifies which of the current PID registers should be used to load the MAS1[TID] field on a TLB miss exception. The e500 implementation defines bits 44–45 as reserved and bits 46–47 as follows: 00 PID0 01 PID1 10 PID2 11 TIDZ (0x00) (all zeros)
48–51	—	Reserved, should be cleared.
52–55	TSIZED	Default TSIZE value. Specifies the default value to be loaded into MAS1[TSIZE] on a TLB miss exception.
56	—	Reserved, should be cleared.
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.
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.

2.12.5.6 MAS Register 6 (MAS6)

Figure 2-34 shows the format of MAS6. Note that while the Freescale Book E allows for an SPIDx field of 12 bits, SPID0 on the core complex is only an 8-bit field. Writing to MAS6 requires synchronization, as described in Section 2.16, “Synchronization Requirements for SPRs.”

SPR 630 — Access: Supervisor-only

	32–39	40–47	48–62	63
R/W	—	SPID0	—	SAS
Reset	All zeros			

Figure 2-34. MAS Register 6 (MAS6)

The MAS6 fields are described in Table 2-29.

Table 2-29. MAS6 Field Descriptions

Bits	Name	Description
32–39	—	Reserved, should be cleared.
40–47	SPID0	Search PID0. Specifies the value of PID0 used when searching the TLB during execution of **tlbsx**. For the e500 implementation, this field contains the 8-bit search PID0 value. Specifies the value of PID0 used when searching the TLB during execution of **tlbsx**.
48–62	—	Reserved, should be cleared.
63	SAS	Address space (AS) value for searches. Specifies the value of AS used when searching the TLB during execution of **tlbsx**.

2.12.5.7 MAS Register 7 (MAS7)—e500v2 Only

The MAS7 register contains the high-order address bits of the RPN for implementations that support more than 32 bits of physical address. (The e500v1 supports 32-bit addresses, while the e500v2 supports 36-bit real addresses.) Implementations that do not support more than 32 bits of physical addressing do not implement MAS7. Figure 2-35 shows the format of the MAS7 register. Writing to MAS0 requires synchronization, as described in Section 2.16, “Synchronization Requirements for SPRs

SPR 944 Access: Supervisor-only

	32–59	60–63
R W	—	RPN
Reset	All zeros	

Figure 2-35. MAS Register 7 (MAS7)

The MAS7 fields are described in Table 2-30.

Table 2-30. MAS7 Field Descriptions—High-Order RPN

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

2.13 Debug Registers

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

2.13.1 Debug Control Registers (DBCR0–DBCR2)

The debug control registers are used to enable debug events, reset the processor, control timer operation during debug events, and set the debug mode of the processor.

2.13.1.1 Debug Control Register 0 (DBCR0)

The e500 implements DBCR0 as it is defined by Book E (see the EREF for further details) with the following exceptions:

- DBCR0[RST], bits 34–35, are implemented as shown in Table 2-31.
- IAC3 and IAC4 (DBCR0[42–43]) are not implemented.

Writing to DBCR0 requires synchronization, as described in Section 2.16, “Synchronization Requirements for SPRs.”

Table 2-31. DBCR0 Field Descriptions

Bits	Name	Description
34–35	RST	Reset. Book E defines this field such that 00 is always no action and all other settings are implementation specific. The e500 implements these bits as follows: 0*x* Default (No action) 1*x* Causes a hard reset if MSR[DE] and DBCR0[IDM] are set. Always cleared on subsequent cycle.

2.13.1.2 Debug Control Register 1 (DBCR1)

The e500 implements DBCR1 as it is defined by the Book E architecture (see the EREF for more information), except as follows:

- IAC1ER and IAC2ER values of 01 are reserved.
- IAC3US, IAC3ER, IAC4US, IAC4ER, and IAC34M (DBCR1[48–57]) are not implemented.

Writing to DBCR1 requires synchronization, as described in Section 2.16, “Synchronization Requirements for SPRs.” Table 2-32 describes the DBCR1 fields.

Table 2-32. DBCR1 Implementation-Specific Field Descriptions

Bits	Name	Description
34–35	IAC1ER	Instruction address compare 1 effective/real mode 00 IAC1 debug events are based on effective addresses. 01 Reserved on the e500. 10 IAC1 debug events are based on effective addresses and can occur only if MSR[IS] = 0. 11 IAC1 debug events are based on effective addresses and can occur only if MSR[IS] = 1.
38–39	IAC2ER	Instruction address compare 2 effective/real mode 00 IAC2 debug events are based on effective addresses. 01 Reserved on the e500. 10 IAC2 debug events are based on effective addresses and can occur only if MSR[IS] = 0. 11 IAC2 debug events are based on effective addresses and can occur only if MSR[IS] = 1.

2.13.1.3 Debug Control Register 2 (DBCR2)

The e500 implements DBCR2 as it is defined by the Book E architecture, except as follows:

- DAC1ER and DAC2ER values of 01 are reserved.
- DVC1M, DVC2M, DVC1BE, and DVC2BE (DBCR[44–63]) are not implemented.

Figure 2-36 shows the DBCR2.

SPR 310 — Access: Supervisor-only

	32–33	34–35	36–37	38–39	40–41	42–63
R/W	DAC1US	DAC1ER	DAC2US	DAC2ER	DAC12M	—
Reset	All zeros					

Figure 2-36. Debug Control Register 2 (DBCR2)

Table 2-33 provides bit definitions for DBCR2.

Table 2-33. DBCR2 Implementation-Specific Field Descriptions

Bits	Name	Description
34–35	DAC1ER	Data address compare 1 effective/real mode 00 DAC1 debug events are based on effective addresses. 01 Reserved on the e500. 10 DAC1 debug events are based on effective addresses and can occur only if MSR[DS]=0. 11 DAC1 debug events are based on effective addresses and can occur only if MSR[DS]=1.
38–39	DAC2ER	Data address compare 2 effective/real mode 00 DAC2 debug events are based on effective addresses. 01 Reserved on the e500. 10 DAC2 debug events are based on effective addresses and can occur only if MSR[DS]=0. 11 DAC2 debug events are based on effective addresses and can occur only if MSR[DS]=1.

2.13.2 Debug Status Register (DBSR)

The DBSR provides status information for debug events and for the most recent processor reset. The e500 implements the DBSR as it is defined by the Book E architecture, with the following exceptions:

- It does not implement IAC3 and IAC4 (DBSR[42–43]).
- Implementation-specific events that cause an unconditional debug event are defined in Table 2-34 (DBSR[UDE]).
- The MRR field is affected by the e500 definition of the HRESET signal, as defined in Table 2-34.

DBSR is shown in Figure 2-37.

SPR: 304 Access: Supervisor-only

	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
R/W	IDE	UDE	MRR		ICMP	BRT	IRPT	TRAP	IAC1	IAC2	—		DAC1R	DAC1W	DAC2R	DAC2W
Reset	0	0	undefined		0	0	0	0	0	0	0	0	0	0	0	0

	48	49														63
R/W	RET	—														
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 2-37. Debug Status Register (DBSR)

The DBSR is set through hardware but is read and cleared through software. DBSR is read using **mfspr**. DBSR bits are cleared by writing ones to them; writing zeros has no effect. Table 2-34 describes DBSR field definitions.

Table 2-34. DBSR Implementation-Specific Field Descriptions

Bits	Name	Description
33	UDE	Unconditional debug event. Set if an unconditional debug event occurred. If the $\overline{\text{UDE}}$ signal (level sensitive, active low) is asserted, DBSR[UDE] is affected as follows: MSR[DE] DBCR0[IDM] Action X 0 No action. 0 1 DBSR[UDE] is set. 1 1 DBSR[UDE] is set and a debug interrupt is taken.
34–35	MRR	Most recent reset. Set when a reset occurs. Undefined at power-up. The e500 implements HRESET as follows: 0x No hard reset occurred since this bit was last cleared by software. 1x The previous reset was a hard reset.

2.13.3 Instruction Address Compare Registers (IAC1–IAC4)

The e500 implements the IAC1 and IAC2 as they are defined by the Book E architecture; it does not implement IAC3 and IAC4.

A debug event may be enabled to occur upon an attempt to execute an instruction from an address specified in an IAC, inside or outside a range specified by IAC1 and IAC2, or to blocks of addresses specified by the combination of the IAC1 and IAC2. Because all instruction addresses are required to be word-aligned, the two low-order bits of the IACs are reserved and do not participate in the comparison to the instruction address.

2.13.4 Data Address Compare Registers (DAC1–DAC2)

The e500 implements the DAC1 and DAC2 as they are defined by the Book E architecture. A debug event may be enabled to occur upon loads, stores, or cache operations to an address specified in either DAC1 or DAC2, inside or outside a range specified by the DAC1 and DAC2, or to blocks of addresses specified by the combination of DAC1 and DAC2. The contents of DAC1 or DAC2 are compared to the address generated by a data storage access instruction.

2.14 SPE and SPFP APU Registers

The SPE and SPFP include the signal processing and embedded floating-point status and control register (SPEFSCR), described in Section 2.14.1, "Signal Processing and Embedded Floating-Point Status and Control Register (SPEFSCR)." The SPE implements a 64-bit accumulator, described in Section 2.14.2, "Accumulator (ACC)."

NOTE

The SPE APU and embedded floating-point APU functionality is implemented in all PowerQUICC III devices. However, these instructions will not be supported in devices subsequent to PowerQUICC III. Freescale Semiconductor strongly recommends that use of these instructions be confined to libraries and device drivers. Customer software that uses SPE or embedded floating-point APU instructions at the assembly level or that uses SPE intrinsics will require rewriting for upward compatibility with next-generation PowerQUICC devices.

Freescale Semiconductor offers a libmoto_e500 library that uses SPE instructions. Freescale will also provide libraries to support next-generation PowerQUICC devices.

2.14.1 Signal Processing and Embedded Floating-Point Status and Control Register (SPEFSCR)

The SPEFSCR is used by the SPE and embedded floating-point APUs. Vector floating-point instructions affect both the high element (bits 34-39) and low element floating-point status flags (bits 50–55). Single- and double-precision (e500v2 only) floating-point instructions affect only the low-element floating-point status flags and leave the high-element floating-point status flags undefined.

The SPEFSCR is shown in Figure 2-38.

SPR: 512 Access: Supervisor-only

	High-Word Error Bits										Status Bits					
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
R	SOVH	OVH	FGH	FXH	FINVH	FDBZH	FUNFH	FOVFH	—		FINXS	FINVS	FDBZS	FUNFS	FOVFS	MODE
W																
Reset	0	0	undefined		0	0	0	0	0	0	0	0	0	0	0	0

										Enable Bits						
	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
R	SOV	OV	FG	FX	FINV	FDBZ	FUNF	FOVF	—	FINXE	FINVE	FDBZE	FUNFE	FOVFE	FRMC	
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 2-38. Signal Processing and Embedded Floating-Point Status and Control Register (SPEFSCR)

Table 2-35 describes the SPEFSCR bits.

Table 2-35. SPEFSCR Field Descriptions

Bits	Name	Function
32	SOVH	Summary integer overflow high. Set whenever an instruction (except **mtspr**) sets OVH. SOVH remains set until it is cleared by an **mtspr** instruction.
33	OVH	Integer overflow high. An overflow occurred in the upper half of the register while executing an SPE integer instruction.
34	FGH	Embedded floating-point guard bit high. Floating-point guard bit from the upper half. The value is undefined if the processor takes a floating-point exception due to input error, floating-point overflow, or floating-point underflow.
35	FXH	Embedded floating-point sticky bit high. Floating bit from the upper half. The value is undefined if the processor takes a floating-point exception due to input error, floating-point overflow or floating-point underflow.
36	FINVH	Embedded floating-point invalid operation error high. Set when an input value on the high side is a NaN, Inf, or Denorm. Also set on a divide if both the dividend and divisor are zero.
37	FDBZH	Embedded floating-point divide-by-zero error high. Set if the dividend is non-zero and the divisor is zero.
38	FUNFH	Embedded floating-point underflow error high.
39	FOVFH	Embedded floating-point overflow error high.
40–41	—	Reserved, should be cleared.
42	FINXS	Embedded floating-point inexact sticky. FINXS = FINXS ǀ FGH ǀ FXH ǀ FG ǀ FX.
43	FINVS	Embedded floating-point invalid operation sticky. Location for software to use when implementing true IEEE floating point.
44	FDBZS	Embedded floating-point divide-by-zero sticky. FDBZS = FDBZS ǀ FDBZH ǀ FDBZ
45	FUNFS	Embedded floating-point underflow sticky. Storage location for software to use when implementing true IEEE floating point.

Table 2-35. SPEFSCR Field Descriptions (continued)

Bits	Name	Function
46	FOVFS	Embedded floating-point overflow sticky. Storage location for software to use when implementing true IEEE floating point.
47	MODE	Embedded floating-point mode (read-only on e500)
48	SOV	Integer summary overflow. Set whenever an SPE instruction (except **mtspr**) sets OV. SOV remains set until it is cleared by **mtspr[SPEFSCR]**.
49	OV	Integer overflow. An overflow occurred in the lower half of the register while a SPE integer instruction is being executed.
50	FG	Embedded floating-point guard bit. Floating-point guard bit from the lower half. The value is undefined if the processor takes a floating-point exception due to input error, floating-point overflow, or floating-point underflow.
51	FX	Embedded floating-point sticky bit. Floating bit from the lower half. The value is undefined if the processor takes a floating-point exception due to input error, floating-point overflow or floating-point underflow.
52	FINV	Embedded floating-point invalid operation error. Set when an input value on the high side is a NaN, Inf, or Denorm. Also set on a divide if both the dividend and divisor are zero.
53	FDBZ	Embedded floating-point divide-by-zero error. Set if the dividend is non-zero and the divisor is zero.
54	FUNF	Embedded floating-point underflow error
55	FOVF	Embedded floating-point overflow error
56	—	Reserved, should be cleared.
57	FINXE	Embedded floating-point inexact enable
58	FINVE	Embedded floating-point invalid operation/input error exception enable. 0 Exception disabled 1 Exception enabled. A floating-point data exception is taken if FINV or FINVH is set by a floating-point instruction.
59	FDBZE	Embedded floating-point divide-by-zero exception enable 0 Exception disabled 1 Exception enabled. A floating-point data exception is taken if FDBZ or FDBZH is set by a floating-point instruction
60	FUNFE	Embedded floating-point underflow exception enable 0 Exception disabled 1 Exception enabled. A floating-point data exception is taken if FUNF or FUNFH is set by a floating-point instruction.
61	FOVFE	Embedded floating-point overflow exception enable 0 Exception disabled 1 Exception enabled. a floating-point data exception is taken if FOVF or FOVFH is set by a floating-point instruction.
62–63	FRMC	Embedded floating-point rounding mode control 00 Round to nearest 01 Round toward zero 10 Round toward +infinity 11 Round toward –infinity

2.14.2 Accumulator (ACC)

The 64-bit architectural accumulator register holds the results of the multiply accumulate (MAC) forms of SPE integer instructions. The accumulator allows back-to-back execution of dependent MAC instructions, something that is found in the inner loops of DSP code such as finite impulse response (FIR) filters. The accumulator is partially visible to the programmer in that its results do not have to be explicitly read to use them. Instead, they are always copied into a 64-bit destination GPR specified as part of the instruction. The accumulator, however, has to be explicitly cleared when starting a new MAC loop. Based upon the type of instruction, an accumulator can hold either a single 64-bit value or a vector of two 32-bit elements.

The Initialize Accumulator instruction (**evmra**) is provided to initialize the accumulator.

2.15 Performance Monitor Registers (PMRs)

The Freescale Book E implementation standards defines a set of register resources used exclusively by the performance monitor. PMRs are similar to the SPRs defined in the Book E architecture and are accessed by **mtpmr** and **mfpmr**, which are also defined by the EIS. Table 2-36 lists supervisor-level PMRs. User-level software that attempts to read or write supervisor-level PMRs causes a privilege exception.

Table 2-36. Performance Monitor Registers—Supervisor Level

Abbreviation	Register Name	PMR Number	pmr[0–4]	pmr[5–9]	Section/Page
PMGC0	Performance monitor global control register 0	400	01100	10000	2.15.1/2-53
PMLCa0	Performance monitor local control a0	144	00100	10000	2.15.3/2-55
PMLCa1	Performance monitor local control a1	145	00100	10001	
PMLCa2	Performance monitor local control a2	146	00100	10010	
PMLCa3	Performance monitor local control a3	147	00100	10011	
PMLCb0	Performance monitor local control b0	272	01000	10000	2.15.5/2-56
PMLCb1	Performance monitor local control b1	273	01000	10001	
PMLCb2	Performance monitor local control b2	274	01000	10010	
PMLCb3	Performance monitor local control b3	275	01000	10011	
PMC0	Performance monitor counter 0	16	00000	10000	2.15.7/2-57
PMC1	Performance monitor counter 1	17	00000	10001	
PMC2	Performance monitor counter 2	18	00000	10010	
PMC3	Performance monitor counter 3	19	00000	10011	

User-level PMRs in Table 2-37 are read-only and are accessed with **mfpmr**. Attempting to write user-level registers in supervisor or user mode causes an illegal instruction exception.

Table 2-37. Performance Monitor Registers—User Level (Read-Only)

Abbreviation	Register Name	PMR Number	pmr[0–4]	pmr[5–9]	Section/Page
UPMGC0	User performance monitor global control register 0	384	01100	00000	2.15.2/2-54
UPMLCa0	User performance monitor local control a0	128	00100	00000	2.15.4/2-56
UPMLCa1	User performance monitor local control a1	129	00100	00001	
UPMLCa2	User performance monitor local control a2	130	00100	00010	
UPMLCa3	User performance monitor local control a3	131	00100	00011	
UPMLCb0	User performance monitor local control b0	256	01000	00000	2.15.6/2-57
UPMLCb1	User performance monitor local control b1	257	01000	00001	
UPMLCb2	User performance monitor local control b2	258	01000	00010	
UPMLCb3	User performance monitor local control b3	259	01000	00011	
UPMC0	User performance monitor counter 0	0	00000	00000	2.15.8/2-58
UPMC1	User performance monitor counter 1	1	00000	00001	
UPMC2	User performance monitor counter 2	2	00000	00010	
UPMC3	User performance monitor counter 3	3	00000	00011	

2.15.1 Global Control Register 0 (PMGC0)

The performance monitor global control register (PMGC0), shown in Figure 2-39, controls all performance monitor counters.

PMGC0 (PMR400)
UPMGC0 (PMR384)

Access: PMGC0: Supervisor-only
UPMGC0: Supervisor/user read-only

	32	33	34	35–50	51–52	53–54	55	56–63
R/W	FAC	PMIE	FCECE	—	TBSEL[1]	—	TBEE[1]	—
Reset	All zeros							

[1] e500v2 only

Figure 2-39. Performance Monitor Global Control Register 0 (PMGC0)/ User Performance Monitor Global Control Register 0 (UPMGC0)

PMGC0 is cleared by a hard reset. Reading this register does not change its contents. Table 2-38 describes the PMGC0 fields.

Table 2-38. PMGC0 Field Descriptions

Bits	Name	Description
32	FAC	Freeze all counters. When FAC is set by hardware or software, PMLCx[FC] maintains its current value until it is changed by software. 0 The PMCs are incremented (if permitted by other PM control bits). 1 The PMCs are not incremented.
33	PMIE	Performance monitor interrupt enable 0 Performance monitor interrupts are disabled. 1 Performance monitor interrupts are enabled and occur when an enabled condition or event occurs.
34	FCECE	Freeze counters on enabled condition or event 0 The PMCs can be incremented (if permitted by other PM control bits). 1 The PMCs can be incremented (if permitted by other PM control bits) only until an enabled condition or event occurs. When an enabled condition or event occurs, PMGC0[FAC] is set. It is up to software to clear FAC.
35–50	—	Reserved, should be cleared.
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). (e500v2 only) 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 multiprocessor systems in which TB registers are synchronized among processors, time base transition events can be used to correlate the performance monitor data obtained by the several processors. For this use, software must specify the same TBSEL value for all processors in the system. Because the time-base frequency is implementation-dependent, software should invoke a system service program to obtain the frequency before choosing a value for TBSEL.
53–54	—	Reserved, should be cleared.
55	TBEE	Time base transition event exception enable. (e500v2 only) 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.
56–63	—	Reserved, should be cleared.

2.15.2 User Global Control Register 0 (UPMGC0)

The contents of PMGC0 are reflected to UPMGC0, which is read by user-level software. UPMGC0 is read with the **mfpmr** instruction using PMR384.

2.15.3 Local Control A Registers (PMLCa0–PMLCa3)

The local control A registers 0–3 (PMLCa0–PMLCa3), shown in Figure 2-40, function as event selectors and give local control for the corresponding performance monitor counters. PMLCa works with the corresponding PMLCb register.

PMLCa0 (PMR144)
PMLCa1 (PMR145)
PMLCa2 (PMR146)
PMLCa3 (PMR147)

UPMLCa0 (PMR128)
UPMLCa1 (PMR129)
UPMLCa2 (PMR130)
UPMLCa3 (PMR131)

Access: PMLCa0–PMLCa3: Supervisor-only
UPMLCa0–UPMLCa3: Supervisor/user read-only

	32	33	34	35	36	37	38–40	41–47	48–63
R/W	FC	FCS	FCU	FCM1	FCM0	CE	—	EVENT	—
Reset	All zeros								

Figure 2-40. Local Control A Registers (PMLCa0–PMLCa3)/ User Local Control A Registers (UPMLCa0–UPMLCa3)

Table 2-39 describes the PMLCa fields.

Table 2-39. PMLCa0–PMLCa3 Field Descriptions

Bits	Name	Description
32	FC	Freeze counter 0 The PMC is incremented (if permitted by other PM control bits). 1 The PMC is not incremented.
33	FCS	Freeze counter in supervisor state 0 The PMC is incremented (if permitted by other PM control bits). 1 The PMC is not incremented if MSR[PR] = 0.
34	FCU	Freeze counter in user state 0 The PMC is incremented (if permitted by other PM control bits). 1 The PMC is not incremented if MSR[PR] = 1.
35	FCM1	Freeze counter while mark = 1 0 The PMC is incremented (if permitted by other PM control bits). 1 The PMC is not incremented if MSR[PMM] = 1.
36	FCM0	Freeze counter while mark = 0 0 The PMC is incremented (if permitted by other PM control bits). 1 The PMC is not incremented if MSR[PMM] = 0.
37	CE	Condition enable 0 PMC*x* overflow conditions cannot occur. (PMC*x* cannot cause interrupts, cannot freeze counters.) 1 Overflow conditions occur when the most-significant-bit of PMC*x* is equal to one. It is recommended that CE be cleared when counter PMC*x* is selected for chaining.
38–40	—	Reserved, should be cleared.
41–47	EVENT	Event selector. Up to 128 events selectable.
48–63	—	Reserved, should be cleared.

2.15.4 User Local Control A Registers (UPMLCa0–UPMLCa3)

The contents of PMLCa0–PMLCa3 are reflected to UPMLCa0–UPMLCa3, which are read by user-level software with mfpmr using PMR numbers in Table 2-37.

2.15.5 Local Control B Registers (PMLCb0–PMLCb3)

Local control B registers (PMLCb0–PMLCb3), shown in Figure 2-41, specify a threshold value and a multiple to apply to a threshold event selected for the corresponding performance monitor counter. For the e500, thresholding is supported only for PMC0 and PMC1. PMLCb works with the corresponding PMLCa.

PMLCb0 (PMR272) UPMLCb0 (PMR256)
PMLCb1 (PMR273) UPMLCb1 (PMR257)
PMLCb2 (PMR274) UPMLCb2 (PMR258)
PMLCb3 (PMR275) UPMLCb3 (PMR259)

Access: PMLCb0–PMLCb3: Supervisor-only
UPMLCb0–UPMLCb3: Supervisor/user read-only

	32–52	53–55	56–57	58–63
R W	—	THRESHMUL	—	THRESHOLD
Reset	All zeros			

Figure 2-41. Local Control B Registers (PMLCb0–PMLCb3)/ User Local Control B Registers (UPMLCb0–UPMLCb3)

Table 2-40 describes the PMLCb fields.

Table 2-40. PMLCb0–PMLCb3 Field Descriptions

Bits	Name	Description
32–52	—	Reserved, should be cleared.
53–55	THRESHMUL	Threshold multiple 000 Threshold field is multiplied by 1 (PMLCb*n*[THRESHOLD] × 1) 001 Threshold field is multiplied by 2 (PMLCb*n*[THRESHOLD] × 2) 010 Threshold field is multiplied by 4 (PMLCb*n*[THRESHOLD] × 4) 011 Threshold field is multiplied by 8 (PMLCb*n*[THRESHOLD] × 8) 100 Threshold field is multiplied by 16 (PMLCb*n*[THRESHOLD] × 16) 101 Threshold field is multiplied by 32 (PMLCb*n*[THRESHOLD] × 32) 110 Threshold field is multiplied by 64 (PMLCb*n*[THRESHOLD] × 64) 111 Threshold field is multiplied by 128 (PMLCb*n*[THRESHOLD] × 128)
56–57	—	Reserved, should be cleared.
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.

2.15.6 User Local Control B Registers (UPMLCb0–UPMLCb3)

The contents of PMLCb0–PMLCb3 are reflected to UPMLCb0–UPMLCb3, which are read by user-level software with mfpmr using the PMR numbers in Table 2-37.

2.15.7 Performance Monitor Counter Registers (PMC0–PMC3)

The performance monitor counter registers PMC0–PMC3, shown in Figure 2-42, are 32-bit counters that can be programmed to generate interrupt signals when they overflow. Each counter is enabled to count 128 events.

PMC0 (PMR16) UPMC0 (PMR0)
PMC1 (PMR17) UPMC1 (PMR1)
PMC2 (PMR18) UPMC2 (PMR2)
PMC3 (PMR19) UPMC3 (PMR3)

Access: PMC0–PMC3: Supervisor-only
UPMC0–UPMC3: Supervisor/user read-only

	32	33 — 63
R/W	OV	Counter value
Reset	All zeros	

Figure 2-42. Performance Monitor Counter Registers (PMC0–PMC3)/ User Performance Monitor Counter Registers (UPMC0–UPMC3)

Table 2-41 describes the PMC register fields.

Table 2-41. PMC0–PMC3 Field Descriptions

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

Counters overflow when the high-order bit (the sign bit) becomes set; that is, they reach the value 2,147,483,648 (0x8000_0000). However, an exception is not signaled unless PMGC0[PMIE] and PMLCa*n*[CE] are also set as appropriate.

The interrupts are masked by clearing MSR[EE]. An interrupt that is signaled while MSR[EE] is zero is not taken until MSR[EE] is set. Setting PMGC0[FCECE] forces counters to stop counting when an enabled condition or event occurs.

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 PMCx, an interrupt may be generated without an event counting having taken place.

PMC registers are accessed with **mtpmr** and **mfpmr** using the PMR numbers in Table 2-36.

2.15.8 User Performance Monitor Counter Registers (UPMC0–UPMC3)

The contents of PMC0–PMC3 are reflected to UPMC0–UPMC3, which are read by user-level software with the mfpmr instruction using the PMR numbers in Table 2-37.

2.16 Synchronization Requirements for SPRs

Synchronization requirements for accessing certain SPRs are shown in Table 2-42. Except for these SPRs, there are no synchronization requirements for accessing SPRs beyond those stated in Book E.

Table 2-42. Synchronization Requirements for SPRs

Registers	Instruction	Instruction Required Before	Instruction Required After
BBEAR	**mtspr bbear**	None	**isync**
BBTAR	**mtspr bbtar**	None	**isync**
BUCSR	**mtspr bucsr**	None	**isync**
DBCR0	**mtspr dbcr0**	None	**isync**
DBCR1	**mtspr dbcr1**	None	**isync**
HID0	**mtspr hid0**	None	**isync**
HID1	**mtspr hid1**	None	**isync**
L1CSR0	**mtspr l1csr0**	**msync**, **isync**	**isync**
L1CSR1	**mtspr l1csr1**	None	**isync**
MAS[0-4,6]	**mtspr mas[0–4,6]**	None	**isync**
MMUCSR0	**mtspr mmucsr0**	None	**isync**
PID0–PID2	**mtspr pid[0–2]**	None	**isync**
SPEFSCR	**mtspr spefscr**	None	**isync**

Chapter 3
Instruction Model

The e500 core complex is a 32-bit implementation of the Book E architecture as defined in the Book E architecture specification. This architecture specification allows for different processor implementations, which may provide extensions to or deviations from the architectural descriptions. This chapter provides information about the Book E architecture as it relates specifically to the e500v1 and e500v2. References to e500 apply to both the e500v1 and the e500v2.

Detailed, architectural descriptions of these instructions are provided in the *EREF: A Reference for Freescale Book E and the e500 Core*. The e500 core complex also implements several auxiliary processing units (APUs), which define additional instructions, registers, and interrupts. Instructions defined by APUs are summarized here. For a full description of APU functionality, see Chapter 10, "Auxiliary Processing Units (APUs)."

Specific information about how these instructions are executed is provided in Chapter 4, "Execution Timing."

3.1 Operand Conventions

This section describes operand conventions as they are represented in the Book E architecture. These conventions follow the basic descriptions in the classic PowerPC architecture with some changes in terminology. For example, distinctions between user and supervisor-level instructions are maintained, but the designations—UISA, VEA, and OEA—do not apply. Detailed descriptions are provided of conventions used for storing values in registers and memory, accessing processor registers, and representing data in these registers.

3.1.1 Data Organization in Memory and Data Transfers

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

Memory operands can be bytes, half words, words, or double words or, for the load/store multiple instruction type, a sequence of bytes or words. The address of a memory operand is the address of its first byte (that is, of its lowest-numbered byte). Operand length is implicit for each instruction.

3.1.2 Alignment and Misaligned Accesses

The operand of a single-register memory access instruction has an alignment boundary equal to its length. An operand's address is misaligned if it is not a multiple of its width.

The concept of alignment is also applied more generally to data in memory. For example, a 12-byte data item is said to be word-aligned if its address is a multiple of four.

Some instructions require their memory operands to have certain alignment. In addition, alignment can affect performance. For single-register memory access instructions, the best performance is obtained when memory operands are aligned.

Instructions are 32 bits (one word) long and must be word-aligned.

Memory operands for single-register memory access instructions have the characteristics described in Table 3-1.

Table 3-1. Address Characteristics of Aligned Operands

Operand	Operand Length	Addr[60–63] if Aligned
Byte	8 bits	xxxx[1]
Half word	2 bytes	xxx0
Word	4 bytes	xx00
Double word	8 bytes	x000

[1] An x in an address bit position indicates that the bit can be 0 or 1 independent of the state of other bits in the address.

Note that **lmw**, **stmw**, **lwarx**, and **stwcx.** instructions that are not word aligned cause an alignment exception.

3.1.3 e500 Floating-Point Implementation

The e500 does not implement the floating-point instructions as they are defined in Book E. Attempts to execute a Book E–defined floating-point instruction result in an illegal instruction exception.

The e500 implements the following:

- The vector single-precision floating-point APU supports single-precision vector (64-bit, two 32-bit operand) instructions.
- The scalar single-precision floating-point APU supports single-precision floating-point operations using the lower 32 bits of the GPRs.
- The scalar double-precision floating-point APU (implemented on the e500v2) supports double-precision floating-point operations using both halves of the GPRs.

These instructions are described in Section 3.8.1.4, “Embedded Floating-Point APU Instructions.” Unlike the PowerPC UISA, the SPFP APUs store floating-point operands as single-precision values in true 32-bit, single-precision format rather than in a 64-bit double-precision format used with FPRs.

NOTE

The SPE APU and embedded floating-point APU functionality is implemented in all PowerQUICC III devices. However, these instructions will not be supported in devices subsequent to PowerQUICC III. Freescale Semiconductor strongly recommends that use of these instructions be confined to libraries and device drivers. Customer software that uses SPE or embedded floating-point APU instructions at the assembly level or that uses SPE intrinsics will require rewriting for upward compatibility with next-generation PowerQUICC devices.

Freescale Semiconductor offers a libmoto_e500 library that uses SPE instructions. Freescale will also provide libraries to support next-generation PowerQUICC devices..

3.1.4 Unsupported Book E Instructions

Because the e500 core complex uses a 32-bit Book E core, all of the instructions defined only for 64-bit implementations of the Book E architecture are illegal in the e500. These instructions are not listed in Table 3-2. The e500 core complex takes an illegal instruction exception-type program interrupt upon encountering a 64-bit Book E instruction.

NOTE

Extended addressing forms of all load and store instructions are illegal because they calculate a 64-bit effective address. Also, except for certain vector instructions, all double-word instruction forms are illegal because only 64-bit implementations allow double-word operands.

The e500 does not support the Book E instructions listed in Table 3-2. An illegal instruction exception is generated if the processor attempts to execute one of these instructions. Some instructions have the following optional features indicated by square brackets:

- Condition register (CR) update—The dot (**.**) suffix on the mnemonic enables the update of the CR.
- Overflow option—The **o** suffix indicates that the overflow bit in the XER is enabled.

Table 3-2 lists 32-bit instructions that are not implemented in the e500.

Table 3-2. Unsupported Book E Instructions (32-Bit)

Name	Mnemonic
Floating Absolute Value [and record CR]	**fabs**[.]
Floating Add [Single] [and record CR]	**fadd**[**s**][.]
Floating Convert From Integer Double Word	**fcfid**
Floating Compare Ordered	**fcmpo**
Floating Compare Unordered	**fcmpu**
Floating Convert To Integer Double Word	**fctid**
Floating Convert To Integer Double Word [and round to Zero]	**fctid**[**z**]
Floating Convert To Integer Word [and round to Zero] [and record CR]	**fctiw**[**z**][.]
Floating Divide [Single] [and record CR]	**fdiv**[**s**][.]
Floating Multiply-Add [Single] [and record CR]	**fmadd**[**s**][.]
Floating Move Register [and record CR]	**fmr**[.]
Floating Multiply-Subtract [Single] [and record CR]	**fmsub**[**s**][.]
Floating Multiply [Single] [and record CR]	**fmul**[**s**][.]
Floating Negative Absolute Value [and record CR]	**fnabs**[.]
Floating Negate [and record CR]	**fneg**[.]
Floating Negative Multiply-Add [Single] [and record CR]	**fnmadd**[**s**][.]
Floating Negative Multiply-Subtract [Single] [and record CR]	**fnmsub**[**s**][.]
Floating Reciprocal Estimate Single [and record CR]	**fres**[.]
Floating Round to Single-Precision [and record CR]	**frsp**[.]
Floating Reciprocal Square Root Estimate [and record CR]	**frsqrte**[.]
Floating Select [and record CR]	**fsel**[.]
Floating Square Root [Single] [and record CR]	**fsqrt**[**s**][.]
Floating Subtract [Single] [and record CR]	**fsub**[**s**][.]
Load Floating-Point Double [with Update] [Indexed]	**lfd**[**u**][**x**]
Load Floating-Point Single [with Update] [Indexed]	**lfs**[**u**][**x**]
Load String Word Immediate	**lswi**
Load String Word Indexed	**lswx**
Move From APID Indirect	**mfapidi**
Move From Device Control Register	**mfdcr**
Move From FPSCR [and record CR]	**mffs**[.]
Move To Device Control Register	**mtdcr**
Move To FPSCR Bit 0 [and record CR]	**mtfsb0**[.]
Move To FPSCR Bit 1 [and record CR]	**mtfsb1**[.]
Move To FPSCR Field [Immediate] [and record CR]	**mtfsf**[**i**][.]
Store Floating-Point Double [with Update] [Indexed]	**stfd**[**u**][**x**]

Table 3-2. Unsupported Book E Instructions (32-Bit) (continued)

Name	Mnemonic
Store Floating-Point as Integer Word Indexed	**stfiwx**
Store Floating-Point Single [with Update] [Indexed]	**stfs**[**u**][**x**]
Store String Word Immediate	**stswi**
Store String Word Indexed	**stswx**

3.2 Instruction Set Summary

This chapter describes instructions and addressing modes defined for the e500. These instructions are divided into the following functional categories:

- Integer instructions—These include arithmetic and logical instructions. For more information, see Section 3.3.1.1, "Integer Instructions."
- Floating-point instructions—These include floating-point vector and scalar arithmetic instructions. See Section 3.8.1.4, "Embedded Floating-Point APU Instructions." The e500 does not support Book E–defined floating-point instructions or floating-point registers.
- Load and store instructions— See Section 3.3.1.2, "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 3.3.1.3, "Branch and Flow Control Instructions."
- Processor control instructions—These instructions are used for synchronizing memory accesses. See Section 3.3.1.5, "Processor Control Instructions."
- Memory synchronization instructions—These instructions are used for memory synchronizing. See Section 3.3.1.6, "Memory Synchronization Instructions."
- Memory control instructions—These instructions provide control of caches and TLBs. See Section 3.3.1.8, "Memory Control Instructions," and Section 3.3.2.2, "Supervisor-Level Memory Control Instructions."
- Signal processing instructions—These include a set of vector arithmetic and logic instructions optimized for signal processing tasks. See Section 3.8.1, "SPE and Embedded Floating-Point APUs."

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 4, "Execution Timing."

Integer instructions operate on word operands. The PowerPC architecture uses instructions that are 4 bytes long and word-aligned. It provides for byte, half-word, and word operand loads and stores between memory and a set of 32 general-purpose registers (GPRs).

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 C, "Simplified Mnemonics for PowerPC Instructions," for a complete list of simplified mnemonics. Programs written to be portable across the various assemblers for the PowerPC architecture should not assume the existence of mnemonics not described in that document.

3.2.1 Classes of Instructions

The e500 instructions belong to one of the following four classes:

- Defined instructions
- Allocated instructions
- Preserved instructions
- Reserved (illegal or no-op) instructions

These classes are defined in the "Instruction Model" chapter of the EREF. The class is determined by examining the primary opcode and any extended opcode. If the opcode, or combination of opcode and extended opcode, is not that of a defined, allocated, preserved, or reserved instruction, the instruction is illegal.

3.2.2 Definition of Boundedly Undefined

If instructions are encoded with incorrectly set bits in reserved fields, the results on execution can be said to be boundedly undefined. If a user-level program executes the incorrectly coded instruction, the resulting undefined results are bounded in that a spurious change from user to supervisor 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.

3.2.3 Synchronization Requirements

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

Changing a value in certain system registers and invalidating TLB entries can have the side effect of altering the context in which data addresses and instruction addresses are interpreted, and in which instructions are executed. For example, changing MSR[IS] from 0 to 1 has the side effect of changing address space. These effects need not occur in program order (that is, the strict order in which they occur in the program) and therefore may require explicit synchronization by software.

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 3-3 and Table 3-5.

A context-synchronizing interrupt (that is, any interrupt except non-recoverable machine check) can be used instead of a context-synchronizing instruction. If it is, references in this section to the synchronizing instruction should be interpreted as meaning the instruction at which the interrupt occurs. If no software synchronization is required either before or after a context-altering instruction, the phrase ‘the synchronizing instruction before (or after) the context-altering instruction’ should be interpreted as meaning the context-altering instruction itself.

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

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

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

No software synchronization is required before altering the MSR (except when altering the WE bit) because **mtmsr** is execution synchronizing. No software synchronization is required before most other alterations shown in Table 3-5, because all instructions before the context-altering instruction are fetched and decoded before the context-altering instruction is executed. (The processor must determine whether any of the preceding instructions are context-synchronizing.)

Table 3-3 identifies the software synchronization requirements for data access for all context-altering instructions.

Table 3-3. Data Access Synchronization Requirements

Context Altering Instruction or Event	Required Before	Required After	Notes
Interrupt	None	None	—
rfi	None	None	—
rfci	None	None	—
sc	None	None	—
mtmsr (PR)	None	CSI [1]	—
mtmsr (ME)	None	CSI [1]	2
mtmsr (DS)	None	CSI [1]	—
mtmsr (WE)	**msync**	**isync**	3
mtspr (DAC1, DAC2)	—	—	4
mtspr (DBCR0, DBCR2)	—	—	4
mtspr (DBSR)	—	—	4
mtspr (PID)	CSI [1]	CSI [1]	—
tlbivax	CSI [1]	CSI [1] and possibly **msync**	5,6
tlbwe	CSI [1]	CSI [1] and possibly **msync**	5, 6

1 CSI indicates any context-synchronizing instruction (that is, **sc**, **isync**, **rfci**, or **rfi**).

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 See Section 6.4.1, "Software Considerations for Power Management."

4 Synchronization requirements for changing any of the debug facility registers are implementation dependent.

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

6 The context-synchronizing instruction after **tlbwe** 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** or **tlbivax** must be followed by an **msync** and by a context-synchronizing instruction.

3.2.3.1 Synchronization Requirements for e500-Specific SPRs

Software requirements for synchronization before and after accessing certain SPRs are shown in Table 3-4. Except for these registers, there are no synchronization requirements for accessing SPRs beyond those stated in Book E and described in Section 3.2.3, "Synchronization Requirements."

Table 3-4. Synchronization Requirements for e500-Specific SPRs

Registers	Instruction	Instruction Required Before	Instruction Required After
BBEAR	**mtspr bbear**	None	**isync**
BBTAR	**mtspr bbtar**	None	**isync**

Table 3-4. Synchronization Requirements for e500-Specific SPRs (continued)

Registers	Instruction	Instruction Required Before	Instruction Required After
BUCSR	**mtspr bucsr**	None	CSI[1]
DBCR0	**mtspr dbcr0**	None	CSI [1]
DBCR1	**mtspr dbcr1**	None	CSI [1]
HID0	**mtspr hid0**	CSI [1]	CSI [1]
HID1	**mtspr hid1**	**msync**	CSI [1]
L1CSR0	**mtspr l1csr0**	**msync**, **isync**	CSI [1]
L1CSR1	**mtspr l1csr1**	None	**isync**
MMUCSR0	**mtspr mmucsr0**	CSI [1]	CSI [1]
PID0–PID2	**mtspr pid[0–2]**	None	**isync**
SPEFSCR	**mtspr spefscr**	None	**isync**

[1] CSI indicates any context-synchronizing instruction (that is, **sc**, **isync**, **rfci**, or **rfi**).

Table 3-5 below identifies the software synchronization requirements for instruction fetch and/or execution for all context-altering instructions.

Table 3-5. Instruction Fetch and/or Execution Synchronization Requirements

Context Altering Instruction or Event	Required Before	Required After	Notes
Interrupt	None	None	
mtmsr (CE)	None	None	1
mtmsr (DE)	None	CSI [4]	
mtmsr (EE)	None	None	1
mtmsr (FE0)	None	CSI [4]	
mtmsr (FE1)	None	CSI [4]	
mtmsr (FP)	None	CSI [4]	
mtmsr (IS)	None	CSI [4]	2
mtmsr (ME)	None	CSI [4]	5,3
mtmsr (PR)	None	CSI [4]	
mtmsr (WE)	The e500 requires an **msync**.	The e500 requires an **isync**.	5,6
mtpmr	None	CSI[7]	
mtspr (DAC*n*)	—	CSI	8
mtspr (DBCR*n*)	—	CSI	8
mtspr (DBSR)	—	CSI	8
mtspr (DEC)	None	None	9
mtspr (IAC*n*)	—	CSI	8
mtspr (IVOR*n*)	None	None	
mtspr (IVPR)	None	None	

Table 3-5. Instruction Fetch and/or Execution Synchronization Requirements (continued)

Context Altering Instruction or Event	Required Before	Required After	Notes
mtspr (PID)	None	CSI [4]	2
mtspr (TCR)	None	None	9
mtspr (TSR)	None	None	9
rfci	None	None	
rfi	None	None	
sc	None	None	
tlbivax	None	CSI [4] or **msync**	10,11
tlbwe	None	CSI [4] or **msync**	10,11
wrtee, wrteei	None	None	1

1 The effect of changing MSR[EE] or MSR[CE] is immediate.
If **mtmsr**, **wrtee**, or **wrteei** clears MSR[EE], an external input, decrementer or fixed-interval timer interrupt does not occur after the instruction is executed.
If **mtmsr**, **wrtee**, or **wrteei** changes MSR[EE] from 0 to 1 when an external input, decrementer, fixed-interval timer, or higher priority enabled exception exists, the corresponding interrupt occurs immediately after the **mtmsr**, **wrtee**, or **wrteei** is executed, and before the next instruction executes in the program that set MSR[EE].

2 The alteration must not cause an implicit branch in real address space. Thus the real address of the context-altering instruction and of each subsequent instruction, up to and including the next context-synchronizing instruction, must be independent of whether the alteration has taken effect.

3 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 so may not be context-synchronizing.

4 CSI indicates any context-synchronizing instruction (that is, **sc**, **isync**, **rfci**, **rfmci**, or **rfi**).

5 Synchronization requirements for changing the wait state enable are implementation-dependent.

6 For more information about synchronization requirements with **mtmsr** (**WE**), see Section 6.4.1, "Software Considerations for Power Management."

7 CSI indicates any context-synchronizing instruction (that is, **sc**, **isync**, **rfci**, **rfmci**, or **rfi**).

8 Synchronization requirements for changing any debug facility registers are implementation-dependent.

9 The elapsed time between the DEC reaching zero, or the transition of the selected time base bit for the fixed-interval or watchdog timer, and the signalling of the decrementer, fixed-interval timer, or watchdog timer exception is not defined.

10 For data accesses, the context-synchronizing instruction before the **tlbwe** or **tlbivax** instruction ensures that all accesses due to preceding instructions have completed to a point at which they have reported all exceptions they will cause. See Section 3.2.3.2, "Synchronization with **tlbwe** and **tlbivax** Instructions."

11 The context-synchronizing instruction after **tlbwe** or **tlbivax** ensures that subsequent accesses (data and instruction) use the updated value in the affected TLB entries. It does not ensure that all accesses previously translated by the TLB entries being updated have completed with respect to memory; if these completions must be ensured, **tlbwe** or **tlbivax** must be followed by an **msync** and by a context-synchronizing instruction. See Section 3.2.3.2, "Synchronization with **tlbwe** and **tlbivax** Instructions."

3.2.3.2 Synchronization with tlbwe and tlbivax Instructions

The following sequence shows why, for data accesses, it is necessary to ensure that all memory accesses due to instructions before the **tlbwe** or **tlbivax** have completed to a point at which they have reported all exceptions they cause. Assume that 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** or **tlbivax** that invalidates the corresponding TLB entry.
3. The load or store instruction finally executes, and gets a TLB miss exception.

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

3.2.3.3 Context Synchronization

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

1. The operation is not initiated or, in the case of **isync**, does not complete until all instructions already in execution have completed to a point at which they have reported all exceptions they cause.
2. The instructions that precede the operation complete execution 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 5.11, “Exception Priorities.”
4. The instructions that follow the operation are fetched and executed in the context established by the operation as required by the sequential execution model. (This requirement dictates that any prefetched instructions be discarded and that any effects and side effects of executing them speculatively may also be discarded, except as described in the “Cache and MMU Background” chapter in the EREF.)

As described in Section 3.2.3.4, “Execution Synchronization,” a context-synchronizing operation is necessarily execution synchronizing. Unlike **msync** and **mbar**, such operations do not affect the order of memory accesses with respect to other mechanisms.

3.2.3.4 Execution Synchronization

An instruction is execution synchronizing if it satisfies items 1 and 2 of the definition of context synchronization (see Section 3.2.3.3, “Context Synchronization”). **msync** is treated like **isync** with respect to item 1 (that is, the conditions described in item 1 apply to completion of **msync**).

Execution synchronizing instructions include **msync**, **mtmsr**, **wrtee**, and **wrteei**. All context-synchronizing instructions are execution synchronizing.

Unlike a context-synchronizing operation, an execution synchronizing instruction need not ensure that the 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.

3.2.3.5 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.

Examples of interrupts that can be caused directly by the execution of an instruction include but are not limited to the following:

- 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 (privileged instruction exception-type program interrupt)
- An attempt by an application program to access a privileged SPR (privileged instruction exception-type program interrupt)
- An attempt by an application program to access an SPR that does not exist (unimplemented operation instruction exception-type program interrupt)
- An attempt by a system program to access an SPR that does not exist (boundedly undefined)
- Execution of a defined instruction using an invalid form (illegal instruction exception-type program interrupt, unimplemented operation exception-type program interrupt, or privileged instruction exception-type program interrupt)
- An attempt to access a memory location that is either unavailable (instruction TLB error interrupt or data TLB error interrupt) or not permitted (instruction storage interrupt or data storage interrupt)
- An attempt to access memory with an effective address alignment not supported by the implementation (alignment interrupt)
- Execution of a system call instruction (system call interrupt)
- Execution of a **trap** instruction whose trap condition is met (trap type program interrupt)
- Execution of a defined instruction that is not implemented by the implementation (illegal instruction exception or unimplemented operation exception-type program interrupt)
- Execution of an allocated instruction that is not implemented by the implementation (illegal instruction exception or unimplemented operation exception-type program interrupt)

- Execution of an allocated instruction that causes an auxiliary enabled exception (enabled exception-type program interrupt).

APUs, such as the SPE, may define additional instruction-caused exceptions and interrupts. The invocation of an interrupt is precise, except that if one of the imprecise modes for invoking the floating-point enabled exception-type program interrupt is in effect the invocation of the floating-point enabled exception-type program interrupt may be imprecise. When the interrupt is invoked imprecisely, the excepting instruction does not appear to complete before the next instruction starts (because one of the effects of the excepting instruction, namely the invocation of the interrupt, has not yet occurred).

Chapter 5, "Interrupts and Exceptions," describes interrupt conditions in detail.

3.3 Instruction Set Overview

This section provides a overview of the PowerPC instructions implemented in the e500 and highlights any special information with respect to how the e500 implements a particular instruction. Note that some instructions have the following optional features:

- CR update—The dot (**.**) suffix on the mnemonic enables the update of the CR.
- Overflow option—The **o** suffix indicates that the overflow bit in the XER is enabled.

3.3.1 Book E User-Level Instructions

This section discusses the user-level instructions defined in the Book E architecture.

3.3.1.1 Integer Instructions

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

- Integer arithmetic instructions
- Integer compare instructions
- Integer logical instructions
- Integer rotate and shift instructions

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

3.3.1.1.1 Integer Arithmetic Instructions

Table 3-6 lists the integer arithmetic instructions for the PowerPC processors.

Table 3-6. Integer Arithmetic Instructions

Name	Mnemonic	Syntax
Add	**add** (**add. addo addo.**)	**r**D,**r**A,**r**B
Add Carrying	**addc** (**addc. addco addco.**)	**r**D,**r**A,**r**B
Add Extended	**adde** (**adde. addeo addeo.**)	**r**D,**r**A,**r**B
Add Immediate	**addi**	**r**D,**r**A,SIMM
Add Immediate Carrying	**addic**	**r**D,**r**A,SIMM
Add Immediate Carrying and Record	**addic.**	**r**D,**r**A,SIMM
Add Immediate Shifted	**addis**	**r**D,**r**A,SIMM
Add to Minus One Extended	**addme** (**addme. addmeo addmeo.**)	**r**D,**r**A
Add to Zero Extended	**addze** (**addze. addzeo addzeo.**)	**r**D,**r**A
Divide Word	**divw** (**divw. divwo divwo.**)	**r**D,**r**A,**r**B
Divide Word Unsigned	**divwu divwu. divwuo divwuo.**	**r**D,**r**A,**r**B
Multiply High Word	**mulhw** (**mulhw.**)	**r**D,**r**A,**r**B
Multiply High Word Unsigned	**mulhwu** (**mulhwu.**)	**r**D,**r**A,**r**B
Multiply Low Immediate	**mulli**	**r**D,**r**A,SIMM
Multiply Low Word	**mullw** (**mullw. mullwo mullwo.**)	**r**D,**r**A,**r**B
Negate	**neg** (**neg. nego nego.**)	**r**D,**r**A
Subtract From	**subf** (**subf. subfo subfo.**)	**r**D,**r**A,**r**B
Subtract from Carrying	**subfc** (**subfc. subfco subfco.**)	**r**D,**r**A,**r**B
Subtract from Extended	**subfe** (**subfe. subfeo subfeo.**)	**r**D,**r**A,**r**B
Subtract from Immediate Carrying	**subfic**	**r**D,**r**A,SIMM
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 can be achieved by using an **addi** instruction with the immediate operand negated. Simplified mnemonics are provided that include this negation. Subtract instructions subtract the second operand (**r**A) from the third operand (**r**B). Simplified mnemonics are provided in which the third operand is subtracted from the second. See Appendix C, "Simplified Mnemonics for PowerPC Instructions," for examples.

According to Book E, an implementation that executes instructions with the overflow exception enable bit (OE) set or that sets the carry bit (CA) can either execute these instructions slowly or prevent execution of the subsequent instruction until the operation completes. Chapter 4, "Execution Timing," describes how the e500 handles CR dependencies. The summary overflow (SO) and overflow (OV) bits in the XER are set to reflect an overflow condition of a 32-bit result only if the instruction's OE bit is set.

3.3.1.1.2 Integer Compare Instructions

The integer compare instructions algebraically or logically compare the contents of register **r**A with either the zero-extended value of the UIMM operand, the sign-extended value of the SIMM operand, or the contents of **r**B. The comparison is signed for **cmpi** and **cmp** and unsigned for **cmpli** and **cmpl**. Table 3-7 lists integer compare instructions. Note that the L bit must be 0 for 32-bit implementations.

Table 3-7. Integer 32-Bit Compare Instructions (L = 0)

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 for the integer compare instructions see Appendix C, "Simplified Mnemonics for PowerPC Instructions."

3.3.1.1.3 Integer Logical Instructions

The logical instructions shown in Table 3-8 perform bit-parallel operations on the specified operands. Logical instructions with the CR updating enabled (uses dot suffix) and instructions **andi.** and **andis.** set CR field CR0 to characterize the result of the logical operation. Logical instructions do not affect XER[SO], XER[OV], or XER[CA].

See Appendix C, "Simplified Mnemonics for PowerPC Instructions," for simplified mnemonic examples for integer logical operations.

Table 3-8. Integer Logical Instructions

Name	Mnemonic	Syntax	Implementation Notes
AND	**and (and.)**	**r**A,**r**S,**r**B	—
AND Immediate	**andi.**	**r**A,**r**S,UIMM	—
AND Immediate Shifted	**andis.**	**r**A,**r**S,UIMM	—
AND with Complement	**andc** (**andc.**)	**r**A,**r**S,**r**B	—
Count Leading Zeros Word	**cntlzw (cntlzw.**)	**r**A,**r**S	—
Equivalent	**eqv** (**eqv.**)	**r**A,**r**S,**r**B	—
Extend Sign Byte	**extsb** (**extsb.**)	**r**A,**r**S	—
Extend Sign Half Word	**extsh** (**extsh.**)	**r**A,**r**S	—
NAND	**nand** (**nand.**)	**r**A,**r**S,**r**B	—
NOR	**nor** (**nor.**)	**r**A,**r**S,**r**B	—

Table 3-8. Integer Logical Instructions (continued)

Name	Mnemonic	Syntax	Implementation Notes
OR	**or** (**or.**)	rA,rS,rB	—
OR Immediate	**ori**	rA,rS,UIMM	Book E defines **ori r0,r0,0** as the preferred form for a no-op. The dispatcher may discard this instruction and dispatch it only to the completion queue but not to any execution unit.
OR Immediate Shifted	**oris**	rA,rS,UIMM	—
OR with Complement	**orc** (**orc.**)	rA,rS,rB	—
XOR	**xor** (**xor.**)	rA,rS,rB	—
XOR Immediate	**xori**	rA,rS,UIMM	—
XOR Immediate Shifted	**xoris**	rA,rS,UIMM	—

3.3.1.1.4 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 3-9, rotate the contents of a register. The result is either inserted into the target register under control of a mask (if a mask bit is set the associated bit of the rotated data is placed into the target register, and if the mask bit is cleared the associated bit in the target register is unchanged) or ANDed with a mask before being placed into the target register. Appendix C, "Simplified Mnemonics for PowerPC Instructions," lists simplified mnemonics that allow simpler coding of often-used functions such as clearing the left- or right-most bits of a register, left or right justifying an arbitrary field, and simple rotates and shifts.

Table 3-9. Integer Rotate Instructions

Name	Mnemonic	Syntax
Rotate Left Word Immediate then AND with Mask	**rlwinm** (**rlwinm.**)	rA,rS,SH,MB,ME
Rotate Left Word then AND with Mask	**rlwnm** (**rlwnm.**)	rA,rS,rB,MB,ME
Rotate Left Word Immediate then Mask Insert	**rlwimi** (**rlwimi.**)	rA,rS,SH,MB,ME

The integer shift instructions (Table 3-10) perform left and right shifts. Immediate-form logical (unsigned) shift operations are obtained by specifying masks and shift values for certain rotate instructions. Simplified mnemonics (shown in Appendix C, "Simplified Mnemonics for PowerPC Instructions") are provided to simplify coding of such shifts. The integer shift instructions are summarized in Table 3-10.

Table 3-10. Integer Shift Instructions

Name	Mnemonic	Syntax
Shift Left Word	**slw** (**slw.**)	rA,rS,rB
Shift Right Word	**srw** (**srw.**)	rA,rS,rB
Shift Right Algebraic Word Immediate	**srawi** (**srawi.**)	rA,rS,SH
Shift Right Algebraic Word	**sraw** (**sraw.**)	rA,rS,rB

3.3.1.2 Load and Store Instructions

Load and store instructions are issued and translated in program order; however, the accesses can occur out of order. Synchronizing instructions are provided to enforce strict ordering. The e500 supports load and store instructions as follows:

- Integer load instructions
- Integer store instructions
- Integer load and store with byte-reverse instructions
- Integer load and store multiple instructions
- Memory synchronization instructions
- SPE APU load and store instructions for reading and writing 64-bit GPRs. These are described in Section 3.8.1, "SPE and Embedded Floating-Point APUs."

The e500 does not implement Book E floating-point load and store instructions.

Implementation Notes—The following describes how the e500 handles misalignment:

The e500 provides hardware support for misaligned memory accesses. It performs those accesses within a single cycle if the operand lies within a double-word boundary. Misaligned memory accesses that cross a double-word boundary degrade performance.

Although many misaligned memory accesses are supported in hardware, the frequent use of them is discouraged because they can compromise the overall performance of the processor. Only one outstanding misalignment at a time is supported, which means it is non-pipelined.

Accesses that cross a translation boundary can be restarted. That is, a misaligned access that crosses a page boundary is completely restarted if the second portion of the access causes a page fault. This can cause the first access to be repeated.

3.3.1.2.1 Self-Modifying Code

When a processor modifies any memory location that can contain an instruction, software must ensure that the instruction cache is made consistent with data memory and that the modifications are made visible to the instruction fetching mechanism. This must be done even if the cache is disabled or if the page is marked caching-inhibited.

The following instruction sequence can be used to accomplish this when the instructions being modified are in memory that is memory-coherency required and one processor both modifies the instructions and executes them. (Additional synchronization is needed when one processor modifies instructions that another processor will execute.)

The following sequence synchronizes the instruction stream (using either **dcbst** or **dcbf**):

```
dcbst (or dcbf)  |update memory
msync            |wait for update
icbi             |remove (invalidate) copy in instruction cache
msync            |ensure the ICBI invalidate is complete
isync            |remove copy in own instruction buffer
```

These operations are required because the data cache is a write-back cache. Because instruction fetching bypasses the data cache, changes to items in the data cache cannot be reflected in memory until the fetch operations complete. The **msync** after the **icbi** is required to ensure that the **icbi** invalidation has completed in the instruction cache.

Special care must be taken to avoid coherency paradoxes in systems that implement unified secondary caches (like the e500), and designers should carefully follow the guidelines for maintaining cache coherency discussed in Chapter 11, "L1 Caches."

3.3.1.2.2 Integer Load and Store Address Generation

Integer load and store operations generate effective addresses using register indirect with immediate index mode, register indirect with index mode, or register indirect mode, which are described as follows:

- Register indirect with immediate index addressing for integer loads and stores. Instructions using this addressing mode contain a signed 16-bit immediate index (d operand), which is sign extended and added to the contents of a general-purpose register specified in the instruction (**r**A operand), to generate the effective address. If the **r**A field of the instruction specifies **r0**, a value of zero is added to the immediate index (d operand) in place of the contents of **r0**. The option to specify **r**A or 0 is shown in the instruction descriptions as (**r**A|0). Figure 3-1 shows how an effective address is generated using this addressing mode.

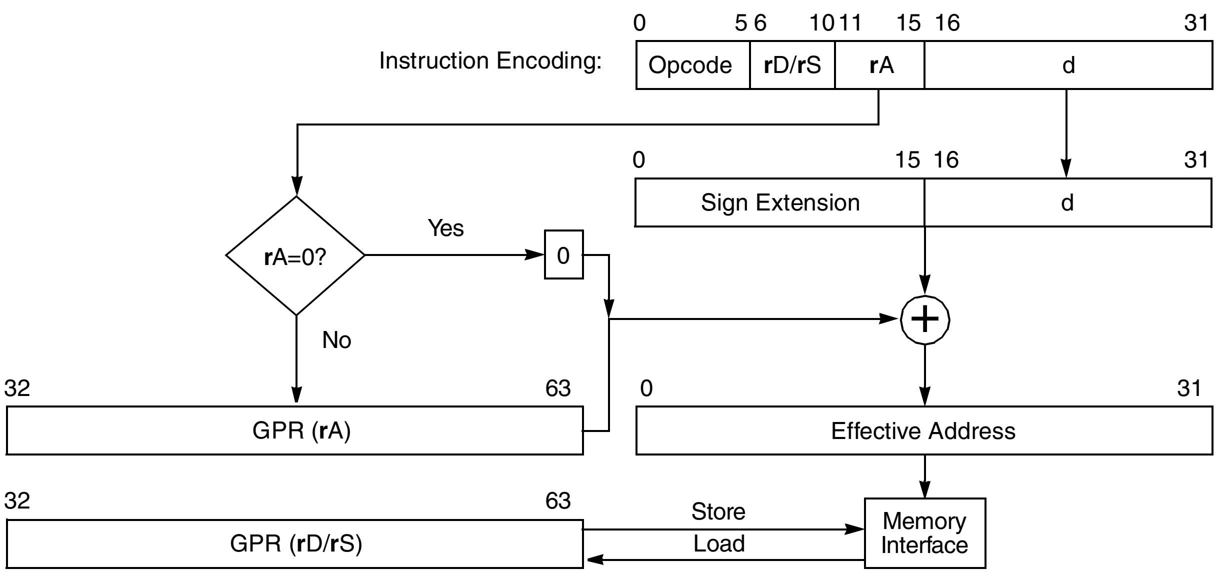


Figure 3-1. Register Indirect with Immediate Index Addressing for Integer Loads/Stores

- Register indirect with index addressing for integer loads and stores. Instructions using this addressing mode cause the contents of two general-purpose registers (specified as operands **r**A and **r**B) to be added in the generation of the effective address. A zero in place of the **r**A operand causes a zero to be added to the contents of the general-purpose register specified in operand **r**B. The option to specify **r**A or 0 is shown in the instruction descriptions as (**r**A|0).

  Figure 3-2 shows how an effective address is generated using this addressing mode.

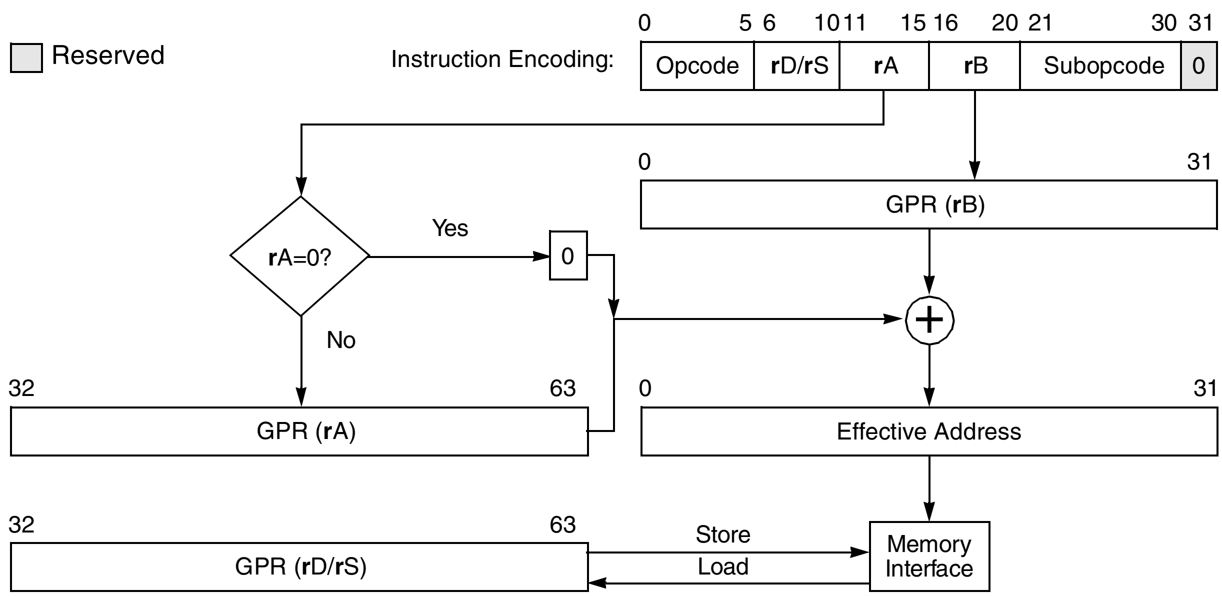


Figure 3-2. Register Indirect with Index Addressing for Integer Loads/Stores

- Register indirect addressing for integer loads and stores. Instructions using this addressing mode use the contents of the GPR specified by the **r**A operand as the effective address. A zero in the **r**A operand causes an effective address of zero to be generated. The option to specify **r**A or 0 is shown in the instruction descriptions as (**r**A|0).

Figure 3-3 shows how an effective address is generated using register indirect addressing.

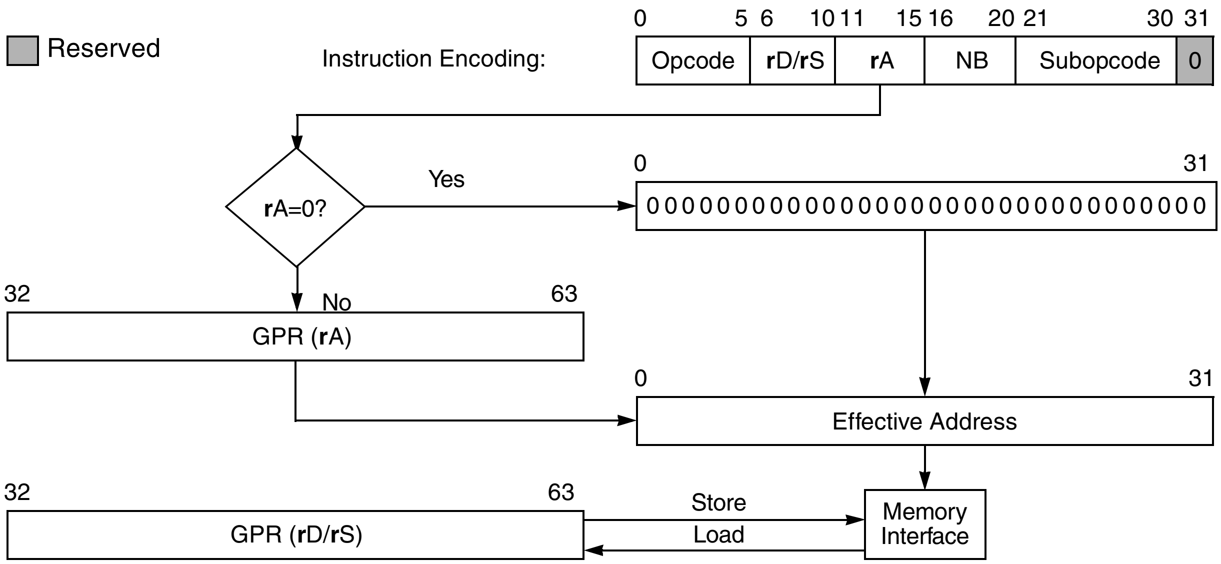


Figure 3-3. Register Indirect Addressing for Integer Loads/Stores

The instruction model chapter in the EREF describes effective address calculation. Note that in some implementations, operations that are not naturally aligned can suffer performance degradation. Section 5.7.6, “Alignment Interrupt,” for additional information about load and store address alignment interrupts.

3.3.1.2.3 Integer Load Instructions

Table 3-11 summarizes the integer load instructions.

Table 3-11. Integer Load Instructions

Name	Mnemonic	Syntax
Load Byte and Zero	**lbz**	**r**D,**d**(**r**A)
Load Byte and Zero Indexed	**lbzx**	**r**D,**r**A,**r**B
Load Byte and Zero with Update	**lbzu**	**r**D,**d**(**r**A)
Load Byte and Zero with Update Indexed	**lbzux**	**r**D,**r**A,**r**B
Load 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)

Table 3-11. Integer Load Instructions (continued)

Name	Mnemonic	Syntax
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

The following notes describe the e500 implementation of integer load instructions:

- Book E cautions programmers that some implementations of the architecture can execute the load half algebraic (**lha**, **lhax**) instructions with greater latency than other types of load instructions. This is not the case for the e500; these instructions operate with the same latency as other load instructions.
- Book E cautions programmers that some implementations can run the load/store byte-reverse (**lhbrx**, **lbrx**, **sthbrx**, **stwbrx**) instructions with greater latency than other types of load/store instructions. This is not the case for the e500. These instructions operate with the same latency as the other load/store instructions.
- The Book E architecture defines **lwarx** and **stwcx.** as a way to update memory atomically. In the e500, reservations are made on behalf of aligned 32-byte sections of the memory address space. Executing **lwarx** and **stwcx.** to a page marked write-through causes a data storage interrupt if the page is marked cacheable write-through (WIM = 10x), but as with other memory accesses, data storage interrupts can result for other reasons such as protection violations or page faults.

3.3.1.2.4 Integer Store Instructions

For integer store instructions, the **r**S contents are stored into the byte, half word, word, or double word in memory addressed by the EA (effective address). Many store instructions have an update form in which **r**A is updated with the EA. For these forms, the following rules apply:

- If **r**A ≠ 0, the effective address is placed into **r**A.
- If **r**S = **r**A, the contents of register **r**S are copied to the target memory element and the generated EA is placed into **r**A (**r**S).

The Book E architecture defines store with update instructions with **r**A = 0 as an invalid form. In addition, it defines integer store instructions with the CR update option enabled (Rc field, bit 31, in the instruction encoding = 1) to be an invalid form. Table 3-12 summarizes integer store instructions.

Table 3-12. Integer Store Instructions

Name	Mnemonic	Syntax
Store Byte	**stb**	**r**S,**d(r**A**)**
Store Byte Indexed	**stbx**	**r**S,**r**A,**r**B

Table 3-12. Integer Store Instructions (continued)

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

3.3.1.2.5 Integer Load and Store with Byte-Reverse Instructions

Table 3-13 describes integer load and store with byte-reverse instructions. These books were defined in part to support the original PowerPC definition of little-endian byte ordering. Note that Book E supports true little endian on a per-page basis.

Table 3-13. Integer Load and Store with Byte-Reverse Instructions

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

3.3.1.2.6 Integer Load and Store Multiple Instructions

The load/store multiple instructions are used to move blocks of data to and from the GPRs. The load multiple and store multiple instructions can have operands that require memory accesses crossing a 4-Kbyte page boundary. As a result, these instructions can be interrupted by a data storage interrupt associated with the address translation of the second page. Note that if one of these instructions is interrupted, it may be restarted, requiring multiple memory accesses.

The Book E architecture defines the Load Multiple Word (**lmw**) instruction with **r**A in the range of registers to be loaded as an invalid form. Load and store multiple accesses must be word aligned; otherwise, they cause an alignment exception.

The load/store multiple instructions are listed in Table 3-14.

Table 3-14. 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**)

3.3.1.3 Branch and Flow Control Instructions

Some branch instructions can redirect instruction execution conditionally based on the value of bits in the CR. Information about branch instruction address calculation is provided in the EREF.

3.3.1.3.1 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 3-16 as having the value *y*, is used by some implementations for branch prediction as described below.

NOTE

The e500 does not implement the static branch prediction defined in Book E and described here. In the e500, the BO operand is ignored for branch prediction. The e500 instead implements dynamic branch prediction as part of the branch table buffer (BTB), described in Section 4.4.1, "Branch Unit Execution."

Table 3-15. 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	Used for the *y* bit, which provides a hint about whether a conditional branch is likely to be taken (static branch prediction) and may be used by some implementations to improve performance. The e500 does not use static branch prediction and ignores this bit.

The encodings for the BO operands are shown in Table 3-16.

Table 3-16. BO Operand Encodings

BO	Description
0000*y*	Decrement the CTR, then branch if the decremented CTR ≠ 0 and the condition is FALSE.
0001*y*	Decrement the CTR, then branch if the decremented CTR = 0 and the condition is FALSE.

Table 3-16. BO Operand Encodings (continued)

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

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.
The *y* bit provides a hint about whether a conditional branch is likely to be taken and may be used by some implementations to improve performance.

The branch always encoding of the BO operand does not have a *y* bit.

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 the branch instructions contain immediate addressing operands, the target addresses can be computed sufficiently ahead of the branch instruction that instructions can be fetched along the target path. If the branch instructions use the link and count registers, instructions along the target path can be fetched if the link or count register 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 effective address 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.

3.3.1.3.2 Branch Instructions

Table 3-17 lists branch instructions provided by the Book E processors. A set of simplified mnemonics and symbols is provided for the most frequently used forms of branch conditional, compare, trap, rotate and shift, and certain other instructions; see Appendix C, "Simplified Mnemonics for PowerPC Instructions." Note that the e500 does not use the BO operand for static branch prediction.

Table 3-17. 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

Note that the e500 implements the Integer Select instruction, **isel**, which can be used to more efficiently handle sequences with multiple conditional branches. Its syntax is given in Section 3.8.2, "Integer Select (**isel**) APU." A detailed description including an example of how **isel** can be used can be found in the APUs chapter of the EREF.

3.3.1.3.3 Condition Register Logical Instructions

CR logical instructions, shown in Table 3-18, and the Move Condition Register Field (**mcrf**) instruction are also defined as flow control instructions.

Table 3-18. 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

Note that if the LR update option is enabled for any of these instructions, the Book E architecture defines these forms of the instructions as invalid.

3.3.1.3.4 Trap Instructions

The trap instructions shown in Table 3-19 test for a specified set of conditions. If any of the conditions tested by a trap instruction are met, the system trap type program interrupt is taken. For more information, see Section 5.7.7, "Program Interrupt." If the tested conditions are not met, instruction execution continues normally. See Appendix C, "Simplified Mnemonics for PowerPC Instructions."

Table 3-19. Trap Instructions

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

3.3.1.4 System Linkage Instruction

The System Call (**sc**) instruction permits a program to call on the system to perform a service; see Table 3-20 and Section 3.3.2.1, "System Linkage Instructions."

Table 3-20. System Linkage Instruction

Name	Mnemonic	Syntax
System Call	**sc**	—

Executing this instruction causes the system call interrupt handler to be invoked. For more information, see Section 5.7.8, "System Call Interrupt."

3.3.1.5 Processor Control Instructions

Processor control instructions are used to read from and write to the CR, machine state register (MSR), and special-purpose registers (SPRs).

3.3.1.5.1 Move to/from Condition Register Instructions

Table 3-21 summarizes the instructions for reading from or writing to the CR.

Table 3-21. Move to/from Condition Register Instructions

Name	Mnemonic	Syntax
Move to Condition Register Fields	**mtcrf**	CRM,**r**S
Move to Condition Register from XER	**mcrxr**	**cr**D
Move from Condition Register	**mfcr**	**r**D

Implementation Note—The Book E architecture states that the Move to Condition Register Fields (**mtcrf)** instruction can perform more slowly when only a portion of the fields are updated as opposed to all the fields. This is not the case for the e500.

3.3.1.5.2 Move to/from Special-Purpose Register Instructions

Table 3-22 lists the **mtspr** and **mfspr** instructions.

Table 3-22. Move to/from Special-Purpose Register Instructions

Name	Mnemonic	Syntax
Move to Special-Purpose Register	**mtspr**	SPR,**r**S
Move from Special-Purpose Register	**mfspr**	**r**D,SPR

Table 3-23 summarizes all SPRs defined in Book E, indicating which are user-level access. The SPR number column lists register numbers used in the instruction mnemonics.

Table 3-23. Book E Special-Purpose Registers (by SPR Abbreviation)

SPR Abbreviation	Name	Defined SPR Number		Access	Supervisor Only	Section/ Page
		Decimal	Binary			
CSRR0	Critical save/restore register 0	58	00001 11010	Read/Write	Yes	2.7.1.2/2-18
CSRR1	Critical save/restore register 1	59	00001 11011	Read/Write	Yes	2.7.1.2/2-18
CTR	Count register	9	00000 01001	Read/Write	No	2.4.3/2-10
DAC1	Data address compare 1	316	01001 11100	Read/Write	Yes	2.13.4/2-48
DAC2	Data address compare 2	317	01001 11101	Read/Write	Yes	2.13.4/2-48
DBCR0	Debug control register 0	308	01001 10100	Read/Write	Yes	2.13.1/2-46
DBCR1	Debug control register 1	309	01001 10101	Read/Write	Yes	2.13.1/2-46
DBCR2	Debug control register 2	310	01001 10110	Read/Write	Yes	2.13.1/2-46
DBSR	Debug status register	304	01001 10000	Read/Clear[1]	Yes	2.13.2/2-47
DEAR	Data exception address register	61	00001 11101	Read/Write	Yes	2.7.1.3/2-18
DEC	Decrementer	22	00000 10110	Read/Write	Yes	2.6.4/2-16
DECAR	Decrementer auto-reload	54	00001 10110	Write-only	Yes	2.6.4/2-16
ESR	Exception syndrome register	62	00001 11110	Read/Write	Yes	2.7.1.6/2-20
IAC1	Instruction address compare 1	312	01001 11000	Read/Write	Yes	2.13.3/2-48
IAC2	Instruction address compare 2	313	01001 11001	Read/Write	Yes	2.13.3/2-48
IVOR0	Critical input	400	01100 10000	Read/Write	Yes	2.7.1.5/2-19
IVOR1	Critical input interrupt offset	401	01100 10001	Read/Write	Yes	2.7.1.5/2-19
IVOR2	Data storage interrupt offset	402	01100 10010	Read/Write	Yes	2.7.1.5/2-19
IVOR3	Instruction storage interrupt offset	403	01100 10011	Read/Write	Yes	2.7.1.5/2-19
IVOR4	External input interrupt offset	404	01100 10100	Read/Write	Yes	2.7.1.5/2-19
IVOR5	Alignment interrupt offset	405	01100 10101	Read/Write	Yes	2.7.1.5/2-19
IVOR6	Program interrupt offset	406	01100 10110	Read/Write	Yes	2.7.1.5/2-19
IVOR8	System call interrupt offset	408	01100 11000	Read/Write	Yes	2.7.1.5/2-19
IVOR10	Decrementer interrupt offset	410	01100 11010	Read/Write	Yes	2.7.1.5/2-19
IVOR11	Fixed-interval timer interrupt offset	411	01100 11011	Read/Write	Yes	2.7.1.5/2-19
IVOR12	Watchdog timer interrupt offset	412	01100 11100	Read/Write	Yes	2.7.1.5/2-19
IVOR13	Data TLB error interrupt offset	413	01100 11101	Read/Write	Yes	2.7.1.5/2-19
IVOR14	Instruction TLB error interrupt offset	414	01100 11110	Read/Write	Yes	2.7.1.5/2-19
IVOR15	Debug interrupt offset	415	01100 11111	Read/Write	Yes	2.7.1.5/2-19
IVPR	Interrupt vector	63	00001 11111	Read/Write	Yes	2.7.1.4/2-19
LR	Link register	8	00000 01000	Read/Write	No	2.4.2/2-10
PID	Process ID register [2]	48	00001 10000	Read/Write	Yes	2.12.1/2-36
PIR	Processor ID register	286	01000 11110	Read only	Yes	2.5.2/2-12
PVR	Processor version register	287	01000 11111	Read only	Yes	2.5.3/2-13

Table 3-23. Book E Special-Purpose Registers (by SPR Abbreviation) (continued)

SPR Abbreviation	Name	Defined SPR Number		Access	Supervisor Only	Section/ Page
		Decimal	Binary			
SPRG0	SPR general 0	272	01000 10000	Read/Write	Yes	2.8/2-24
SPRG1	SPR general 1	273	01000 10001	Read/Write	Yes	2.8/2-24
SPRG2	SPR general 2	274	01000 10010	Read/Write	Yes	2.8/2-24
SPRG3	SPR general 3	259	01000 00011	Read only	No[3]	2.8/2-24
		275	01000 10011	Read/Write	Yes	2.8/2-24
SPRG4	SPR general 4	260	01000 00100	Read only	No	2.8/2-24
		276	01000 10100	Read/Write	Yes	2.8/2-24
SPRG5	SPR general 5	261	01000 00101	Read only	No	2.8/2-24
		277	01000 10101	Read/Write	Yes	2.8/2-24
SPRG6	SPR general 6	262	01000 00110	Read only	No	2.8/2-24
		278	01000 10110	Read/Write	Yes	2.8/2-24
SPRG7	SPR general 7	263	01000 00111	Read only	No	2.8/2-24
		279	01000 10111	Read/Write	Yes	2.8/2-24
SRR0	Save/restore register 0	26	00000 11010	Read/Write	Yes	2.7.1.1/2-18
SRR1	Save/restore register 1	27	00000 11011	Read/Write	Yes	2.7.1.1/2-18
TBL	Time base lower	268	01000 01100	Read only	No	2.6.3/2-16
		284	01000 11100	Write-only	Yes	2.6.3/2-16
TBU	Time base upper	269	01000 01101	Read only	No	2.6.3/2-16
		285	01000 11101	Write-only	Yes	2.6.3/2-16
TCR	Timer control register	340	01010 10100	Read/Write	Yes	2.6.1/2-15
TSR	Timer status register	336	01010 10000	Read/Clear[4]	Yes	2.6.2/2-16
USPRG0	User SPR general 0[5]	256	01000 00000	Read/Write	No	2.8/2-24
XER	Integer exception register	1	00000 00001	Read/Write	No	2.3.2/2-9

1 The DBSR is read using **mfspr**. It cannot be directly written to. Instead, DBSR bits corresponding to 1 bits in the GPR can be cleared using **mtspr**.

2 Implementations may support more than one PID. The e500 implements the Book E–defined PID as PID0.

3 User-mode read access to SPRG3 is implementation-dependent.

4 The TSR is read using mfspr. It cannot be directly written to. Instead, TSR bits corresponding to 1 bits in the GPR can be cleared using mtspr.

5 USPRG0 is a separate physical register from SPRG0.

Table 3-24 lists e500-specific SPRs, indicating which can be accessed by user-level software. Compilers should recognize SPR names when parsing instructions.

Table 3-24. Implementation-Specific SPRs (by SPR Abbreviation)

SPR Abbreviation	Name	SPR Number	Access	Supervisor Only	Section/ Page
BBEAR	Branch buffer entry address register	513	Read/Write	No	2.9.1/2-25
BBTAR	Branch buffer target address register	514	Read/Write	No	2.9.2/2-25
BUCSR	Branch unit control and status register	1013	Read/Write	Yes	2.9.3/2-26
HID0	Hardware implementation dependent register 0	1008	Read/Write	Yes	2.10.1/2-27
HID1	Hardware implementation dependent register 1	1009	Read/Write	Yes	2.10.1/2-27
IVOR32	SPE APU unavailable interrupt offset	528	Read/Write	Yes	2.7.1.5/2-19
IVOR33	Embedded floating-point data exception interrupt offset	529	Read/Write	Yes	2.7.1.5/2-19
IVOR34	Embedded floating-point round exception interrupt offset	530	Read/Write	Yes	2.7.1.5/2-19
IVOR35	Performance monitor	531	Read/Write	Yes	2.7.1.5/2-19
L1CFG0	L1 cache configuration register 0	515	Read only	No	2.11.3/2-34
L1CFG1	L1 cache configuration register 1	516	Read only	No	2.11.4/2-35
L1CSR0	L1 cache control and status register 0	1010	Read/Write	Yes	2.11.1/2-31
L1CSR1	L1 cache control and status register 1	1011	Read/Write	Yes	2.11.2/2-33
MAS0	MMU assist register 0	624	Read/Write	Yes	2.12.5.1/2-40
MAS1	MMU assist register 1	625	Read/Write	Yes	2.12.5.2/2-41
MAS2	MMU assist register 2	626	Read/Write	Yes	2.12.5.3/2-42
MAS3	MMU assist register 3	627	Read/Write	Yes	2.12.5.4/2-43
MAS4	MMU assist register 4	628	Read/Write	Yes	2.12.5.5/2-43
MAS6	MMU assist register 6	630	Read/Write	Yes	2.12.5.6/2-44
MCAR	Machine check address register	573	Read only	Yes	2.7.2.3/2-22
MCSR	Machine check syndrome register	572	Read/Write	Yes	2.7.2.4/2-23
MCSRR0	Machine-check save/restore register 0	570	Read/Write	Yes	2.7.2.1/2-22
MCSRR1	Machine-check save/restore register 1	571	Read/Write	Yes	2.7.2.2/2-22
MMUCFG	MMU configuration register	1015	Read only	Yes	2.12.3/2-37
MMUCSR0	MMU control and status register 0	1012	Read/Write	Yes	2.12.2/2-36
PID0	Process ID register 0. Book E defines only this PID register and refers to as PID rather than PID0.	48	Read/Write	Yes	2.12.1/2-36
PID1	Process ID register 1	633	Read/Write	Yes	
PID2	Process ID register 2	634	Read/Write	Yes	
SPEFSCR	Signal processing and embedded floating-point status and control register	512	Read/Write	No	2.14.1/2-49
TLB0CFG	TLB configuration register 0	688	Read only	Yes	2.12.4/2-37
TLB1CFG	TLB configuration register 1	689	Read only	Yes	2.12.4.2/2-39

3.3.1.6 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 Section 3.3.1.7, “Atomic Update Primitives Using **lwarx** and **stwcx.**,” for additional information about these instructions and about related aspects of memory synchronization. See Table 3-25 for a summary.

Table 3-25. Memory Synchronization Instructions

Name	Mnemonic	Syntax	Implementation Notes
Instruction Synchronize	**isync**	—	**isync** is refetch serializing; the e500 waits for previous instructions (including any interrupts they generate) to complete before **isync** executes, which purges all instructions from the core and refetches the next instruction. **isync** does not wait for pending stores in the store queue to complete. Any subsequent instruction sees all effects of instructions before the **isync**. Because it prevents execution of subsequent instructions until preceding 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 the branch target is the next sequential instruction to be executed.
Load Word and Reserve Indexed	**lwarx**	**r**D,**r**A,**r**B	**lwarx** with **stwcx.** can emulate semaphore operations such as test and set, compare and swap, exchange memory, and fetch and add. Both instructions must use the same EA. Reservation granularity is implementation-dependent. The e500 makes reservations on behalf of aligned 32-byte sections of address space. Executing **lwarx** and **stwcx.** to a page marked write-through (WIMG = 10*xx*) or when the data cache is locked causes a data storage interrupt. If the location is not word-aligned, an alignment interrupt occurs. See Section 3.3.1.7, “Atomic Update Primitives Using **lwarx** and **stwcx.**”
Memory Barrier	**mbar**	MO	**mbar** provides a memory barrier. (Note that **mbar** uses the same opcode as **eieio**, defined by the Classic PowerPC architecture, and with which **mbar** (MO=1) is identical, as defined by the EIS). The behavior of **mbar** depends on the value of MO operand. 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—The EIS defines **mbar** to function identically to **eieio**, as defined by the classic PowerPC architecture. For more information, see Section 3.3.1.6.1, “mbar (MO = 1).” The following sequence shows one use of **mbar** in supporting shared data, ensuring the action is completed before the lock is released. P1 P2 lock . . . read & write . . . mbar . . . free lock . . . . . . lock . . . read & write . . . mbar . . . free lock

Table 3-25. Memory Synchronization Instructions (continued)

Name	Mnemonic	Syntax	Implementation Notes
Memory Synchronize	**msync**	—	**msync** provides a memory barrier throughout the memory hierarchy. In the e500, **msync** waits for proceeding data memory accesses to become visible to the entire memory hierarchy; then it is broadcast on the bus. **msync** completes only after its address tenure is performed without being ARTRYed. Subsequent instructions can execute out of order but complete only after the **msync** completes. **msync** latency depends on the processor state when it is dispatched and on various system-level conditions. Frequent use of **msync** degrades performance. System designs with an external cache should take care to recognize the hardware signaling caused by an MSYNC bus operation and perform the appropriate actions to guarantee that memory references that can be queued internally to the external cache have been performed globally. Note the following: • **msync** is used to ensure that all stores into a data structure caused by store instructions executed in a critical section of a program are performed with respect to another processor before the store that releases the lock is performed with respect to that processor. **mbar** is preferable in many cases. • The Freescale EIS further requires that, unlike a context-synchronizing operation, **msync** does not discard prefetched instructions. The e500 broadcasts **mbar** only if ABE = 1 to allow management of external L2 caches and other L1 caches in the system. Section 3.5.1, "Lock Acquisition and Import Barriers," describes how the **msync** and **mbar** instructions can be used to control memory access ordering when memory is shared between programs.
Store Word Conditional Indexed	**stwcx.**	rS,**r**A,**r**B	**lwarx** with **stwcx.** can emulate semaphore operations such as test and set, compare and swap, exchange memory, and fetch and add. Both instructions must use the same EA. Reservation granularity is implementation-dependent. The e500 makes reservations on behalf of aligned 32-byte sections of address space. Executing **lwarx** and **stwcx.** to a page marked write-through (WIMG = 10*xx*) or when the data cache is locked causes a data storage interrupt. If the location is not word-aligned, an alignment interrupt occurs. See Section 3.3.1.7, "Atomic Update Primitives Using **lwarx** and **stwcx.**."

3.3.1.6.1 mbar (MO = 1)

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

- Caching-inhibited, guarded loads and stores to memory and write-through-required stores to memory. **mbar** (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** 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.

Except for **dcbz** and **dcba**, **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 that are 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 memory access appears to be on the same side of the barrier to both the core and the I/O device.

Because the core performs 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.

Note that **mbar** (MO = 1) does not connect hardware considerations to it such as multiprocessor implementations that send an **mbar** (MO = 1) address-only broadcast (useful in some designs). For example, if a design has an external buffer that re-orders loads and stores for better bus efficiency, **mbar** (MO = 1) broadcasts signals to that buffer that previous loads/stores (marked caching-inhibited, guarded, or write-through required) must complete before any following loads/stores (marked caching-inhibited, guarded, or write-through required).

Section 3.5.1, "Lock Acquisition and Import Barriers," describes how the **msync** and **mbar** instructions can be used to control memory access ordering when memory is shared between programs.

3.3.1.7 Atomic Update Primitives Using lwarx and stwcx.

The **lwarx** and **stwcx.** instructions together permit atomic update of a memory location. Book E provides word and double-word forms of each of these instructions. Described here is the operation of **lwarx** and **stwcx.**.

A specified memory location that may be modified by other processors or mechanisms requires memory coherency. If the location is in write-through-required or caching-inhibited memory, the implementation determines whether these instructions function correctly or cause the system data storage error handler to be invoked. The e500 takes a data storage interrupt if the location is write-through but does not take the interrupt if the location is caching inhibited.

Note the following:

- The memory coherency required attribute on other processors and mechanisms ensures that their stores to the specified location cause the reservation created by the **lwarx** to be cancelled.
- **Warning:** Support for load and reserve and store conditional instructions for which the specified location is in caching-inhibited memory is being phased out of Book E. It is likely not to be provided on future implementations. New programs should not use these instructions to access caching inhibited memory.

A **lwarx** instruction is a load from a word-aligned location with the following side effects.

- A reservation for a subsequent **stwcx.** instruction is created.
- The memory coherency mechanism is notified that a reservation exists for the location accessed by the **lwarx**.

The **stwcx.** is a store to a word-aligned location that is conditioned on the existence of the reservation created by the **lwarx** and on whether both instructions specify the same location. To emulate an atomic operation, both **lwarx** and **stwcx.** must access the same location. **lwarx** and **stwcx.** are ordered by a dependence on the reservation, and the program is not required to insert other instructions to maintain the order of memory accesses caused by these two instructions.

A **stwcx.** performs a store to the target location only if the location accessed by the **lwarx** that established the reservation has not been stored into by another processor or mechanism between supplying a value for the **lwarx** and storing the value supplied by the **stwcx.**. If the instructions specify different locations, the store is not necessarily performed. CR0 is modified to indicate whether the store was performed, as follows:

CR0[LT,GT,EQ,SO] = 0b00 || store_performed || XER[SO]

If a **stwcx.** completes but does not perform the store because a reservation no longer exists, CR0 is modified to indicate that the **stwcx.** completed without altering memory.

A **stwcx.** that performs its store is said to succeed.

A successful **stwcx.** to a given location may complete before its store has been performed with respect to other processors and mechanisms. As a result, a subsequent load or **lwarx** from the given location on another processor may return a stale value. However, a subsequent **lwarx** from the given location on the other processor followed by a successful **stwcx.** on that processor is guaranteed to have returned the value stored by the first processor's **stwcx.** (in the absence of other stores to the given location).

3.3.1.7.1 Reservations

The ability to emulate an atomic operation using **lwarx** and **stwcx.** is based on the conditional behavior of **stwcx.**, the reservation set by **lwarx**, and the clearing of that reservation if the target location is modified by another processor or mechanism before the **stwcx.** performs its store.

A reservation is held on an aligned unit of real memory called a reservation granule. The size of the reservation granule is implementation-dependent, but is a multiple of 4 bytes for **lwarx**. The reservation granule associated with effective address EA contains the real address to which EA maps. ('real_addr(EA)' in the RTL for the load and reserve and store conditional instructions stands for 'real address to which EA maps.') When one processor holds a reservation and another processor performs a store, the first processor's reservation is cleared if the store affects any bytes in the reservation granule.

NOTE

One use of **lwarx** and **stwcx.** is to emulate a compare and swap primitive like that provided by the IBM System/370 compare and swap instruction, which checks only that the old and current values of the word being tested are equal, with the result that programs that use such a compare and swap to control a shared resource can err if the word has been modified and the old value is subsequently restored.

The use of **lwarx** and **stwcx.** improves on such a compare and swap because the reservation reliably binds **lwarx** and **stwcx.** together. The reservation is always lost if the word is modified by another processor or mechanism between the **lwarx** and **stwcx.**, so the **stwcx.** never succeeds unless the word has not been stored into (by another processor or mechanism) since the **lwarx**.

A processor has at most one reservation at any time. Book E states that a reservation is established by executing a **lwarx** and is lost (or may be lost, in the case of the fourth and fifth bullets) if any of the following occurs.

- The processor holding the reservation executes another **lwarx**; this clears the first reservation and establishes a new one.
- The processor holding the reservation executes any **stwcx.**, regardless of whether the specified address matches that of the **lwarx**.
- Another processor executes a store or **dcbz** to the same reservation granule.
- Another processor executes a **dcbtst**, **dcbst**, or **dcbf** to the same reservation granule; whether the reservation is lost is undefined.

- Another processor executes a **dcba** to the reservation granule. The reservation is lost if the instruction causes the target block to be newly established in the data cache or to be modified; otherwise, whether the reservation is lost is undefined.
- Some other mechanism modifies a location in the same reservation granule.

Interrupts are not guaranteed to clear reservations. (However, system software invoked by interrupts may clear reservations.)

In general, programming conventions must ensure that **lwarx** and **stwcx.** specify addresses that match; a **stwcx.** should be paired with a specific **lwarx** to the same location. Situations in which a **stwcx.** may erroneously be issued after some **lwarx** other than that with which it is intended to be paired must be scrupulously avoided. For example, there must not be a context switch in which the processor holds a reservation on behalf of the old context, and the new context resumes after a **lwarx** and before the paired **stwcx.**. The **stwcx.** in the new context might succeed, which is not what was intended by the programmer.

Such a situation must be prevented by issuing a **stwcx.** to a dummy writable word-aligned location as part of the context switch, thereby clearing any reservation established by the old context. Executing **stwcx.** to a word-aligned location is enough to clear the reservation.

In the e500, a reservation is lost for any of the following reasons:

- Execution of a **stwcx.**
- Any of the following interrupts occur:
  — External
  — Performance monitor
  — Critical input interrupt
  — Machine check
  — Fixed-interval timer
  — Decrementer
  — Unconditional debug event
  — Watchdog timer
- Snoops
  — RWITM, RCLAIM
  — Writes, flush, kill, dkill
- Another processor executes any of the following to the reservation granule:
  — **dcbtst**
  — **dcbf**
  — dcba

— **dcbst** (The e500 broadcasts **dcbst** as a flush; if another processor implements **dcbst** as a clean, the reservation is not cleared.)

3.3.1.7.2 Forward Progress

Forward progress in loops that use **lwarx** and **stwcx.** is achieved by a cooperative effort among hardware, operating system software, and application software.

Book E guarantees one of the following when a processor executes a **lwarx** to obtain a reservation for location X and then a **stwcx.** to store a value to location X:

1. The **stwcx.** succeeds and the value is written to location X.
2. The **stwcx.** fails because some other processor or mechanism modified location X.
3. The **stwcx.** fails because the processor’s reservation was lost for some other reason.

In cases 1 and 2, the system as a whole makes progress in the sense that some processor successfully modifies location X. Case 3 covers reservation loss required for correct operation of the rest of the system. This includes cancellation caused by some other processor writing elsewhere in the reservation granule for X, as well as cancellation caused by the operating system in managing certain limited resources such as real memory or context switches. It may also include implementation-dependent causes of reservation loss.

An implementation may make a forward progress guarantee, defining the conditions under which the system as a whole makes progress. Such a guarantee must specify the possible causes of reservation loss in case 3. Although Book E alone cannot provide such a guarantee, the conditions in cases 1 and 2 are necessary for a guarantee. An implementation and operating system can build on them to provide such a guarantee.

Note that Book E does not guarantee fairness. In competing for a reservation, two processors can indefinitely lock out a third.

3.3.1.7.3 Reservation Loss Due to Granularity

Lock words should be allocated such that contention for the locks and updates to nearby data structures do not cause excessive reservation losses due to false indications of sharing that can occur due to the reservation granularity.

A processor holding a reservation on any word in a reservation granule loses its reservation if some other processor stores anywhere in that granule. Such problems can be avoided only by ensuring that few such stores occur. This can most easily be accomplished by allocating an entire granule for a lock and wasting all but one word.

Reservation granularity may vary for each implementation. There are no architectural restrictions bounding the granularity implementations must support, so reasonably portable code must dynamically allocate aligned and padded memory for locks to guarantee absence of granularity-induced reservation loss.

3.3.1.8 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 3.3.2.2, "Supervisor-Level Memory Control Instructions," for information about supervisor-level cache and translation lookaside buffer management instructions.

This section does not describe the cache-locking APU instructions, which are described in Section 3.8.4, "Cache Locking APU."

3.3.1.8.1 User-Level Cache Instructions

The instructions listed in Table 3-26 help user-level programs manage on-chip caches if they are implemented. See Chapter 11, "L1 Caches," for more information about cache topics. The following sections describe how these operations are treated with respect to the e500's caches. The e500 supports the following CT values, defined by the EIS:

- CT = 0 indicates the L1 cache.
- CT = 1 indicates the L2 cache.

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, an **msync** must be placed after those instructions.

Note that the e500 interprets cache control instructions (**icbi**, **dcbi**, **dcbf**, **dcbz**, and **dcbst**) as if they pertain only to local caches. On some implementations, HID1[ABE] must be set to allow management of external L2 caches as well as other L1 caches in the system.

Section 3.8.4, "Cache Locking APU," describes cache-locking APU instructions.

Table 3-26. User-Level Cache Instructions

Name	Mnemonic	Syntax	Implementation Notes
Data Cache Block Allocate	**dcba**	**r**A,**r**B	The EA is computed, translated, and checked for protection violations. For cache hits, 32 bytes of zeros are written to the cache block and the tag is marked modified. For cache misses with the replacement block marked non-dirty, a zero reload is performed and the block is marked modified. However, if the replacement block is marked modified, the contents are written back to memory first. If WIMG = xx1x (coherency enforced), the address is broadcast to the bus before the zero reload fill. A no-op occurs if the cache is disabled or locked, if the page is marked write-through or cache-inhibited, or if a TLB protection violation occurs.
Data Cache Block Flush[1]	**dcbf**	**r**A,**r**B	The EA is computed, translated, and checked for protection violations: • For cache hits with the tag marked modified, the cache block is written back to memory and the cache entry is invalidated. • For cache hits with the tag marked not modified, the entry is invalidated. • For cache misses, no further action is taken. A **dcbf** is broadcast if WIMG = xx1x (coherency enforced).**dcbf** acts like a load with respect to address translation and memory protection. It executes in the LSU regardless of whether the cache is disabled or locked.
Data Cache Block Set to Zero [1]	**dcbz**	**r**A,**r**B	The EA is computed, translated, and checked for protection violations. For cache hits, 32 bytes of zeros are written to the cache block and the tag is marked modified. For cache misses with the replacement block marked not modified, the zero reload is performed and the cache block is marked modified. However, if the replacement block is marked modified, the contents are written back to memory first. **dcbz** takes an alignment interrupt if the cache is locked or disabled or if the cache is marked WT or CI. If WIMG = xx1x (coherency enforced), the address is broadcast to the bus before the zero reload fill. The interrupt priorities (from highest to lowest) are as follows: 1 Cache Is locked—alignment interrupt 2 Page marked write-through or cache-inhibited—alignment interrupt 3 TLB protection violation—data storage interrupt **dcbz** is broadcast if WIMG = xx1x (coherency enforced).
Data Cache Block Store [1]	**dcbst**	**r**A,**r**B	The EA is computed, translated, and checked for protection violations. • For cache hits with the tag marked not modified, no further action is taken. • For cache hits with the tag marked modified, the cache block is written back to memory and marked exclusive. If WIMG = xx1x (coherency enforced) **dcbst** is broadcast. **dcbst** acts like a load with respect to address translation and memory protection. It executes regardless of whether the cache is disabled or locked.
Data Cache Block Touch [2]	**dcbt**	CT,**r**A,**r**B	**dcbt** allows potential performance enhancements through software-initiated prefetch hints. Implementations are not required to take action based on execution of **dcbt** but can prefetch the cache block corresponding to the EA into their cache. When **dcbt** executes, the e500 checks for protection violations (as for a load instruction). **dcbt** is treated as a no-op in the following cases: • The access causes a protection violation.The page is mapped cache-inhibited. • All lines that this entry maps to are locked or the cache is disabled. • HID0[NOPTI] = 1 Otherwise, if no data is in the cache location, the e500 requests a cache line fill. Data brought into the cache is validated as if it were a load instruction. The memory reference of a **dcbt** sets the reference bit.

Table 3-26. User-Level Cache Instructions (continued)

Name	Mnemonic	Syntax	Implementation Notes
Data Cache Block Touch for Store [1, 2]	**dcbtst**	CT,**r**A,**r**B	**dcbtst** can be no-oped by setting HID0[NOPTI]. **dcbtst** behaves similarly to **dcbt**, except that the line-fill request on the bus is signaled as read or read-claim, and the data is marked as exclusive in the L1 data cache if there is no shared response on the bus. More specifically, the following cases occur depending on where the block currently exists or does not exist in the e500. • **dcbtst** hits in the L1 data cache. In this case, the **dcbtst** does nothing and the state of the block in the cache is not changed. Thus, if the block was in the shared state, a subsequent store hits on this shared block and incur the associated latency penalties. • **dcbtst** misses in the L1 data cache and hits in the L2 cache. In this case, **dcbtst** reloads the L1 data cache with the state found in the L2 cache. Again, if the block was in the shared state in the L2, a subsequent store hits on this shared block and incur the associated latency penalties. • **dcbtst** misses in L1 data cache, L2 caches. In this case, e500 requests the block from memory with read or read-claim and reload the L1 data cache in the exclusive state. As subsequent store hits on exclusive and can perform the store to the L1 data cache immediately. **dcbtst** is no-oped if its target address is mapped as write-through.
Instruction Cache Block Invalidate [1]	**icbi**	**r**A,**r**B	**icbi** is broadcast on the bus. It should always be followed by an **msync** and an **isync** to make sure its effects are seen by instruction fetches following the **icbi** itself.
Instruction Cache Block Touch	**icbt**	CT,**r**A,**r**B	If CT = 0, the e500 treats **icbt** as a no-op. If CT = 1, **icbt** executes as follows: • For L1 data cache hit-to-modified, **icbt** performs like a load on the bus; e500 ignores data (for L2). • For L1 data cache hit-to-modified—cast out (for L2) • If NOPTI is 0, **icbt** does a touch load to the L2 cache.

[1] On some implementations, such as the e500, HID1[ABE] must be set to allow management of external L2 caches (for implementations with L2 caches) as well as other L1 caches in the system.

[2] A program that uses **dcbt** and **dcbtst** improperly is less efficient. To improve performance, HID0[NOPTI] can be set, which causes **dcbt** and **dcbtst** to be no-oped at the cache. They do not cause bus activity and cause only a 1-clock execution latency. The default state of this bit is zero, which enables the use of these instructions.

3.3.2 Supervisor-Level Instructions

The Book E architecture includes the structure of the memory management model, supervisor-level registers, and the interrupt model. This section describes the supervisor-level instructions implemented by e500.

3.3.2.1 System Linkage Instructions

This section describes the system linkage instructions (see Table 3-27). The user-level **sc** instruction lets a user program call on the system to perform a service and causes the processor to take a system call interrupt. The supervisor-level **rfi** instruction is used for returning from an interrupt handler. The **rfci** instruction is used for critical interrupts; **rfmci** is used for machine check interrupts.

Table 3-27. System Linkage Instructions—Supervisor-Level

Name	Mnemonic	Syntax	Implementation Notes
Return from Interrupt	**rfi**	—	**rfi** is context-synchronizing, which for the e500 means it works its way to the final execute stage, updates architected registers, and redirects the instruction flow.
Return from Machine Check Interrupt	**rfmci**	—	(e500-specific) When **rfmci** is executed, the values in the machine check interrupt save and restore registers (MCSRR0 and MCSRR1) are restored. **rfmci** is context-synchronizing; it works its way to the final execute stage, updates architected registers, and redirects instruction flow.
Return from Critical Interrupt	**rfci**	—	When **rfci** executes, the values in the critical interrupt save and restore registers (CSRR0 and CSRR1) are restored. **rfci** is context-synchronizing, which for the e500 means it works its way to the final execute stage, updates architected registers, and redirects the instruction flow.
System Call	**sc**	—	The **sc** instruction is context-synchronizing.

Table 3-28 lists instructions for accessing the MSR.

Table 3-28. Move to/from Machine State Register Instructions

Name	Mnemonic	Syntax	Description
Move from Machine State Register	**mfmsr**	**r**D	—
Move to Machine State Register	**mtmsr**	**r**S	—
Write MSR External Enable	**wrtee**	**r**S	Bit 48 of the contents of **r**S is placed into MSR[EE]. No other MSR bits are affected.
Write MSR External Enable Immediate	**wrteei**	E	The value specified in the E field is placed into MSR[EE]. No other MSR bits are affected.

Certain encodings of the SPR field of **mtspr** and **mfspr** instructions (shown in Table 3-22) provide access to supervisor-level SPRs. Table 3-23 lists encodings for architecture-defined SPRs. Encodings for e500-specific, supervisor-level SPRs are listed in Table 3-24. Simplified mnemonics are provided for **mtspr** and **mfspr**. See the EREF for more information on context synchronization requirements when altering certain SPRs.

3.3.2.2 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 3.3.1.8, "Memory Control Instructions," describes user-level memory control instructions.

3.3.2.2.1 Supervisor-Level Cache Instruction

Table 3-29 lists the only supervisor-level cache management instruction.

Table 3-29. Supervisor-Level Cache Management Instruction

Name	Mnemonic	Syntax	Implementation Notes
Data Cache Block Invalidate	**dcbi**	rA,rB	**dcbi** executes as described in Book E. The e500 core invalidates the cache block without pushing it out to memory. See Section 3.3.1.8.1, “User-Level Cache Instructions.” In the e500, **dcbi** cannot generate a cache-locking exception**.** The e500 broadcasts **dcbi** only if HID1[ABE] is set. ABE must be set to allow management of external L2 caches (for implementations with L2 caches) and other L1 caches in the system.

See Section 3.3.1.8.1, “User-Level Cache Instructions,” for cache instructions that provide user-level programs the ability to manage the on-chip caches.

3.3.2.2.2 Supervisor-Level TLB Management Instructions

The address translation mechanism is defined in terms of TLBs and page table entries (PTEs) Book E processors use to locate the logical-to-physical address mapping for a particular access. See Chapter 12, “Memory Management Units,” for more information about TLB operations. Table 3-30 summarizes the operation of the TLB instructions in the e500.

Table 3-30. TLB Management Instructions

Name	Mnemonic	Syntax	Implementation Notes
TLB Invalidate Virtual Address Indexed	**tlbivax**	rA, rB	A TLB invalidate operation is performed whenever **tlbivax** is executed. **tlbivax** invalidates any TLB entry 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 12.3, “Translation Lookaside Buffers (TLBs).” On some implementations, HID1[ABE] must be set to allow management of external L2 caches (for implementations with L2 caches) as well as other L1 caches in the system.
TLB Read Entry	**tlbre**	—	**tlbre** causes the contents of a single TLB entry to be extracted from the MMU and be placed in the corresponding fields of the MMU assist (MAS) registers. The entry extracted is specified by the TLBSEL, ESEL, and EPN fields of MAS0 and MAS2. The contents extracted from the MMU are placed in MAS0–MAS3. Note that for the e500v2, if HID0[EN_MAS7_UPDATE] = 1, MAS7 is also updated with the four highest-order bits of physical address for the TLB entry. See Section 12.3, “Translation Lookaside Buffers (TLBs).” The RTL for the Freescale implementation of **tlbre** is as follows: `tlb_entry_id = MAS0(TLBSEL, ESEL | MAS2(EPN)` `result = MMU(tlb_entry_id)` `MAS0, MAS1, MAS2, MAS3, (and MAS7 if HID0[EN_MAS7_UPDATE] = 1) = result`

Table 3-30. TLB Management Instructions (continued)

Name	Mnemonic	Syntax	Implementation Notes
TLB Search Indexed	**tlbsx**	**r**A, **r**B	**tlbsx** updates the MAS registers conditionally based on the success or failure of a lookup in the MMU. The lookup is controlled by the EA provided by GPR[**r**B] specified in the instruction encoding and MAS6[SAS,SPID]. The values placed into MAS registers differ, depending on whether a successful or unsuccessful search occurred. See Section 12.3, "Translation Lookaside Buffers (TLBs)." The RTL for the e500 implementation of **tlbsx** is as follows: `if RA!=0 then generate exception` `EA =` $^{32}0$ `|| GPR(RB)`$_{32:63}$ `ProcessID = MAS6(SPID), 0b0000_0000` `AS = MAS6(SAS)` `VA = AS || ProcessID || EA` `if Valid_TLB_matching_entry_exists (VA)` `then result = see Table 12-15, column "tlbsx hit"` `else result = see Table 12-15, column "tlbsx miss"` `MAS0, MAS1, MAS2, MAS3, and MAS7 = result` Note that RA=0 is a preferred form for **tlbsx** and that some Freescale implementations, such as the e500, take an illegal instruction exception program interrupt if RA != 0.
TLB Synchronize	**tlbsync**	—	Causes a TLBSYNC transaction on the e500 core complex bus. This transaction is retried if any processor, including the one that executed the **tlbsync**, has pending memory accesses issued before any previous **tlbivax** completed.See Section 12.3, "Translation Lookaside Buffers (TLBs)." The e500 broadcasts cache **tlbsync** only if ABE = 1 to allow management of external L2 caches (for implementations with L2 caches) as well as other L1 caches in the system
TLB Write Entry	**tlbwe**	—	**tlbwe** causes the contents of certain fields of MAS0, MAS1, MAS2, and MAS3 (and MAS7 on e500v2) to be written into a single TLB entry in the MMU. The entry written is specified by the TLBSEL, ESEL, and EPN fields of MAS0 and MAS2. Execution of **tlbwe** on the e500v2 core also causes the upper 4 bits of the RPN that reside in MAS7 to be written to the selected TLB entry. See Section 12.3, "Translation Lookaside Buffers (TLBs)." The RTL for the e500 implementation of **tlbwe** is as follows: `tlb_entry_id = MAS0(TLBSEL, ESEL)|| MAS2(EPN)` `MMU(tlb_entry_id) = MAS0, MAS1, MAS2, MAS3, (and MAS7 on e500v2)`

Implementation Note—The presence and exact semantics of the TLB management instructions are implementation dependent. To minimize compatibility problems, system software should incorporate uses of these instructions into subroutines.

3.3.3 Recommended Simplified Mnemonics

The description of each instruction includes the mnemonic and a formatted list of operands. Book E–compliant assemblers support the mnemonics and operand listed. To simplify assembly language programming, a set of simplified mnemonics and symbols is provided for some of the most frequently used instructions; refer to Appendix C, "Simplified Mnemonics for PowerPC Instructions," for a complete list. Programs written to be portable across the various assemblers for the Book E architecture should not assume the existence of mnemonics not described in this document.

3.3.4 Book E Instructions with Implementation-Specific Features

Book E defines several instructions in a general way, leaving the details of the execution up to the implementation. These are listed in Table 3-31. This section describes how the e500 core complex implements those instructions. The implementation-specific TLB instructions (listed below) are described in more detail in Section 12.4, "TLB Instructions—Implementation."

Table 3-31. Implementation-Specific Instructions Summary

Name	Mnemonic	Syntax	Category
TLB Invalidate Virtual Address Indexed	**tlbivax**	rA, rB	These are described generally in Section 3.3.2.2.2, "Supervisor-Level TLB Management Instructions." They are described in greater detail in Section 12.4, "TLB Instructions—Implementation."
TLB Read Entry	**tlbre**	—	
TLB Search Indexed	**tlbsx**	rA, rB	
TLB Write Entry	**tlbwe**	—	

A list of user-level instructions defined by both the classic PowerPC architecture and Book E can be found in Section 3.10, "Instruction Listing."

3.3.5 e500 Instructions

The e500 core complex implements the new instructions listed in Table 3-32 (with cross references to more detailed descriptions) that extend the Book E instruction set in accordance with Book E. SPE and embedded floating-point APU instructions are listed in Table 3-36 and Table 3-37.

Table 3-32. e500-Specific Instructions (Except SPE and SPFP Instructions)

Name	Mnemonic	Syntax	Section #/Page
Branch Buffer Load Entry and Lock Set	**bblels**	—	3.9.1/3-63
Branch Buffer Entry Lock Reset	**bbelr**	—	
Data Cache Block Lock Clear	**dcblc**	CT, rA, rB	3.8.4/3-61
Data Cache Block Touch and Lock Set	**dcbtls**	CT, rA, rB	
Data Cache Block Touch for Store and Lock Set	**dcbtstls**	CT, rA, rB	
Instruction Cache Block Lock Clear	**icblc**	CT, rA, rB	
Instruction Cache Block Touch and Lock Set	**icbtls**	CT, rA, rB	
Integer Select	**isel**	rD, rA, rB, crB	3.8.2/3-60
Move from Performance Monitor Register	**mfpmr**	rD,PMRN	3.8.2/3-60
Move to Performance Monitor Register	**mtpmr**	PMRN,rS	
Return from Machine Check Interrupt	**rfmci**	—	3.8.5/3-63

3.3.6 Context Synchronization

Context synchronization is achieved by post- and presynchronizing instructions. An instruction is presynchronized by completing all instructions before dispatching the presynchronized instruction. Post-synchronizing is implemented by not dispatching any later instructions until the post-synchronized instruction is completely finished.

3.4 Memory Access Alignment Support

The e500 core complex provides hardware support for misaligned memory accesses, but at the cost of performance degradation. For loads that hit in the cache, the LSU’s throughput degrades to one misaligned load every 3 cycles. Similarly, stores can be translated at a rate of one misaligned store every 3 cycles. Additionally, after translation, each misaligned store is treated as two distinct entries in the store queue, each requiring a cache access.

A word or half-word memory access requires multiple accesses if it crosses a double-word boundary but not if it crosses a natural boundary. Vector loads and stores cause alignment interrupts if they cross natural alignment boundaries (as shown in Table 3-33).

Frequent use of misaligned memory accesses can greatly degrade performance.

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

Table 3-33. Natural Alignment Boundaries for Extended Vector Instructions

Instruction	Boundary
evld{d,w,h} **evld{d,w,h}x** **evstd{d,w,h}** **evstd{d,w,h}x**	Double word
evlwwsplat{x} **evlwhe{x}** **evlwhou{x}** **evlwhos{x}** **evlwhsplat{x}** **evstwwe{x}** **evstwwo{x}** **evstwhe{x}** **evstwho{x}**	Word
evlhhesplat{x} **evlhhousplat{x}** **evlhhossplat{x}**	Half word

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

3.5 Using msync and mbar to Order Memory Accesses

This section gives examples of how dependencies and the **msync** and **mbar** instructions can be used to control memory access ordering when memory is shared between programs.

3.5.1 Lock Acquisition and Import Barriers

An import barrier is an instruction or sequence of instructions that prevents memory accesses caused by instructions following the barrier from being performed before memory accesses that acquire a lock have been performed. An import barrier can be used to ensure that a shared data structure protected by a lock is not accessed until the lock is acquired. An **msync** can always be used as an import barrier, but the approaches shown in this section generally yield better performance because they order only the relevant memory accesses.

3.5.1.1 Acquire Lock and Import Shared Memory

If **lwarx** and **stwcx.** instructions are used to obtain the lock, an import barrier can be constructed by placing an **isync** instruction immediately following the loop containing the **lwarx** and **stwcx.**. The following example uses the ‘Compare and Swap’ primitive to acquire the lock.

In this example it is assumed that the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, the value to which the lock should be set is in GPR 5, the old value of the lock is returned in GPR 6, and the address of the shared data structure is in GPR 9.

```
loop:  lwarx   r6,0,r3       # load lock and reserve
       cmp     cr0,0,r4,r6   # skip ahead if
       bc      4,2,wait      # lock not free
       stwcx.  r5,0,r3       # try to set lock
       bc      4,2,loop      # loop if lost reservation
       isync                 # import barrier
       lwz     r7,data1(r9)  # load shared data
       .
       .
wait: ...                    # wait for lock to free
```

The second **bc** does not complete until CR0 has been set by the **stwcx.**. The **stwcx.** does not set CR0 until it has completed (successfully or unsuccessfully). The lock is acquired when the **stwcx.** completes successfully. Together, the second **bc** and the subsequent **isync** create an import barrier that prevents the load from data1 from being performed until the branch has been resolved not to be taken.

3.5.1.2 Obtain Pointer and Import Shared Memory

If **lwarx** and **stwcx.** instructions are used to obtain a pointer into a shared data structure, an import barrier is not needed if all the accesses to the shared data structure depend on the value obtained for the pointer.

The following example uses the ‘Fetch and Add’ primitive (see the section entitled ‘Synchronization Primitives’ in Section I) to obtain and increment the pointer.

In this example it is assumed that the address of the pointer is in GPR 3, the value to be added to the pointer is in GPR 4, and the old value of the pointer is returned in GPR 5.

```
loop:  lwarx   r5,0,r3       # load pointer and reserve
       add     r0,r4,r5      # increment the pointer
       stwcx.  r0,0,r3       # try to store new value
       bc      4,2,loop      # loop if lost reservation
       lwz     r7,data1(r5)  # load shared data
```

The load from data1 cannot be performed until the pointer value has been loaded into GPR 5 by the **lwarx**. The load from data1 may be performed out-of-order before the **stwcx.**. But if the **stwcx.** fails, the branch is taken and the value returned by the load from data1 is discarded. If the **stwcx.** succeeds, the value returned by the load from data1 is valid even if the load is performed out-of-order, because the load uses the pointer value returned by the instance of the **lwarx** that created the reservation used by the successful **stwcx.**.

An **isync** could be placed between the **bne-** and the subsequent **lwz**, but no **isync** is needed if all accesses to the shared data structure depend on the value returned by the **lwarx**.

3.5.1.3 Lock Release and Export Barriers

An export barrier is an instruction or sequence of instructions that prevents the store that releases a lock from being performed before stores caused by instructions preceding the barrier have been performed. An export barrier can be used to ensure that all stores to a shared data structure protected by a lock be performed with respect to any other processor (to the extent required by the associated memory coherence required attributes) before the store that releases the lock is performed with respect to that processor.

3.5.1.3.1 Export Shared Memory and Release Lock

An **msync** instruction can always be used as an export barrier, independent of the memory control attributes (for example, presence or absence of the caching inhibited attribute) of the memory containing the lock and the shared data structure. Unless both the lock and the shared data structure are in memory that is neither caching inhibited nor write-through required, an **msync** instruction must be used as the export barrier.

In this example it is assumed that the lock is in memory that is caching inhibited, the shared data structure is in memory that is not caching inhibited, the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, and the address of the shared data structure is in GPR 9.

```
stw      r7,data1(r9)      # store shared data (last)
msync                      # export barrier
stw      r4,lock(r3)       # release lock
```

The **msync** ensures that the store that releases the lock are not performed with respect to any other processor until all stores caused by instructions preceding the **msync** have been performed with respect to that processor.

3.5.1.3.2 Export Shared Memory and Release Lock using mbar (MO = 0)

If both the lock and the shared data structure are in memory that is neither caching inhibited nor write-through required, an **mbar** (MO = 0) instruction can be used as the export barrier. Using **mbar** rather than **msync** yields better performance in most systems.

In this example it is assumed that both the lock and the shared data structure are in memory that is neither caching inhibited nor write-through required, the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, and the address of the shared data structure is in GPR 9.

```
stw      r7,data1(r9)       #store shared data (last)
mbar     0                  #export barrier
stw      r4,lock(r3)        #release lock
```

The **mbar** (MO = 0) ensures that the store that releases the lock is not performed with respect to any other processor until all stores caused by instructions preceding the **mbar** have been performed with respect to that processor.

Recall that, for memory that is neither caching inhibited nor write-through required, **mbar** orders only stores and has no effect on loads. If the portion of the program preceding the **mbar** contains loads from the shared data structure and the stores to the shared data structure do not depend on the values returned by those loads, the store that releases the lock could be performed before those loads. If it is necessary to ensure that those loads are performed before the store that releases the lock, the programmer can either use the **msync** instruction as in Section 3.5.1.3.1, "Export Shared Memory and Release Lock," or use the technique described in Section 3.5.2, "Safe Fetch."

3.5.2 Safe Fetch

If a load must be performed before a subsequent store (for example, the store that releases a lock protecting a shared data structure), a technique similar to the following can be used.

In this example it is assumed that the address of the memory operand to be loaded is in GPR 3, the contents of the memory operand are returned in GPR 4, and the address of the memory operand to be stored is in GPR 5.

```
lwz      r4,0(r3)           #load shared data
cmp      cr0,0,r4,r4        #set CR0 to 'equal'
bc       4,2,$-8            #branch never taken
stw      r7,0(r5)           #store other shared data
```

Alternatively, a technique similar to that described in Section 3.5.1.2, "Obtain Pointer and Import Shared Memory," can be used, by causing the **stw** to depend on the value returned by the **lwz** and omitting the **cmp** and **bc**. The dependency could be created by ANDing the value returned by the **lwz** with zero and then adding the result to the value to be stored by the **stw**.

3.6 Update Instructions

Load-with-update and store-with-update instructions are described in Book E. Internally, the e500 breaks these instructions into two sub-instructions. The update portion of the instruction is executed by one of the simple units, and the load portion is executed by the load/store unit.

Programmers should be aware that the simple unit used is busy for one cycle executing the update portion of the update instruction.

3.7 Memory Synchronization

The **msync** instruction provides a memory barrier throughout the memory hierarchy. It waits for preceding data memory accesses to reach the point of coherency (that is, visible to the entire memory hierarchy); then it is broadcast on the e500 core complex bus. Only after the address bus tenure of the **msync** is successful (that is, without being ARTRYed) is **msync** completed. No subsequent instructions in the stream are initiated until after **msync** completes. Note that **msync** uses the same opcode as the **sync** instruction.

The **msync** instruction is described in Section 3.3.1.6, “Memory Synchronization Instructions.”

The e500 core complex implements two variations of the **mbar** instruction. The desired behavior is selected with the MO field (bits 6–10) of **mbar**, as follows:

- When MO = 0, **mbar** behaves as defined by the Book E architecture.
- When MO = 1, the EIS defines **mbar** to function identically to the Classic PowerPC architecture definition of **eieio**.
- If MO is not 1, the e500 executes **mbar** as though MO = 0.

The e500 core complex implements **lwarx** and **stwcx.** as described in Book E. If the EA is not a multiple of four for either instruction, an alignment interrupt is invoked. If either instruction tries to access a page marked as write-through required, a DSI interrupt is invoked.

As specified in Book E, the e500 core complex requires that, for **stwcx.** to succeed, the EA of **stwcx.** must be to the same reservation granule as the EA of a preceding **lwarx**. Reservation granularity is implementation dependent. The e500 core complex makes reservations on behalf of aligned 32-byte sections of the memory address space.

For the purposes of memory coherency, the reservation granule for **lwarx** and **stwcx.** is also a cache block. A reservation-killing snoop to any address within a cache block that contains the reservation causes the reservation to be invalidated.

The reservation is invalidated when an external interrupt is signaled.

3.8 EIS-Defined Instructions and APUs Implemented on the e500

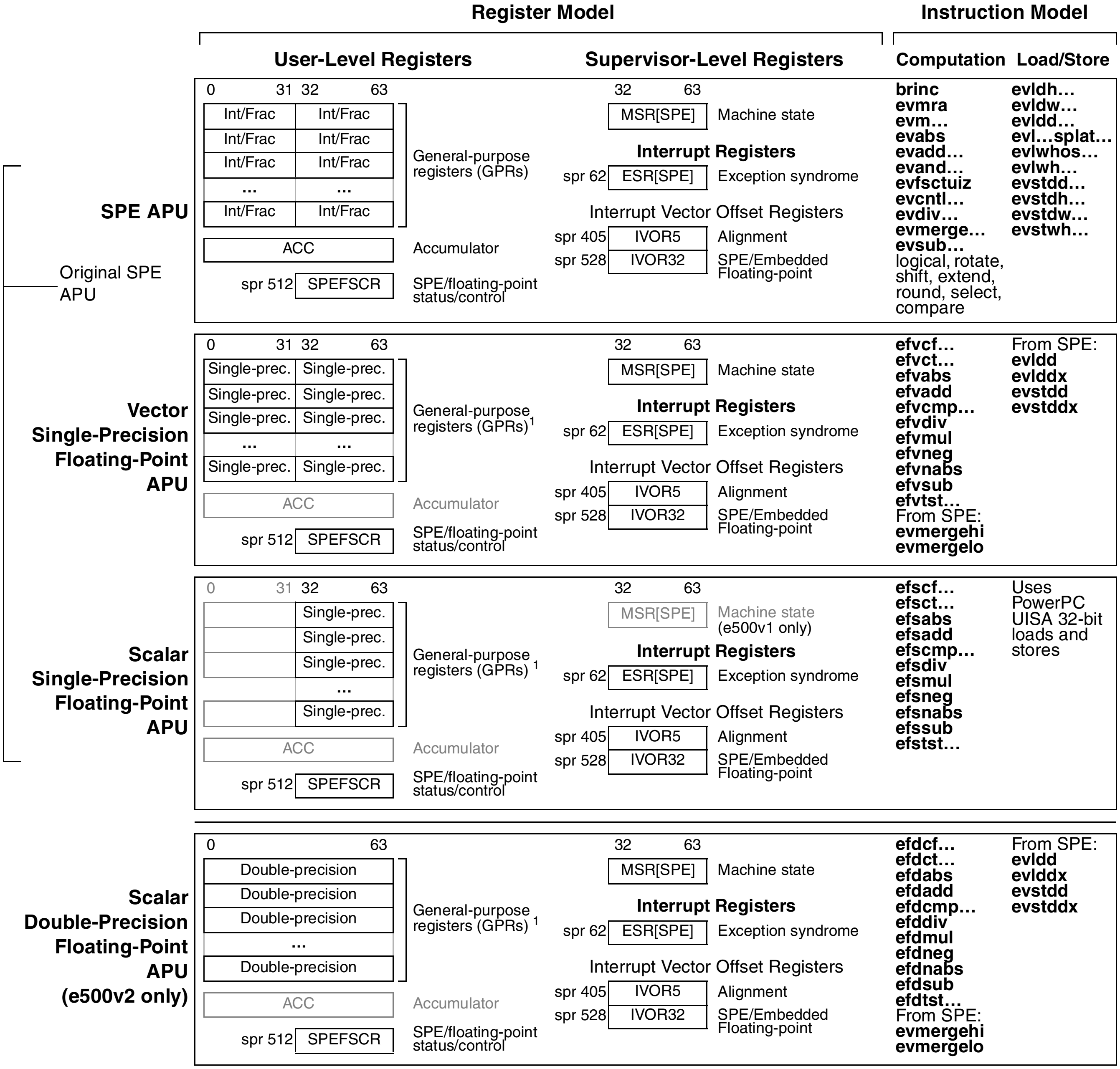
Instructions that are specific to the e500 core are implemented as auxiliary processing units (APUs) and are described in the following sections.

3.8.1 SPE and Embedded Floating-Point APUs

The e500 core complex provides a GPR file with 32, 64-bit registers. The 32-bit Book E instructions operate on the lower (least-significant) 32 bits of the 64-bit register. SPE APU vector instructions and embedded vector SPFP instructions treat 64-bit registers as containing two 32-bit elements or four 16-bit elements as described in Section 3.8.1.3, “SPE APU Instructions.” However, like 32-bit Book E instructions, scalar SPFP APU floating-point instructions use bits 32–63 of the GPRs to hold 32-bit single-precision operands, as described in Section 3.8.1.4, “Embedded Floating-Point APU Instructions.”

The embedded double-precision floating-point APU (e500v2 only) uses the 64-bit GPRs to hold 64-bit, double-precision operands.

Figure 3-4 shows how the SPE and floating-point APU programming models compare, indicating how each APU uses the GPRs.



Note: Gray text indicates that APU does not use this register or register field.

[1] Formatting of floating-point operands is as defined by IEEE 754, as described in the APU chapter of the EREF.

Figure 3-4. SPE and Floating-Point APU GPR Usage

There is no record form of SPE or embedded floating-point instructions. Vector compare instructions store the result of the comparison into the CR. The meaning of the CR bits is now overloaded for vector operations. Vector compare instructions specify a CR field and two source registers as well as the type of compare: greater than, less than, or equal. Two bits in the CR field are written with the result of the vector compare, one for each element. The two defined bits could be used either by a vector select instruction or by a UISA branch instruction.

A partially visible accumulator register is architected for the integer and fractional multiply accumulate SPE instructions. It is described in Section 2.14.2, “Accumulator (ACC).”

Full descriptions of these instructions can be found in the “Instruction Set” chapter of the EREF.

3.8.1.1 SPE Operands: Signed Fractions

In signed fractional format, the N-bit operand is represented in a 1.[N–1] format (1 sign bit, N–1 fraction bits). Signed fractional numbers are in the following range:

$$-1.0 \le SF \le 1.0 - 2^{-(N-1)}$$

The real value of the binary operand SF[0:N–1] is as follows:

$$SF = -1.0 \bullet SF(0) + \sum_{i=1}^{N-1} SF(i) \bullet 2^{-i}$$

The most negative and positive numbers representable in fractional format are as follows:

- The most negative number is represented by SF(0) = 1 and SF[1:N–1] = 0 (that is, N=32; 0x8000_0000 = –1.0).
- The most positive number is represented by SF(0) = 0 and SF[1:N–1] = all 1s (that is, N=32; 0x7FFF_FFFF = $1.0 - 2^{-(N-1)}$).

3.8.1.2 SPE Integer and Fractional Operations

Figure 3-5 shows data formats for signed integer and fractional multiplication. Note that low word versions of signed saturate and signed modulo fractional instructions are not supported. Attempting to execute an opcode corresponding to these instructions causes boundedly undefined results.

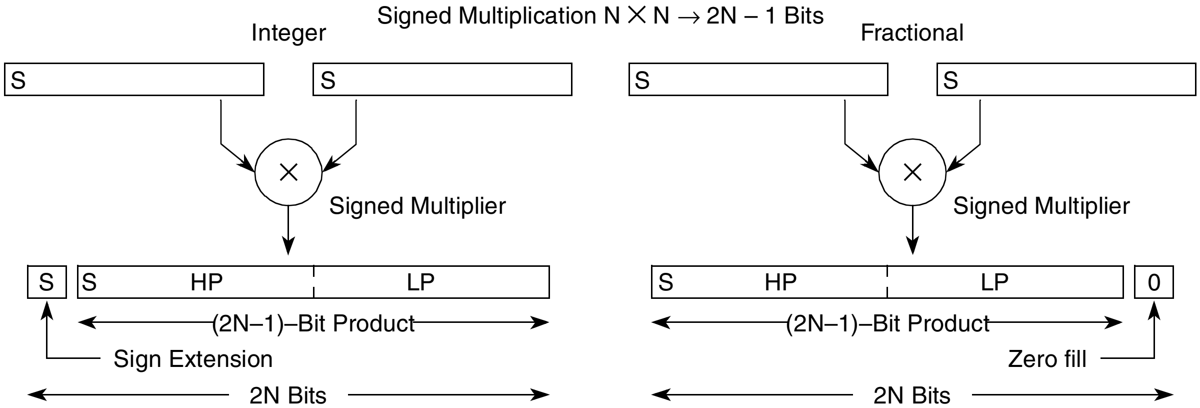


Figure 3-5. Integer and Fractional Operations

3.8.1.3 SPE APU Instructions

SPE APU instructions treat 64-bit GPRs as being composed of a vector of two 32-bit elements. (Some instructions also read or write 16-bit elements.) The SPE APU supports a number of forms of multiply and multiply-accumulate operations, and of add and subtract to accumulator operations. The SPE supports signed and unsigned forms, and optional fractional forms. For these instructions, the fractional form does not apply to unsigned forms because integer and fractional forms are identical for unsigned operands.

Table 3-34 shows how SPE APU vector multiply instruction mnemonics are structured.

Table 3-34. SPE APU Vector Multiply Instruction Mnemonic Structure

Prefix	Multiply Element		Data Type Element		Accumulate Element	
evm	**ho**	half odd (16x16→32)	**usi**	unsigned saturate integer	**a**	write to ACC
	he	half even (16x16→32)	**umi**	unsigned modulo integer	**aa**	write to ACC & added ACC
	hog	half odd guarded (16x16→32)	**ssi**	signed saturate integer	**an**	write to ACC & negate ACC
	heg	half even guarded (16x16→32)	**ssf**[1]	signed saturate fractional	**aaw**	write to ACC & ACC in words
	wh	word high (32x32→32)	**smi**	signed modulo integer	**anw**	write to ACC & negate ACC in words
	wl	word low (32x32→32)	**smf**[1]	signed modulo fractional		
	whg	word high guarded (32x32→32)				
	wlg	word low guarded (32x32→32)				
	w	word (32x32→64)				

[1] Low word versions of signed saturate and signed modulo fractional instructions are not supported. Attempting to execute an opcode corresponding to these instructions causes boundedly undefined results.

Table 3-35 defines mnemonic extensions for these instructions.

Table 3-35. Mnemonic Extensions for Multiply-Accumulate Instructions

Extension	Meaning	Comments
	Multiply Form	
he	Half word even	16×16→32
heg	Half word even guarded	16×16→32, 64-bit final accumulator result
ho	Half word odd	16×16→32
hog	Half word odd guarded	16×16→32, 64-bit final accumulator result
w	Word	32×32→64
wh	Word high	32×32→32, high-order 32 bits of product
wl	Word low	32×32→32, low-order 32 bits of product
	Data Type	
smf	Signed modulo fractional	(Wrap, no saturate)
smi	Signed modulo integer	(Wrap, no saturate)
ssf	Signed saturate fractional	
ssi	Signed saturate integer	
umi	Unsigned modulo integer	(Wrap, no saturate)
usi	Unsigned saturate integer	
	Accumulate Options	
a	Update accumulator	Update accumulator (no add)
aa	Add to accumulator	Add result to accumulator (64-bit sum)
aaw	Add to accumulator (words)	Add word results to accumulator words (pair of 32-bit sums)
an	Add negated	Add negated result to accumulator (64-bit sum)
anw	Add negated to accumulator (words)	Add negated word results to accumulator words (pair of 32-bit sums)

Table 3-36 lists SPE APU instructions.

Table 3-36. SPE APU Vector Instructions

Instruction	Mnemonic	Syntax
Bit Reversed Increment	**brinc**	rD,rA,rB
Initialize Accumulator	**evmra**	rD,rA
Multiply Half Words, Even, Guarded, Signed, Modulo, Fractional and Accumulate	**evmhegsmfaa**	rD,rA,rB
Multiply Half Words, Even, Guarded, Signed, Modulo, Fractional and Accumulate Negative	**evmhegsmfan**	rD,rA,rB
Multiply Half Words, Even, Guarded, Signed, Modulo, Integer and Accumulate	**evmhegsmiaa**	rD,rA,rB
Multiply Half Words, Even, Guarded, Signed, Modulo, Integer and Accumulate Negative	**evmhegsmian**	rD,rA,rB
Multiply Half Words, Even, Guarded, Unsigned, Modulo, Integer and Accumulate	**evmhegumiaa**	rD,rA,rB
Multiply Half Words, Even, Guarded, Unsigned, Modulo, Integer and Accumulate Negative	**evmhegumian**	rD,rA,rB
Multiply Half Words, Odd, Guarded, Signed, Modulo, Fractional and Accumulate	**evmhogsmfaa**	rD,rA,rB
Multiply Half Words, Odd, Guarded, Signed, Modulo, Fractional and Accumulate Negative	**evmhogsmfan**	rD,rA,rB

Table 3-36. SPE APU Vector Instructions (continued)

Instruction	Mnemonic	Syntax
Multiply Half Words, Odd, Guarded, Signed, Modulo, Integer and Accumulate	**evmhogsmiaa**	rD,rA,rB
Multiply Half Words, Odd, Guarded, Signed, Modulo, Integer and Accumulate Negative	**evmhogsmian**	rD,rA,rB
Multiply Half Words, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate	**evmhogumiaa**	rD,rA,rB
Multiply Half Words, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate Negative	**evmhogumian**	rD,rA,rB
Vector Absolute Value	**evabs**	rD,rA
Vector Add Immediate Word	**evaddiw**	rD,rB,UIMM
Vector Add Signed, Modulo, Integer to Accumulator Word	**evaddsmiaaw**	rD,rA,rB
Vector Add Signed, Saturate, Integer to Accumulator Word	**evaddssiaaw**	rD,rA
Vector Add Unsigned, Modulo, Integer to Accumulator Word	**evaddumiaaw**	rD,rA
Vector Add Unsigned, Saturate, Integer to Accumulator Word	**evaddusiaaw**	rD,rA
Vector Add Word	**evaddw**	rD,rA,rB
Vector AND	**evand**	rD,rA,rB
Vector AND with Complement	**evandc**	rD,rA,rB
Vector Compare Equal	**evcmpeq**	crD,rA,rB
Vector Compare Greater Than Signed	**evcmpgts**	crD,rA,rB
Vector Compare Greater Than Unsigned	**evcmpgtu**	crD,rA,rB
Vector Compare Less Than Signed	**evcmplts**	crD,rA,rB
Vector Compare Less Than Unsigned	**evcmpltu**	crD,rA,rB
Vector Convert Floating-Point to Unsigned Integer with Round toward Zero	**evfsctuiz**	rD,rB
Vector Count Leading Sign Bits Word	**evcntlsw**	rD,rA
Vector Count Leading Zeros Word	**evcntlzw**	rD,rA
Vector Divide Word Signed	**evdivws**	rD,rA,rB
Vector Divide Word Unsigned	**evdivwu**	rD,rA,rB
Vector Equivalent	**eveqv**	rD,rA,rB
Vector Extend Sign Byte	**evextsb**	rD,rA
Vector Extend Sign Half Word	**evextsh**	rD,rA
Vector Load Double into Half Words	**evldh**	rD,d(rA)
Vector Load Double into Half Words Indexed	**evldhx**	rD,rA,rB
Vector Load Double into Two Words	**evldw**	rD,d(rA)
Vector Load Double into Two Words Indexed	**evldwx**	rD,rA,rB
Vector Load Double Word into Double Word [1]	**evldd**	rD,d(rA)
Vector Load Double Word into Double Word Indexed [1]	**evlddx**	rD,rA,rB
Vector Load Half Word into Half Word Odd Signed and Splat	**evlhhossplat**	rD,d(rA)
Vector Load Half Word into Half Word Odd Signed and Splat Indexed	**evlhhossplatx**	rD,rA,rB
Vector Load Half Word into Half Word Odd Unsigned and Splat	**evlhhousplat**	rD,d(rA)
Vector Load Half Word into Half Word Odd Unsigned and Splat Indexed	**evlhhousplatx**	rD,rA,rB
Vector Load Half Word into Half Words Even and Splat	**evlhhesplat**	rD,d(rA)
Vector Load Half Word into Half Words Even and Splat Indexed	**evlhhesplatx**	rD,rA,rB

Table 3-36. SPE APU Vector Instructions (continued)

Instruction	Mnemonic	Syntax
Vector Load Word into Half Words and Splat	**evlwhsplat**	**r**D,**d(r**A**)**
Vector Load Word into Half Words and Splat Indexed	**evlwhsplatx**	**r**D,**r**A,**r**B
Vector Load Word into Half Words Odd Signed (with sign extension)	**evlwhos**	**r**D,**d(r**A**)**
Vector Load Word into Half Words Odd Signed Indexed (with sign extension)	**evlwhosx**	**r**D,**r**A,**r**B
Vector Load Word into Two Half Words Even	**evlwhe**	**r**D,**d(r**A**)**
Vector Load Word into Two Half Words Even Indexed	**evlwhex**	**r**D,**r**A,**r**B
Vector Load Word into Two Half Words Odd Unsigned (zero-extended)	**evlwhou**	**r**D,**d(r**A**)**
Vector Load Word into Two Half Words Odd Unsigned Indexed (zero-extended)	**evlwhoux**	**r**D,**r**A,**r**B
Vector Load Word into Word and Splat	**evlwwsplat**	**r**D,**d(r**A**)**
Vector Load Word into Word and Splat Indexed	**evlwwsplatx**	**r**D,**r**A,**r**B
Vector Merge High [1]	**evmergehi**	**r**D,**r**A,**r**B
Vector Merge High/Low	**evmergehilo**	**r**D,**r**A,**r**B
Vector Merge Low [1]	**evmergelo**	**r**D,**r**A,**r**B
Vector Merge Low/High	**evmergelohi**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Modulo, Fractional	**evmhesmf**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Modulo, Fractional and Accumulate into Words	**evmhesmfaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Modulo, Fractional and Accumulate Negative into Words	**evmhesmfanw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Modulo, Fractional, Accumulate	**evmhesmfa**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Modulo, Integer	**evmhesmi**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Modulo, Integer and Accumulate into Words	**evmhesmiaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Modulo, Integer and Accumulate Negative into Words	**evmhesmianw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Modulo, Integer, Accumulate	**evmhesmia**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Saturate, Fractional	**evmhessf**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Saturate, Fractional and Accumulate into Words	**evmhessfaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Saturate, Fractional and Accumulate Negative into Words	**evmhessfanw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Saturate, Fractional, Accumulate	**evmhessfa**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Saturate, Integer and Accumulate into Words	**evmhessiaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Saturate, Integer and Accumulate Negative into Words	**evmhessianw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Unsigned, Modulo, Integer	**evmheumi**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Unsigned, Modulo, Integer and Accumulate into Words	**evmheumiaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Unsigned, Modulo, Integer and Accumulate Negative into Words	**evmheumianw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Unsigned, Modulo, Integer, Accumulate	**evmheumia**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Unsigned, Saturate, Integer and Accumulate into Words	**evmheusiaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Unsigned, Saturate, Integer and Accumulate Negative into Words	**evmheusianw**	**r**D,**r**A,**r**B

Table 3-36. SPE APU Vector Instructions (continued)

Instruction	Mnemonic	Syntax
Vector Multiply Half Words, Odd, Signed, Modulo, Fractional	**evmhosmf**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Modulo, Fractional and Accumulate into Words	**evmhosmfaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Modulo, Fractional and Accumulate Negative into Words	**evmhosmfanw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Modulo, Fractional, Accumulate	**evmhosmfa**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Modulo, Integer	**evmhosmi**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Modulo, Integer and Accumulate into Words	**evmhosmiaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Modulo, Integer and Accumulate Negative into Words	**evmhosmianw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Modulo, Integer, Accumulate	**evmhosmia**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Saturate, Fractional	**evmhossf**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Saturate, Fractional and Accumulate into Words	**evmhossfaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Saturate, Fractional and Accumulate Negative into Words	**evmhossfanw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Saturate, Fractional, Accumulate	**evmhossfa**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Saturate, Integer and Accumulate into Words	**evmhossiaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Saturate, Integer and Accumulate Negative into Words	**evmhossianw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer	**evmhoumi**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer and Accumulate into Words	**evmhoumiaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer and Accumulate Negative into Words	**evmhoumianw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer, Accumulate	**evmhoumia**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Unsigned, Saturate, Integer and Accumulate into Words	**evmhousiaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Unsigned, Saturate, Integer and Accumulate Negative into Words	**evmhousianw**	**r**D,**r**A,**r**B
Vector Multiply Word High Signed, Modulo, Fractional	**evmwhsmf**	**r**D,**r**A,**r**B
Vector Multiply Word High Signed, Modulo, Fractional and Accumulate	**evmwhsmfa**	**r**D,**r**A,**r**B
Vector Multiply Word High Signed, Modulo, Integer	**evmwhsmi**	**r**D,**r**A,**r**B
Vector Multiply Word High Signed, Modulo, Integer and Accumulate	**evmwhsmia**	**r**D,**r**A,**r**B
Vector Multiply Word High Signed, Saturate, Fractional	**evmwhssf**	**r**D,**r**A,**r**B
Vector Multiply Word High Signed, Saturate, Fractional and Accumulate	**evmwhssfa**	**r**D,**r**A,**r**B
Vector Multiply Word High Unsigned, Modulo, Integer	**evmwhumi**	**r**D,**r**A,**r**B
Vector Multiply Word High Unsigned, Modulo, Integer and Accumulate	**evmwhumia**	**r**D,**r**A,**r**B
Vector Multiply Word Low Signed, Modulo, Integer and Accumulate in Words	**evmwlsmiaaw**	**r**D,**r**A,**r**B
Vector Multiply Word Low Signed, Modulo, Integer and Accumulate Negative in Words	**evmwlsmianw**	**r**D,**r**A,**r**B
Vector Multiply Word Low Signed, Saturate, Integer and Accumulate in Words	**evmwlssiaaw**	**r**D,**r**A,**r**B
Vector Multiply Word Low Signed, Saturate, Integer and Accumulate Negative in Words	**evmwlssianw**	**r**D,**r**A,**r**B
Vector Multiply Word Low Unsigned, Modulo, Integer	**evmwlumi**	**r**D,**r**A,**r**B
Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate	**evmwlumia**	**r**D,**r**A,**r**B
Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate in Words	**evmwlumiaaw**	**r**D,**r**A,**r**B

Table 3-36. SPE APU Vector Instructions (continued)

Instruction	Mnemonic	Syntax
Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate Negative in Words	**evmwlumianw**	rD,rA,rB
Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate in Words	**evmwlusiaaw**	rD,rA,rB
Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate Negative in Words	**evmwlusianw**	rD,rA,rB
Vector Multiply Word Signed, Modulo, Fractional	**evmwsmf**	rD,rA,rB
Vector Multiply Word Signed, Modulo, Fractional and Accumulate	**evmwsmfa**	rD,rA,rB
Vector Multiply Word Signed, Modulo, Fractional and Accumulate	**evmwsmfaa**	rD,rA,rB
Vector Multiply Word Signed, Modulo, Fractional and Accumulate Negative	**evmwsmfan**	rD,rA,rB
Vector Multiply Word Signed, Modulo, Integer	**evmwsmi**	rD,rA,rB
Vector Multiply Word Signed, Modulo, Integer and Accumulate	**evmwsmia**	rD,rA,rB
Vector Multiply Word Signed, Modulo, Integer and Accumulate	**evmwsmiaa**	rD,rA,rB
Vector Multiply Word Signed, Modulo, Integer and Accumulate Negative	**evmwsmian**	rD,rA,rB
Vector Multiply Word Signed, Saturate, Fractional	**evmwssf**[2]	rD,rA,rB
Vector Multiply Word Signed, Saturate, Fractional and Accumulate	**evmwssfa** [2]	rD,rA,rB
Vector Multiply Word Signed, Saturate, Fractional and Accumulate [3]	**evmwssfaa**	rD,rA,rB
Vector Multiply Word Signed, Saturate, Fractional and Accumulate Negative [3]	**evmwssfan**	rD,rA,rB
Vector Multiply Word Unsigned, Modulo, Integer	**evmwumi**	rD,rA,rB
Vector Multiply Word Unsigned, Modulo, Integer and Accumulate	**evmwumia**	rD,rA,rB
Vector Multiply Word Unsigned, Modulo, Integer and Accumulate	**evmwumiaa**	rD,rA,rB
Vector Multiply Word Unsigned, Modulo, Integer and Accumulate Negative	**evmwumian**	rD,rA,rB
Vector NAND	**evnand**	rD,rA,rB
Vector Negate	**evneg**	rD,rA
Vector NOR	**evnor**	rD,rA,rB
Vector OR	**evor**	rD,rA,rB
Vector OR with Complement	**evorc**	rD,rA,rB
Vector Rotate Left Word	**evrlw**	rD,rA,rB
Vector Rotate Left Word Immediate	**evrlwi**	rD,rA,UIMM
Vector Round Word	**evrndw**	rD,rA
Vector Select	**evsel**	rD,rA,rB,crS
Vector Shift Left Word	**evslw**	rD,rA,rB
Vector Shift Left Word Immediate	**evslwi**	rD,rA,UIMM
Vector Shift Right Word Immediate Signed	**evsrwis**	rD,rA,UIMM
Vector Shift Right Word Immediate Unsigned	**evsrwiu**	rD,rA,UIMM
Vector Shift Right Word Signed	**evsrws**	rD,rA,rB
Vector Shift Right Word Unsigned	**evsrwu**	rD,rA,rB
Vector Splat Fractional Immediate	**evsplatfi**	rD,SIMM
Vector Splat Immediate	**evsplati**	rD,SIMM
Vector Store Double of Double [1]	**evstdd**	rS,d(rA)
Vector Store Double of Double Indexed [1]	**evstddx**	rS,rA,rB

Table 3-36. SPE APU Vector Instructions (continued)

Instruction	Mnemonic	Syntax
Vector Store Double of Four Half Words	**evstdh**	**r**S,**d(r**A**)**
Vector Store Double of Four Half Words Indexed	**evstdhx**	**r**S,**r**A,**r**B
Vector Store Double of Two Words	**evstdw**	**r**S,**d(r**A**)**
Vector Store Double of Two Words Indexed	**evstdwx**	**r**S,**r**A,**r**B
Vector Store Word of Two Half Words from Even	**evstwhe**	**r**S,**d(r**A**)**
Vector Store Word of Two Half Words from Even Indexed	**evstwhex**	**r**S,**r**A,**r**B
Vector Store Word of Two Half Words from Odd	**evstwho**	**r**S,**d(r**A**)**
Vector Store Word of Two Half Words from Odd Indexed	**evstwhox**	**r**S,**r**A,**r**B
Vector Store Word of Word from Even	**evstwwe**	**r**S,**d(r**A**)**
Vector Store Word of Word from Even Indexed	**evstwwex**	**r**S,**r**A,**r**B
Vector Store Word of Word from Odd	**evstwwo**	**r**S,**d(r**A**)**
Vector Store Word of Word from Odd Indexed	**evstwwox**	**r**S,**r**A,**r**B
Vector Subtract from Word	**evsubfw**	**r**D,**r**A,**r**B
Vector Subtract Immediate from Word	**evsubifw**	**r**D,UIMM,**r**B
Vector Subtract Signed, Modulo, Integer to Accumulator Word	**evsubfsmiaaw**	**r**D,**r**A
Vector Subtract Signed, Saturate, Integer to Accumulator Word	**evsubfssiaaw**	**r**D,**r**A
Vector Subtract Unsigned, Modulo, Integer to Accumulator Word	**evsubfumiaaw**	**r**D,**r**A
Vector Subtract Unsigned, Saturate, Integer to Accumulator Word	**evsubfusiaaw**	**r**D,**r**A
Vector XOR	**evxor**	**r**D,**r**A,**r**B

[1] These instructions are also used by the vector and double-precision scalar floating-point APUs.

[2] The architecture specifies that if the final result cannot be represented in 64 bits, SPEFSCR[OV] should be set (along with the SOV bit, if it is not already set). The e500 violates the architectural specification for these instructions because it sets the overflow bit in cases where there is no overflow.

[3] Although the e500 records any overflow resulting from the addition/subtraction portion of these instructions, a saturate value is not saved to rD or the accumulator. The architecture specifies that the intermediate result should be saturated if it cannot be represented in 64 bits. The also architecture specifies that the final result should be saturated if it cannot be represented in 64 bits. The e500 does not saturate in either case.

3.8.1.4 Embedded Floating-Point APU Instructions

The vector and scalar SPFP APUs perform floating-point operations on single-precision operands. These operations are IEEE 754–compliant with software exception handlers and offer a simpler exception model than the floating-point instructions defined by the PowerPC ISA. Instead of FPRs, these instructions use GPRs to offer improved performance for converting between floating-point, integer, and fractional values. Sharing GPRs allows vector floating-point instructions to use SPE load and store instructions.

The embedded floating-point APUs are described as follows:

- Vector SPFP instructions operate on a vector of two 32-bit, single-precision floating-point numbers that reside in the upper and lower halves of the 64-bit GPRs.
- Scalar SPFP instructions operate on single 32-bit operands that reside in the lower 32-bits of the GPRs.
- Scalar DPFP instructions (e500v2 only) operate on single 64-bit operands that reside in the 64-bit GPRs. Full descriptions of these instructions is provided in Section 10.4, "Double-Precision Floating-Point APU (e500 v2 Only)."

These instructions are listed in Table 3-37.

NOTE

Vector and scalar versions of the instructions have the same syntax.

Table 3-37. Vector and Scalar Floating-Point APU Instructions

Instruction	Single-Precision Scalar	Double-Precision Scalar (e500v2)	Vector	Syntax
Convert Floating-Point Double- from Single-Precision	—	**efdcfs**	—	rD,rB
Convert Floating-Point from Signed Fraction	**efscfsf**	**efdcfsf**	**evfscfsf**	rD,rB
Convert Floating-Point from Signed Integer	**efscfsi**	**efdcfsi**	**evfscfsi**	rD,rB
Convert Floating-Point from Unsigned Fraction	**efscfuf**	**efdcfuf**	**evfscfuf**	rD,rB
Convert Floating-Point from Unsigned Integer	**efscfui**	**efdcfui**	**evfscfui**	rD,rB
Convert Floating-Point Single- from Double-Precision	—	**efscfd**	—	rD,rB
Convert Floating-Point to Signed Fraction	**efsctsf**	**efdctsf**	**evfsctsf**	rD,rB
Convert Floating-Point to Signed Integer	**efsctsi**	**efdctsi**	**evfsctsi**	rD,rB
Convert Floating-Point to Signed Integer with Round toward Zero	**efsctsiz**	**efdctsiz**	**evfsctsiz**	rD,rB
Convert Floating-Point to Unsigned Fraction	**efsctuf**	**efdctuf**	**evfsctuf**	rD,rB
Convert Floating-Point to Unsigned Integer	**efsctui**	**efdctui**	**evfsctui**	rD,rB
Convert Floating-Point to Unsigned Integer with Round toward Zero	**efsctuiz**	**efdctuiz**	**evfsctuiz**	rD,rB
Floating-Point Absolute Value	**efsabs** [1]	**efdabs**	**evfsabs**	rD,rA
Floating-Point Add	**efsadd**	**efdadd**	**evfsadd**	rD,rA,rB
Floating-Point Compare Equal	**efscmpeq**	**efdcmpeq**	**evfscmpeq**	crD,rA,rB
Floating-Point Compare Greater Than	**efscmpgt**	**efdcmpgt**	**evfscmpgt**	crD,rA,rB
Floating-Point Compare Less Than	**efscmplt**	**efdcmplt**	**evfscmplt**	crD,rA,rB
Floating-Point Divide	**efsdiv**	**efddiv**	**evfsdiv**	rD,rA,rB
Floating-Point Multiply	**efsmul**	**efdmul**	**evfsmul**	rD,rA,rB
Floating-Point Negate	**efsneg** [1]	**efdneg**	**evfsneg**	rD,rA
Floating-Point Negative Absolute Value	**efsnabs** [1]	**efdnabs**	**evfsnabs**	rD,rA
Floating-Point Subtract	**efssub**	**efdsub**	**evfssub**	rD,rA,rB

Table 3-37. Vector and Scalar Floating-Point APU Instructions (continued)

Instruction	Single-Precision Scalar	Double-Precision Scalar (e500v2)	Vector	Syntax
Floating-Point Test Equal	**efststeq**	**efdtsteq**	**evfststeq**	**cr**D,**r**A,**r**B
Floating-Point Test Greater Than	**efststgt**	**efdtstgt**	**evfststgt**	**cr**D,**r**A,**r**B
Floating-Point Test Less Than	**efststlt**	**efdtstlt**	**evfststlt**	**cr**D,**r**A,**r**B
SPE Double Word Load/Store Instructions				
Vector Load Double Word into Double Word	—	**evldd**		**r**D,**d(r**A**)**
Vector Load Double Word into Double Word Indexed	—	**evlddx**		**r**D,**r**A,**r**B
Vector Merge High	—	**evmergehi**		**r**D,**r**A,**r**B
Vector Merge Low	—	**evmergelo**		**r**D,**r**A,**r**B
Vector Store Double of Double	—	**evstdd**		**r**S,**d(r**A**)**
Vector Store Double of Double Indexed	—	**evstddx**		**r**S,**r**A,**r**B

Note: on e500v1, floating-point operations that produce a result of zero may generate an incorrect sign.

[1] Exception detection for these instructions is implementation dependent. On the e500, Infinities, NaNs, and Denorms are always be treated as Norms. No exceptions are taken if SPEFSCR[FINVE] = 1.

3.8.2 Integer Select (isel) APU

The integer select APU consists of the **isel** instruction, a conditional register move that helps eliminate branches. Further information about **isel** may be found in the APUs chapter of the EREF.

Table 3-38. Integer Select APU Instruction

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

3.8.3 Performance Monitor APU

The e500 core complex implements a performance monitor as an APU. Software communication with the performance monitor APU is achieved through performance monitor registers (PMRs) rather than SPRs. New instructions are provided to move to and from these PMRs. Performance monitor APU instructions are described in Table 3-39. Full descriptions of these instructions can be found in the EREF chapter, “Instruction Set.”

Table 3-39. Performance Monitor APU Instructions

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

PMR encodings are shown in Table 3-40.

Table 3-40. e500-Defined PMR Encodings

Register Name	PMR			Privilege	Access
	Decimal	pmr[5–9]	pmr[0–4]		
UMMCR0	936	11101	01000	User	Read only
UMMCR1	940	11101	01100	User	Read only
UMMCR2	928	11101	00111	User	Read only
UPMC1	937	11101	01001	User	Read only
UPMC2	938	11101	01010	User	Read only
UPMC3	941	11101	01101	User	Read only
UPMC4	942	11101	01110	User	Read only
USIAR	939	11101	01011	User	Read only

3.8.4 Cache Locking APU

This section describes the instructions in the cache locking APU, which consists of the instructions described in Table 3-41.

Table 3-41. Cache Locking APU Instructions

Name	Mnemonic	Syntax	Implementation Details
Data Cache Block Lock Clear	**dcblc**	CT,**r**A,**r**B	If CT=0 and the line is in the L1 data cache, the data cache lock bit for that line is cleared, making it eligible for replacement. If CT=1 and the line is in the L2 cache, the lock bit for that line is cleared, making it eligible for replacement.
Data Cache Block Touch and Lock Set	**dcbtls**	CT,**r**A,**r**B	If CT=0, the line is loaded and locked into the L1 data cache. If CT=1, the line is loaded and locked in the unified L2 cache. If CT=1 and the block is already in the L2 cache, **dcbtls** marks the block so it is not a candidate for replacement.
Data Cache Block Touch for Store and Lock Set	**dcbtstls**	CT,**r**A,**r**B	If CT = 0, the e500 core fetches the block containing the byte addressed by EA into the data cache. After the block containing the byte is fetched, it is locked. If CT = 0 and the block is in the data cache, **dcbtstls** marks the block locked so it is no longer eligible for replacement. If CT=1 and the block is in the L2 cache, **dcbtstls** marks the block such that it should not be selected for replacement. If CT is not 0 or 1, **dcbtstls** is no-oped. In the L1 data cache, the e500 implements a lock bit for every index and way, allowing a line locking granularity.
Instruction Cache Block Lock Clear	**icblc**	CT,**r**A,**r**B	If CT=0 and the line is in the instruction cache, the lock bit for that line is cleared, making it eligible for replacement. If CT=1 and the line is in the L2 cache, the lock bit for that line is cleared in the L2 cache, making it eligible for replacement. If CT is not 0 or 1, the **icblc** is no-oped.
Instruction Cache Block Touch and Lock Set	**icbtls**	CT,**r**A,**r**B	If CT=0, the line is loaded and locked into the L1 instruction cache. If CT=1, the line is loaded into the unified L2 cache and the line is locked into the L2 cache. If CT=1 and the block already exists in the L2 cache, **icbtls** marks it such that it should not be selected for replacement. If CT is not 0 or 1, **icbtls** is no-oped.

Full descriptions of these instructions can be found in the EREF chapter, “Instruction Set.” Note the following:

- In the L1 data cache the e500 implements a lock bit for every index and way, allowing a line locking granularity. Setting CT = 0 specifies the L1 cache.
- The e500 supports CT = 0 and CT = 1. If CT = 0, the L1 cache is targeted. If CT = 1, the unified L2 cache is targeted.
- If the CT value is not supported, the instruction is treated as a no-op.
- Note that setting L1CSR0[DCLFI] flash invalidates all data cache lock bits and setting L1CSR0[ICLFI] flash invalidates all instruction cache lock bits, allowing system software to clear all cache locking in the L1 cache without knowing the addresses of the lines locked.
- Overlocking occurs when **dcbtls**, **dcbtstls**, or **icbtls** is performed to an index in either the L1 or L2 cache that already has all ways locked. In the e500, overlocking does not generate an exception; instead, if a touch and lock set is performed with CT = 0 to an index in which all cache ways are already locked, the least recently used way is evicted and L1CSR0[CLO] is set indicating an overlock; the new line is not locked or cached.

  To precisely detect an overlock condition in the data cache, system software must perform the following code sequence:

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

  The following code sequence is used to precisely detect an overlock in the instruction cache:

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

- Touch and lock set instructions (**icbtls**, **dcbtls**, and **dcbtstls**) are always executed and are not treated as hints. When one of these instructions is performed to an index and the way cannot be locked, L1CSR1[ICUL] or L1CSR0[CUL] is set to indicate an unable-to-lock condition. This occurs if the instruction must be no-oped.

The e500 implements a flash clear for all data cache lock bits (using L1CSR0[CLFR]) and in the instruction cache (using L1CSR1[ICLFR]). This allows system software to clear all data cache locking bits without knowing the addresses of the lines locked.

3.8.5 Machine Check APU

The machine check APU defines a separate interrupt type for machine check interrupts. It provides additional save and restore SPRs (MCSRR and MCSRR1). The Return from Machine Check Interrupt instruction (**rfmci**), is described in Table 3-42.

Table 3-42. Machine Check APU Instruction

Name	Mnemonic	Syntax	Implementation Notes
Return from Machine Check Interrupt	**rfmci**	—	When **rfmci** is executed, the values in MCSRR0 and MCSRR1 are restored. **rfmci** is context-synchronizing; it works its way to the final execute stage, updates architected registers, and redirects instruction flow.

3.9 e500-Specific Instructions

The e500 implements the branch target buffer locking APU, which is not part of the EIS. It defines the two instructions described in the following section.

3.9.1 Branch Target Buffer (BTB) Locking Instructions

The e500 core complex provides a 512-entry BTB for efficient processing of branch instructions. The BTB is a branch target address cache (BTAC), organized as 128 rows with four-way set associativity, that holds the address of the target instruction of the 512 most-recently taken branches. Table 3-43 lists the BTB instructions.

Table 3-43. Branch Target Buffer (BTB) Instructions

Name	Mnemonic	Syntax
Branch Buffer Entry Lock Reset	**bbelr**	—
Branch Buffer Load Entry and Lock Set	**bblels**	—

The branch buffer entry address register (BBEAR) and the branch buffer target address register (BBTAR) are defined in the e500 core complex for enabling the locking and unlocking of BTB entries. They can be read and written in both user and supervisor modes with **mfspr** and **mtspr**. The user branch locking enable bit, MSR[UBLE], is defined to allow user mode programs to lock or unlock entries in the BTB. See Chapter 4, “Execution Timing.”

bbelr

BTB APU | User

bbelr

Branch Buffer Entry Lock Reset

bbelr

0					5	6														20	21										31
0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	1	1	0	0

```
bbea ← BBEAR0:29
BranchBufferEntryLockReset(bbea)
```

A BTB entry associated with the effective address specified in BBEAR has its lock reset. If no BTB entry is associated with the address, or if the entry exists but it is not locked, the instruction is a no-op and no other status is reported. After **bbelr** executes, the entry continues to be valid in the BTB with all its attributes unchanged.

This instruction can always be executed in supervisor mode. In user mode, if MSR[UBLE] is cleared, a privileged instruction exception is taken; if MSR[UBLE] is set, the instruction executes without a privileged instruction exception.

bblels

BTB APU	User

bblels

Branch Buffer Load Entry and Lock Set

bblels

0–5	6–20	21–30	31
0 1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	1 0 0 0 0 0 0 1 1 0	0

```
bbea ← BBEAR0:29
bbta ← BBTAR0:29
bbiab ← BBEAR30:31, BBTAR30
bbdir ← BBTAR31
BranchBufferLoadEntryAndLockSet(bbea, bbta)
```

An effective address associated with a branch instruction and the corresponding branch target address are loaded into a BTB entry and locked. It is marked with the prediction that the user supplies in BBTAR[31]. 1 is taken, 0 is not taken.

If the BTB is disabled, the instruction is a no-op and BUCSR[BBUL] is set. If there already exists another entry in the BTB associated with the address in the BBEAR and that entry is not locked, the target address of that entry is overwritten and the entry is then locked. If there already exists a locked entry in the BTB associated with the address in the BBEAR, the target address of that entry is overwritten with the target address in the BBTAR and BUCSR[BBLO] is set. If all the ways of the BTB are locked for the index to which the BBEAR maps, one of the existing entries is overwritten with the new one and BUCSR[BBLO] is set.

The user can pick the direction of the locked branch target address by programming bit 31 of BBTAR (BBTAR[BDIR]). If BDIR = 1, the locked address is always predicted as taken; if BDIR = 0, the locked address is always predicted as not taken.

The bbiab is a 3-bit pointer (BBEAR[IAB0,IAB1]|BBTAR[IAB2]) to the instruction after the branch. It has values from 0 to 7, based on the location in the cache block of the instruction following the branch.

This instruction can always be executed in supervisor mode. In user mode, if MSR[UBLE] is cleared, a privileged instruction exception is taken; if MSR[UBLE] is set, the instruction executes without a privileged instruction exception.

3.10 Instruction Listing

Table 3-44 lists instructions defined in Book E, in the PowerPC architecture, and in the e500. A check mark (√) or text in a column indicates that the instruction is defined or implemented. The e500-specific instructions are indicated in the e500 column by the name of the facility (BTB locking, SPE APU, cache locking) that defines the instruction.

Table 3-44. List of Instructions

Mnemonic	Book E	PowerPC AIM	e500	Mnemonic	Book E	PowerPC AIM	e500
addc[**o**][.]	√	√	√	**evmwsmiaa**			SPE APU
adde[**o**][.]	√	√	√	**evmwsmian**			SPE APU
addi	√	√	√	**evmwssf**			SPE APU
addic[.]	√	√	√	**evmwssfa**			SPE APU
addis	√	√	√	**evmwssfaa**			SPE APU
addme[**o**][.]	√	√	√	**evmwssfan**			SPE APU
add[**o**].]	√	√	√	**evmwumi**			SPE APU
addze[**o**][.]	√	√	√	**evmwumia**			SPE APU
andc[.]	√	√	√	**evmwumiaa**			SPE APU
andi.	√	√	√	**evmwumian**			SPE APU
andis.	√	√	√	**evnand**			SPE APU
and[.]	√	√	√	**evneg**			SPE APU
b	√	√	√	**evnor**			SPE APU
ba	√	√	√	**evor**			SPE APU
bbelr			BTB	**evorc**			SPE APU
bblels			BTB	**evrlw**			SPE APU
bc	√	√	√	**evrlwi**			SPE APU
bca	√	√	√	**evrndw**			SPE APU
bcctr	√	√	√	**evsel**			SPE APU
bcctrl	√	√	√	**evslw**			SPE APU
bcl	√	√	√	**evslwi**			SPE APU
bcla	√	√	√	**evsplatfi**			SPE APU
bclr	√	√	√	**evsplati**			SPE APU
bclrl	√	√	√	**evsrwis**			SPE APU
bl	√	√	√	**evsrwiu**			SPE APU
bla	√	√	√	**evsrws**			SPE APU
brinc			SPE APU	**evsrwu**			SPE APU
cmp	√	√	√	**evstdd**			SPE APU
cmpi	√	√	√	**evstddx**			SPE APU
cmpl	√	√	√	**evstdh**			SPE APU
cmpli	√	√	√	**evstdhx**			SPE APU
cntlzw[.]	√	√	√	**evstdw**			SPE APU
crand	√	√	√	**evstdwx**			SPE APU

Table 3-44. List of Instructions (continued)

Mnemonic	Book E	PowerPC AIM	e500	Mnemonic	Book E	PowerPC AIM	e500
crandc	√	√	√	**evstwhe**			SPE APU
creqv	√	√	√	**evstwhex**			SPE APU
crnand	√	√	√	**evstwho**			SPE APU
crnor	√	√	√	**evstwhox**			SPE APU
cror	√	√	√	**evstwwex**			SPE APU
crorc	√	√	√	**evstwwex**			SPE APU
crxor	√	√	√	**evstwwo**			SPE APU
dcba	√	√	√	**evstwwox**			SPE APU
dcbf	√	√	√	**evsubfsmiaaw**			SPE APU
dcbi	√	√	√	**evsubfssiaaw**			SPE APU
dcblc			Cache locking	**evsubfumiaaw**			SPE APU
dcbst	√	√	√	**evsubfusiaaw**			SPE APU
dcbt	√	√	√	**evsubfw**			SPE APU
dcbtls			Cache locking	**evsubifw**			SPE APU
dcbtst	√	√	√	**evxor**			SPE APU
dcbtstls			Cache locking	**extsb**[.]	√	√	√
dcbz	√	√	√	**extsh**[.]	√	√	√
divw[**o**][.]	√	√	√	**extsw.**		64-bit only	
divwu[**o**][.]	√	√	√	**fabs**[.]	√	√	
eciwx		√		**fadds**[.]	√	√	
ecowx		√		**fadd**[.]	√	√	
efdabs			DPFP (e500v2)	**fcfid**[.]	√	√	
efdadd			DPFP (e500v2)	**fcmpo**	√	√	
efdcfs			DPFP (e500v2)				
efdcfsf			DPFP (e500v2)	**fcmpu**	√	√	
efdcfsi			DPFP (e500v2)	**fctidz**[.]	√	√	
efdcfuf			DPFP (e500v2)	**fctid**[.]	√	√	
efdcfui			DPFP (e500v2)	**fctiwz**[.]	√	√	
efdcmpeq			DPFP (e500v2)	**fctiw**[.]	√	√	
efdcmpgt			DPFP (e500v2)	**fdivs**[.]	√	√	
efdcmplt			DPFP (e500v2)	**fdiv**[.]	√	√	
efdctsf			DPFP (e500v2)	**fmadds**[.]	√	√	
efdctsi			DPFP (e500v2)	**fmadd**[.]	√	√	
efdctsiz			DPFP (e500v2)	**fmr**[.]	√	√	
efdctuf			DPFP (e500v2)	**fmsubs**[.]	√	√	
efdctui			DPFP (e500v2)	**fmsub**[.]	√	√	
efdctuiz			DPFP (e500v2)	**fmuls**[.]	√	√	
efddiv			DPFP (e500v2)	**fmul**[.]	√	√	
efdmul			DPFP (e500v2)	**fnabs**[.]	√	√	

Table 3-44. List of Instructions (continued)

Mnemonic	Book E	PowerPC AIM	e500	Mnemonic	Book E	PowerPC AIM	e500
efdnabs			DPFP (e500v2)	**fneg**[.]	√	√	
efdneg			DPFP (e500v2)	**fnmadds**[.]	√	√	
efdsub			DPFP (e500v2)	**fnmadd**[.]	√	√	
efdtsteq			DPFP (e500v2)	**fnmsubs**[.]	√	√	
efdtstgt			DPFP (e500v2)	**fnmsub**[.]	√	√	
efdtstlt			DPFP (e500v2)	**fres**[.]	√	√	
efsabs			Scalar SPFP	**frsp**[.]	√	√	
efsadd			Scalar SPFP	**frsqrte**[.]	√	√	
efscfd			DPFP (e500v2)	**fsel**[.]	√	√	
efscfsf			Scalar SPFP	**fsqrts**[.]	√	√	
efscfsi			Scalar SPFP	**fsqrt**[.]	√	√	
efscfuf			Scalar SPFP	**fsubs**[.]	√	√	
efscfui			Scalar SPFP	**fsub**[.]	√	√	
efscmpeq			Scalar SPFP	**icbi**	√	√	√
efscmpgt			Scalar SPFP	**icblc**			Cache locking
efscmplt			Scalar SPFP	**icbt**	√		√
efsctsf			Scalar SPFP	**icbtls**			Cache locking
efsctsi			Scalar SPFP	**isel**			Integer select
efsctsiz			Scalar SPFP	**isync**	√	√	√
efsctuf			Scalar SPFP	**lbz**	√	√	√
efsctui			Scalar SPFP	**lbzu**	√	√	√
efsctuiz			Scalar SPFP	**lbzux**	√	√	√
efsdiv			Scalar SPFP	**lbzx**	√	√	√
efsmul			Scalar SPFP	**ld**		√	
efsnabs			Scalar SPFP	**ldarx**		√	
efsneg			Scalar SPFP	**ldu**		√	
efssub			Scalar SPFP	**ldux**		√	
efststeq			Scalar SPFP	**ldx**		√	
efststgt			Scalar SPFP	**lfd**	√	√	
efststlt			Scalar SPFP	**lfdu**	√	√	
eieio	Replaced with **mbar**	√		**lfdux**	√	√	
eqv[.]	√	√	√	**lfdx**	√	√	
evabs			SPE APU	**lfs**	√	√	
evaddiw			SPE APU	**lfsu**	√	√	
evaddsmiaaw			SPE APU	**lfsux**	√	√	
evaddssiaaw			SPE APU	**lfsx**	√	√	
evaddumiaaw			SPE APU	**lha**	√	√	√
evaddusiaaw			SPE APU	**lhau**	√	√	√

Table 3-44. List of Instructions (continued)

Mnemonic	Book E	PowerPC AIM	e500	Mnemonic	Book E	PowerPC AIM	e500
evaddw			SPE APU	**lhaux**	√	√	√
evand			SPE APU	**lhax**	√	√	√
evandc			SPE APU	**lhbrx**	√	√	√
evcmpeq			SPE APU	**lhz**	√	√	√
evcmpgts			SPE APU	**lhzu**	√	√	√
evcmpgtu			SPE APU	**lhzux**	√	√	√
evcmplts			SPE APU	**lhzx**	√	√	√
evcmpltu			SPE APU	**lmw**	√	√	√
evcntlsw			SPE APU	**lswi**	√	√	
evcntlzw			SPE APU	**lswx**	√	√	
evdivws			SPE APU	**lwa**		√	
evdivwu			SPE APU	**lwarx**	√	√	√
eveqv			SPE APU	**lwaux**		√	
evextsb			SPE APU	**lwax**		√	
evextsh			SPE APU	**lwbrx**	√	√	√
evfsabs			Vector SPFP	**lwz**	√	√	√
evfsadd			Vector SPFP	**lwzu**	√	√	√
evfscfsf			Vector SPFP	**lwzux**	√	√	√
evfscfsi			Vector SPFP	**lwzx**	√	√	√
evfscfuf			Vector SPFP	**mbar**	√		√
evfscfui			Vector SPFP	**mcrf**	√	√	√
evfscmpeq			Vector SPFP	**mcrfs**	√	√	
evfscmpgt			Vector SPFP	**mcrxr**	√	√	√
evfscmplt			Vector SPFP	**mfapidi**	√		
evfsctsf			Vector SPFP	**mfcr**	√	√	√
evfsctsi			Vector SPFP	**mfdcr**	√		
evfsctsiz			Vector SPFP	**mffs**[.]	√	√	
evfsctuf			Vector SPFP	**mfmsr**	√	√	√
evfsctui			Vector SPFP	**mfpmr**			Performance monitor
evfsctuiz			Vector SPFP	**mfspr**	√	√	√
evfsdiv			Vector SPFP	**mfsr**		√	
evfsmul			Vector SPFP	**mfsrin**		√	
evfsnabs			Vector SPFP	**mftb**		√	
evfsneg			Vector SPFP	**msync**	√		√
evfssub			Vector SPFP	**mtcrf**	√	√	√
evfststeq			Vector SPFP	**mtdcr**	√		
evfststgt			Vector SPFP	**mtfsb0**[.]	√	√	
evfststlt			Vector SPFP	**mtfsb1**[.]	√	√	

Table 3-44. List of Instructions (continued)

Mnemonic	Book E	PowerPC AIM	e500	Mnemonic	Book E	PowerPC AIM	e500
evldd			SPE APU	**mtfsfi**[.]	√	√	
evlddx			SPE APU	**mtfsf**[.]	√	√	
evldh			SPE APU	**mtmsr**	√	√	√
evldhx			SPE APU	**mtmsrd**		64-bit only	
evldw			SPE APU	**mtpmr**			Performance monitor
evldwx			SPE APU	**mtspr**	√	√	√
evlhhesplat			SPE APU	**mtsr**		√	
evlhhesplatx			SPE APU	**mtsrd**		√	
evlhhossplat			SPE APU	**mtsrdin**		√	
evlhhossplatx			SPE APU	**mtsrin**		√	
evlhhousplat			SPE APU	**mulhd.**		√	
evlhhousplatx			SPE APU	**mulhdu.**		√	
evlwhe			SPE APU	**mulhwu**[.]	√	√	√
evlwhex			SPE APU	**mulhw**[.]	√	√	√
evlwhos			SPE APU	**mulld.**		√	
evlwhosx			SPE APU	**mulldo.**		√	
evlwhou			SPE APU	**mulli**	√	√	√
evlwhoux			SPE APU	**mullw**[**o**][.]	√	√	√
evlwhsplat			SPE APU	**nand**[.]	√	√	√
evlwhsplatx			SPE APU	**neg**[**o**][.]	√	√	√
evlwwsplat			SPE APU	**nor**[.]	√	√	√
evlwwsplatx			SPE APU	**orc**[.]	√	√	√
evmergehi			SPE APU	**ori**	√	√	√
evmergehilo			SPE APU	**oris**	√	√	√
evmergelo			SPE APU	**or**[.]	√	√	√
evmergelohi			SPE APU	**rfci**	√		√
evmhegsmfaa			SPE APU	**rfi**	√	√	√
evmhegsmfan			SPE APU	**rfid**		√	
evmhegsmiaa			SPE APU	**rfmci**			Machine check
evmhegsmian			SPE APU	**rldcl.**		√	
evmhegumiaa			SPE APU	**rldcr.**		√	
evmhegumian			SPE APU	**rldic.**		√	
evmhesmf			SPE APU	**rldicl.**		√	
evmhesmfa			SPE APU	**rldicr.**		√	
evmhesmfaaw			SPE APU	**rldimi.**		√	
evmhesmfanw			SPE APU	**rlwimi**[.]	√	√	√
evmhesmi			SPE APU	**rlwinm**[.]	√	√	√
evmhesmia			SPE APU	**rlwnm**[.]	√	√	√

Table 3-44. List of Instructions (continued)

Mnemonic	Book E	PowerPC AIM	e500	Mnemonic	Book E	PowerPC AIM	e500
evmhesmiaaw			SPE APU	**sc**	√	√	√
evmhesmianw			SPE APU	**slbia**		√	
evmhessf			SPE APU	**slbie**		√	
evmhessfa			SPE APU	**sldi**		√	
evmhessfaaw			SPE APU	**slw**[.]	√	√	√
evmhessfanw			SPE APU	**srad.**		√	
evmhessiaaw			SPE APU	**sradi.**		√	
evmhessianw			SPE APU	**srawi**[.]	√	√	√
evmheumi			SPE APU	**sraw**[.]	√	√	√
evmheumia			SPE APU	**srd.**		√	
evmheumiaaw			SPE APU	**srw**[.]	√	√	√
evmheumianw			SPE APU	**stb**	√	√	√
evmheusiaaw			SPE APU	**stbu**	√	√	√
evmheusianw			SPE APU	**stbux**	√	√	√
evmhogsmfaa			SPE APU	**stbx**	√	√	√
evmhogsmfan			SPE APU	**std**		√	
evmhogsmiaa			SPE APU	**stdcx.**		√	
evmhogsmian			SPE APU	**stdu**		√	
evmhogumiaa			SPE APU	**stdux**		√	
evmhogumian			SPE APU	**stdx**		√	
evmhosmf			SPE APU	**stfd**	√	√	
evmhosmfa			SPE APU	**stfdu**	√	√	
evmhosmfaaw			SPE APU	**stfdux**	√	√	
evmhosmfanw			SPE APU	**stfdx**	√	√	
evmhosmi			SPE APU	**stfiwx**	√	√	
evmhosmia			SPE APU	**stfs**	√	√	
evmhosmiaaw			SPE APU	**stfsu**	√	√	
evmhosmianw			SPE APU	**stfsux**	√	√	
evmhossf			SPE APU	**stfsx**	√	√	
evmhossfa			SPE APU	**sth**	√	√	√
evmhossfaaw			SPE APU	**sthbrx**	√	√	√
evmhossfanw			SPE APU	**sthu**	√	√	√
evmhossiaaw			SPE APU	**sthux**	√	√	√
evmhossianw			SPE APU	**sthx**	√	√	√
evmhoumi			SPE APU	**stmw**	√	√	√
evmhoumia			SPE APU	**stswi**	√	√	
evmhoumiaaw			SPE APU	**stswx**	√	√	
evmhoumianw			SPE APU	**stw**	√	√	√
evmhousiaaw			SPE APU	**stwbrx**	√	√	√

Table 3-44. List of Instructions (continued)

Mnemonic	Book E	PowerPC AIM	e500	Mnemonic	Book E	PowerPC AIM	e500
evmhousianw			SPE APU	**stwcx.**	√	√	√
evmra			SPE APU	**stwu**	√	√	√
evmwhsmf			SPE APU	**stwux**	√	√	√
evmwhsmfa			SPE APU	**stwx**	√	√	√
evmwhsmi			SPE APU	**subfc**[**o**][.]	√	√	√
evmwhsmia			SPE APU	**subfe**[**o**][.]	√	√	√
evmwhssf			SPE APU	**subfic**	√	√	√
evmwhssfa			SPE APU	**subfme**[**o**][.]	√	√	√
evmwhumi			SPE APU	**subf**[**o**][.]	√	√	√
evmwhumia			SPE APU	**subfze**[**o**][.]	√	√	√
evmwlsmiaaw			SPE APU	**sync**	Replaced with **msync**	√	Replaced with **msync**
evmwlsmianw			SPE APU	**tlbia**		√	
evmwlssiaaw			SPE APU	**tlbie**		√	
evmwlssianw			SPE APU	**tlbivax**	√		√
evmwlumi			SPE APU	**tlbre**	√		√
evmwlumia			SPE APU	**tlbsx**	√		√
evmwlumiaaw			SPE APU	**tlbsync**	√	√	√
evmwlumianw			SPE APU	**tlbwe**	√		√
evmwlusiaaw			SPE APU	**tw**	√	√	√
evmwlusianw			SPE APU	**twi**	√	√	√
evmwsmf			SPE APU	**wrtee**	√		√
evmwsmfa			SPE APU	**wrteei**	√		√
evmwsmfaa			SPE APU	**xori**[.]	√	√	√
evmwsmfan			SPE APU	**xor**[.]	√	√	√
evmwsmi			SPE APU				
evmwsmia			SPE APU				

Chapter 4
Execution Timing

This chapter describes how the e500 core performs operations defined by instructions and how it reports the results of instruction execution. It gives detailed descriptions of 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 gives examples of instruction sequences, showing potential bottlenecks and how to minimize their effects. Finally, 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.

References to e500 apply to both e500v1 and e500v2.

For specific timing guidelines and diagrams, refer to the *e500 Software Optimization Guide*.

4.1 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

Some of these definitions differ slightly from those used to describe previous processors that implement the PowerPC architecture, in particular with respect to dispatch, issue, finishing, retirement, and write back, so please read this glossary carefully.

- Branch prediction—The process of guessing 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 e500 does not use the Book E–defined hint bits in the BO operand for static prediction. Clearing BUCSR[BPEN] disables dynamic branch prediction; in this case the e500 predicts every branch as not taken.
- Branch resolution—The determination of whether a branch prediction is correct. If it is, instructions following the predicted branch that may have been speculatively executed can complete (*see* Completion). If it is incorrect, the processor redirects fetching to the proper path and marks instructions on the mispredicted path (and any of their results) for purging when the mispredicted branch completes.
- Complete—An instruction is eligible to complete after it finishes executing and makes its results available for subsequent instructions. Instructions must complete in order from the bottom two entries of the completion queue (CQ). The completion unit coordinates how

instructions (which may have executed out of order) affect architected registers to ensure the appearance of serial execution. This guarantees that the completed instruction and all previous instructions can cause no exceptions. An instruction completes when it is retired, that is, deleted from the CQ.

- Decode—The decode stage determines the issue queue to which each instruction is dispatched (*see* Dispatch) and determines whether the required space is available in both that issue queue and the completion queue. If space is available, it decodes instructions supplied by the instruction queue, renames any source/target operands, and dispatches them to the appropriate issue queues.
- Dispatch—Dispatch is the event at the end of the decode stage during which instructions are passed to the issue queues and tracking of program order is passed to the completion queue.
- Fetch—The process of bringing instructions from memory (such as a cache or system memory) into the instruction queue.
- Finish—An executed instruction finishes by signaling the completion queue that execution has concluded. An instruction is said to be finished (but not complete) when the execution results have been saved in rename registers and made available to subsequent instructions, but the completion unit has not yet updated the architected registers.
- Issue—The stage responsible for reading source operands from rename registers and register files. This stage also assigns instructions to the proper execution unit.
- Latency— The number of clock cycles necessary to execute an instruction and make the results of that execution available to subsequent instructions.
- Pipeline—In the context of instruction timing, this term refers to interconnected stages. The events necessary to process an instruction are broken into several cycle-length tasks to allow work to be performed on several instructions simultaneously—analogous to an assembly line. As an instruction is processed, it passes from one stage to the next. When work at one stage is done and the instruction passes to the next stage, another instruction can begin work in the vacated stage.

  Although an individual instruction may have multiple-cycle latency, pipelining makes it possible to overlap processing so the number of instructions processed per clock cycle (throughput) is greater than if pipelining were not implemented.
- Program order—The order of instructions in an executing program. More specifically, this term is used to refer to the original order in which program instructions are fetched into the instruction queue from the cache.
- Rename registers—Temporary buffers for holding results of instructions that have finished execution but have not completed. The ability to forward results to rename registers allows subsequent instructions to access the new values before they have been written back to the architectural registers.

- Reservation station—A buffer between the issue and execute stages that allows instructions to be issued even though resources necessary for execution or results of other instructions on which the issued instruction may depend are not yet available.
- Retirement—Removal of a completed instruction from the completion queue at the end of the completion stage. (In other documents, this is often called deallocation.)
- Speculative instruction—Any instruction that is currently behind an older branch instruction that has not been resolved.
- Stage—Used in two different senses, depending on whether the pipeline is being discussed as a physical entity or a sequence of events. As a physical entity, a stage can be viewed as the hardware that handles operations on an instruction in that part of the pipeline. When viewing the pipeline as a sequence of events, a stage is an element in the pipeline during which certain actions are performed, such as decoding the instruction, performing an arithmetic operation, or writing back the results. Typically, the latency of a stage is one processor clock cycle. Some events, such as dispatch, write-back, and completion, happen instantaneously and may be thought to occur at the end of a stage.

  An instruction can spend multiple cycles in one stage; for example, a divide takes multiple cycles in the execute stage.

  An instruction can also be represented in more than one stage simultaneously, especially in the sense that a stage can be seen as a physical resource. For example, when instructions are dispatched, they are assigned a place in the CQ at the same time they are passed to the issue queues.
- Stall—An occurrence when an instruction cannot proceed to the next stage. Such a delay is initiated to resolve a data or resource hazard, that is, a situation in which a planned instruction cannot execute in the proper clock cycle because data or resources needed to process the instruction are not yet available.
- Superscalar—A superscalar processor is one that can issue multiple instructions concurrently from a conventional linear instruction stream. In a superscalar implementation, multiple instructions can execute in parallel at the same time.
- Throughput—The number of instructions processed per cycle. In particular, throughput describes the performance of a multiple-stage pipeline where a sequence of instructions may pass through with a throughput that is much faster than the latency of an individual instruction. For example, in the four-stage multiple-cycle pipeline (MU), a series of **mulli** instructions has a throughput of one instruction per clock cycle even though it takes 4 cycles for one **mulli** instruction to execute.
- Write-back—Write-back (in the context of instruction handling) occurs when a result is written into the architecture-defined registers (typically the GPRs). On the e500, 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.

4.2 Instruction Timing Overview

The e500 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 e500 implements pipelining, superscalar instruction issue, and multiple execution units that operate independently and in parallel.

Figure 4-1 shows the path instructions take through the seven stages (shaded in the figure) of the e500 master pipeline: two fetch stages, decode/dispatch, issue, execute, complete, and write-back stages. The LSU and MU execution units are also multiple-stage pipelines.

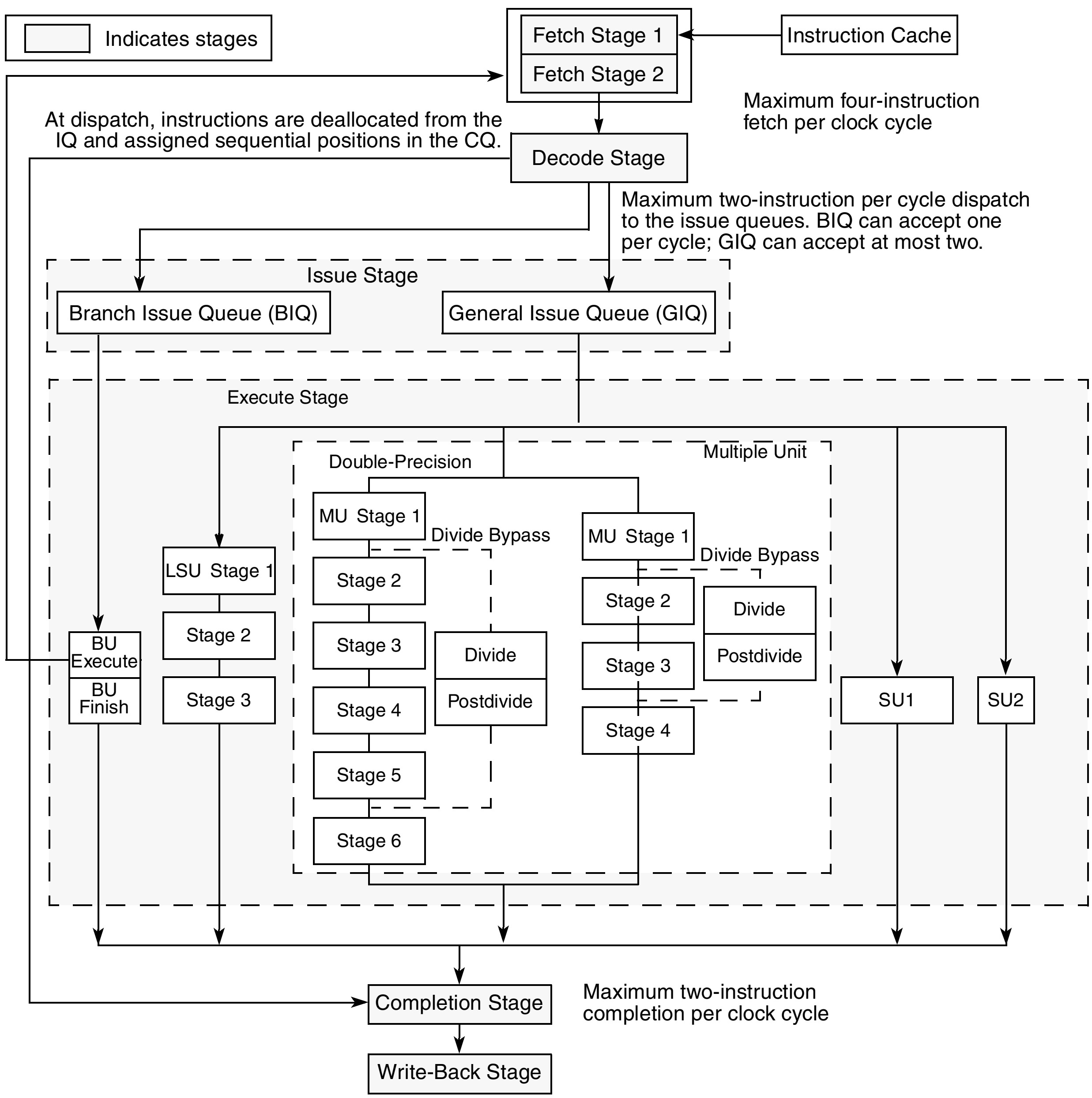


Figure 4-1. Instruction Flow Pipeline Diagram Showing Pipeline Stages

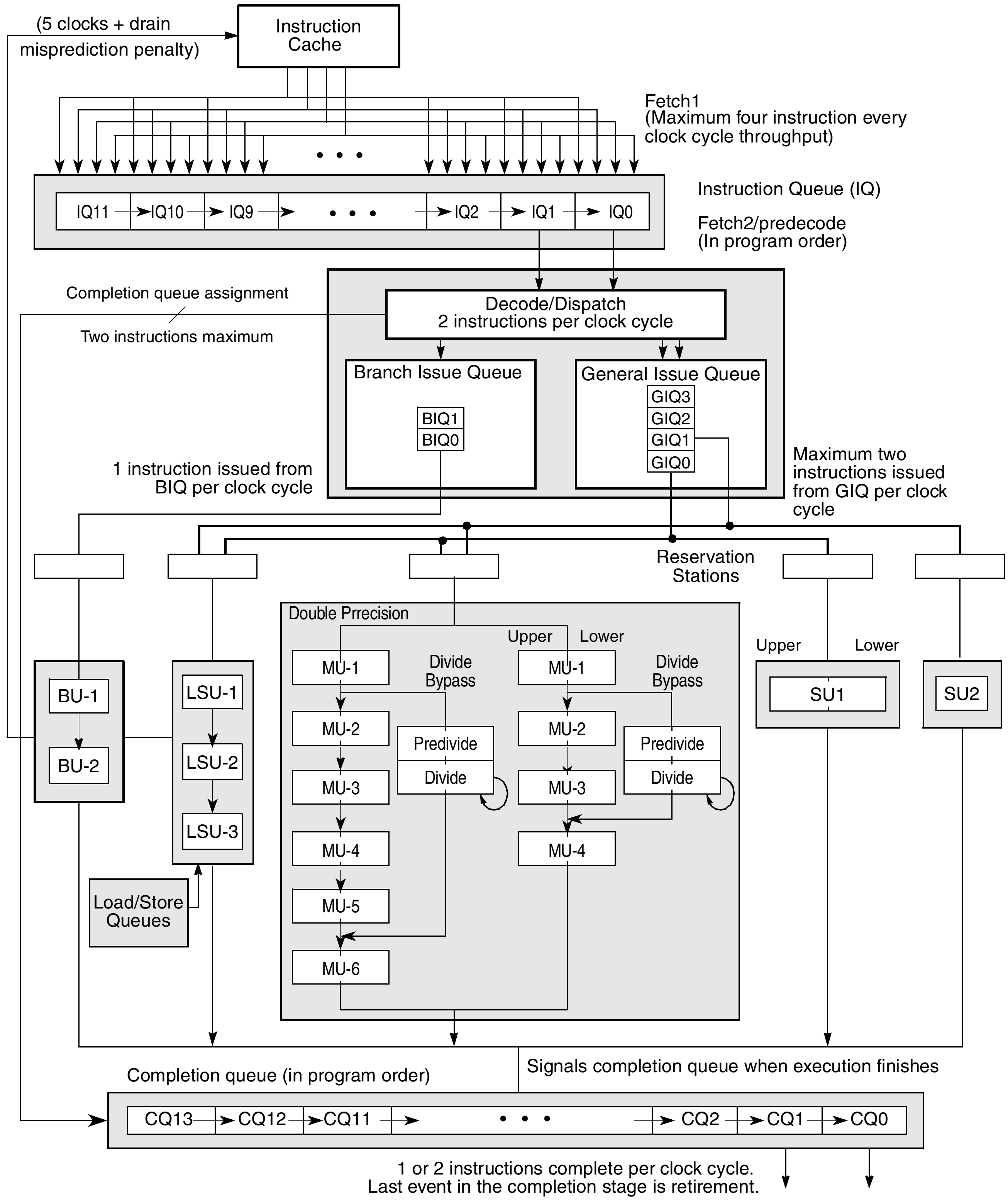


Figure 4-2. e500 Instruction Flow Diagram—Details

The widths of the execution units shown in Figure 4-1 and Figure 4-2 indicate whether a unit can execute instructions with 64-bit operands. LSU, MU, and SU1 have upper and lower halves. Scalar instructions use only the lower halves and update GPR bits 32–63.

Some instructions, such as loads and stores, access memory and require additional clock cycles between the execute and write-back phases. Latencies may be greater if the access is to noncacheable memory, causes a TLB miss, misses in the L1 cache, generates a write-back to memory, causes a snoop hit from another device that generates additional activity, or encounters other conditions that affect memory accesses.

The e500 can complete as many as two instructions on each clock cycle.

The instruction pipeline stages are described as follows:

- Instruction fetch—Includes the clock cycles necessary to request an instruction and the time the memory system takes to respond to the request. Fetched instructions are latched into the instruction queue (IQ) for consideration by the dispatcher.

  The fetcher tries to initiate a fetch in every cycle in which it is guaranteed that the IQ has room for fetched instructions. Instructions are typically fetched from the L1 instruction cache; if caching is disabled, instructions are fetched from the instruction line fill buffer (ILFB), shown in Figure 4-8. Likewise, on a cache miss, as many as four instructions can be forwarded to the fetch unit from the line-fill buffer as the cache line is passed to the instruction cache.

  Fetch timing is affected by many things, such as whether an instruction is in the on-chip instruction cache or an L2 cache (if implemented). Those factors increase when it is necessary to fetch instructions from system memory and include the processor-to-bus clock ratio, the amount of bus traffic, and whether any cache coherency operations are required.

  Fetch timing is also affected by whether effective address translation is available in a TLB, as described in Section 4.3.2.1, "L1 and L2 TLB Access Times."

- The decode/dispatch stage fully decodes each instruction; most instructions are dispatched to the issue queues, but **isync**, **rfi**, **sc**, **nop**s, and others are not. Every dispatched instruction is assigned a GPR rename register and a CR field rename register, even if they do not specify a GPR or CR operand. There is a pair of GPR/CRF rename registers for each CQ entry (even for instructions that do not access the CR or GPRs).

  The two issue queues, BIQ and GIQ, can accept as many as one and two instructions, respectively, in a cycle. Instruction dispatch requires the following:

  — Instructions dispatch only from IQ0 and IQ1.

  — As many as two instructions can be dispatched per clock cycle.

  — Space must be available in the CQ for an instruction to decode and dispatch.

  In this chapter, dispatch is treated as an event at the end of the decode stage. Dispatch dependencies are described in Section 4.7.2, "Dispatch Unit Resource Requirements."

- The issue stage reads source operands from rename registers and register files and determines when instructions are latched into reservation stations.

  The general behavior of the two issue queues is described as follows:

  — The GIQ accepts as many as two instructions from the dispatch unit per cycle. SU1, SU2, MU, and all LSU instructions (including SPE APU loads and stores) are dispatched to the GIQ, shown in Figure 4-3.

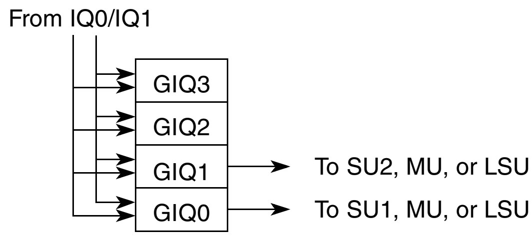


Figure 4-3. GPR Issue Queue (GIQ)

  Instructions can be issued out-of-order from GIQ1–GIQ0. GIQ0 can issue to SU1, MU, and LSU. GIQ1 can issue to SU2, MU, and LSU.

  SU2 executes a subset of the instructions that can be executed in SU1. The ability to identify and dispatch instructions to SU2 increases the availability of SU1 to execute more computationally intensive instructions.

  An instruction in GIQ1 destined for SU2 or the LSU need not wait for an MU instruction in GIQ0 that is stalled behind a long-latency divide.

- The execute stage is comprised of individual non-blocking execution units implemented in parallel. Each execution unit has a reservation station that must be available for an instruction issue to occur. In most cases, instructions are issued both to the reservation station and to the execution unit simultaneously. However, under some circumstances, an instruction may issue only to a reservation station.

  In this stage, operands assigned to the execution stage are latched.

  The e500 has the following execution units:

  — Branch unit (BU)—executes branches and CR logical operations

  — 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, half-word, and word instructions defined by the PowerPC architecture and 64-bit load and store instructions defined as part of the SPE APU. The load/store queues are described in Section 4.4.2.1, "Load/Store Unit Queueing Structures."

  — Two simple units (SU1 and SU2)—execute move to/from SPR instructions, logical instructions, and all computational instructions except multiply and divide instructions. These execution units also execute all vector and scalar computational instructions (except multiply and divide instructions) defined by the SPE and embedded floating-point APUs, as follows:

– SU1 executes 32- and 64-bit SPE and floating-point logical instructions, simple integer arithmetic, and bit manipulation instructions, such as merges and splats.
– SU2 executes a subset of the instructions that can be executed in SU1. These include **brinc** and the embedded floating-point logical instructions, **efsabs**, **efsnabs**, **efsneg**, **efststeq**, **efststgt**, and **efststlt**, and **efdabs**, **efdnabs**, **efdneg**, **efdtsteq**, **efdtstgt**, and **efdtstlt** in the e500v2.

Most SU instructions execute in 1 cycle. Table 4-6 identifies which Book E instructions execute in SU1 and SU2 and shows their latencies; Table 4-8 identifies which SPE and floating-point APU instructions execute in SU1 and SU2 and shows their latencies. Note that most SU instructions execute in 1 cycle, while some instructions (such as **mtspr** and **mfspr**) take longer.

— Multiple-cycle IU (MU) executes integer multiplication and division instructions, and addition, subtraction, multiplication, and division for all vector and scalar instructions.

NOTE

As suggested by Figure 4-1, the MU and SU1 each have upper and lower halves. Both halves are used for SPE and floating-point vector instructions. Only the lower half is used by scalar instructions, including embedded single-precision floating-point instructions.

The execution unit executes the instruction (perhaps over multiple cycles), writes results on its result bus, and notifies the CQ when the instruction finishes. The execution unit reports any exceptions to the completion stage. Instruction-generated exceptions are not taken until the excepting instruction is next to retire.

Most integer instructions have a 1-cycle latency, so results of these instructions are available 1 clock cycle after an instruction enters the execution unit. The LSU and MU are pipelined, as shown in Figure 4-4.

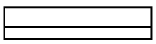
- The complete and write-back stages maintain the correct architectural machine state and commit results to the architecture-defined registers in the proper order. If completion logic detects an instruction containing an exception status or a mispredicted branch, all following instructions are cancelled, their execution results in rename registers are discarded, and the correct instruction stream is fetched.

  The complete stage ends when the instruction is retired. Two instructions can be retired per clock cycle. If no dependencies exist, as many as two instructions are retired in program order. Section 4.7.4, “Completion Unit Resource Requirements” describes completion dependencies.

  The write-back stage occurs in the clock cycle after the instruction is retired.

Conventions used in the instruction timing examples are as follows:

Fetch—Instructions are fetched from memory and placed in the 12-entry IQ. The latency associated with accessing an instruction depends on whether the instruction is in the on-chip cache or system memory (in which case latency is further affected by bus traffic, bus clock speed, and address translation issues). Therefore, in the examples in this chapter, the diagrams and fetch stage shown is for the common case of instructions hitting in the instruction cache.

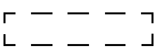


Decode—As many as two eligible instructions dispatch from IQ0–IQ1 to the appropriate issue queue. Note that **isync**, **rfi**, **sc**, and some other instructions do not go to issue queues. At the same time, the instruction is assigned an entry in the completion queue.

Issue—Instructions are dispatched to issue queues from the instruction queue entries. At the end of the issue stage, instructions and their operands, if available, are latched into execution unit reservation stations. The black stripe is a reminder that the instruction occupies an entry in the CQ, described in Figure 4-4.

Execute—The operations specified by an instruction are being performed by the appropriate execution unit. The black stripe is a reminder that the instruction occupies an entry in the CQ, described in Figure 4-4.

Complete—Execution has finished. When all completion requirements are met, the instruction is retired from the CQ. The results are written back to architecture-defined registers in the clock cycle after retirement.



Write back—The instruction has retired and its results are written back to the architecture-defined registers.

Figure 4-4 shows the relationships between stages and events associated with them.

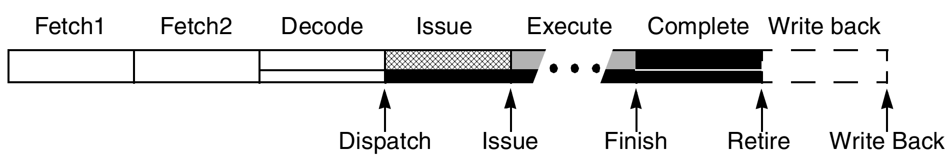


Figure 4-4. Execution Pipeline Stages and Events

The events are described as follows:

- Dispatch (at the end of decode)—An instruction is dispatched to the appropriate issue queue at the end of the decode stage. At dispatch, the instruction passes to the issue pipeline stage by taking a place in the CQ and in one of the two issue queues.
- Issue (at the end of the issue stage)—The issue stage ends when the instruction is issued to the appropriate execution unit.

- Finish (at the end of the execute stage)—An instruction finishes when the CQ is signaled that execution results are available to subsequent instructions. Architecture-defined registers are not updated until the instruction is retired.
- Retire (at the end of the complete stage)—An instruction retires from the CQ after execution is finished and serializing conditions are met.
- Write back (at the end of the write-back stage)—The results of a retired instruction are written back to the architecture-defined register.

Figure 4-5 shows the stages of e500 execution units.

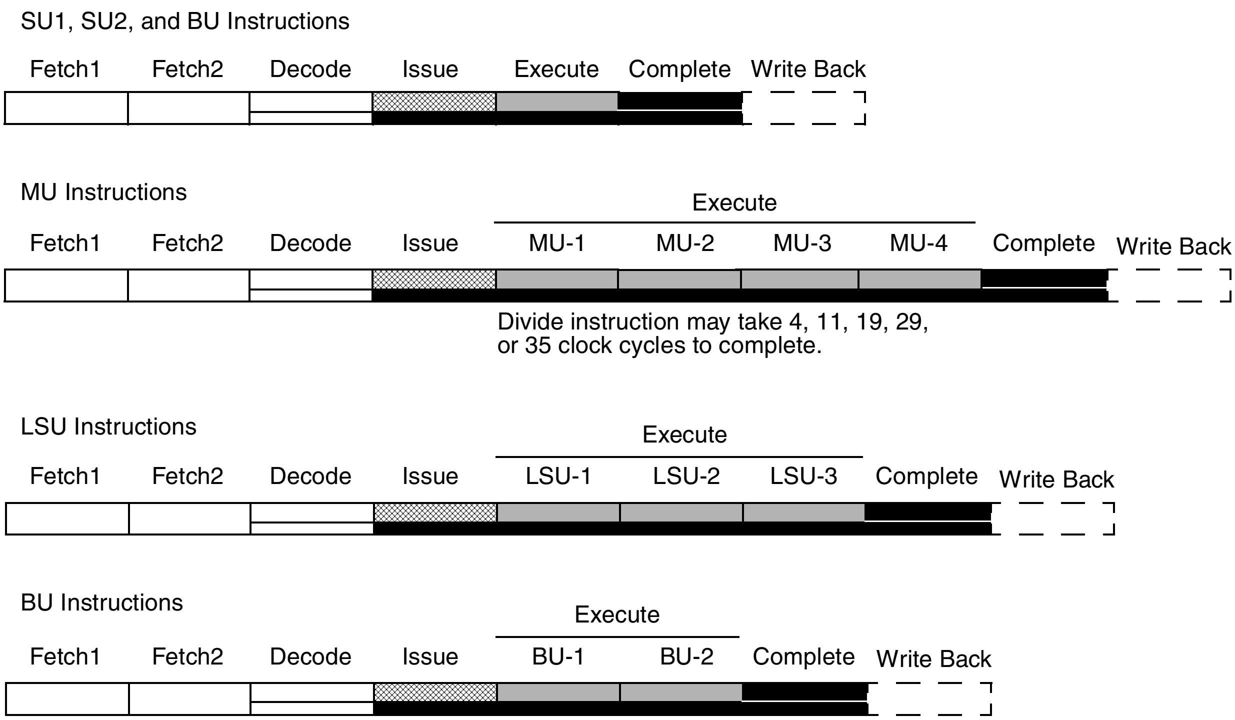


Figure 4-5. Execution Stages

4.3 General Timing Considerations

As many as four instructions can be fetched to the IQ during each clock cycle. Two instructions per clock cycle can be dispatched to the issue queues. Two instructions from the GIQ and one instruction from the BIQ can issue per clock cycle to the appropriate execution units. Two instructions can retire and two can write back per cycle.

The e500 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 or CRF) or from a rename register.

Branch prediction is performed in parallel with the fetch stages using the branch prediction unit (BPU), which incorporates the branch target buffer (BTB). Predictions are resolved in the branch unit (BU). Incorrect predictions are handled as follows:

1. Fetch is redirected to the correct path, and mispredicted instructions are purged.
2. The mispredicted branch is marked as such in the CQ.
3. Eventually, the branch is retired and the CQ, issue queue, and execution units are flushed. If the correct-path instructions reach the IQ before the back half of the pipeline is flushed, they stall in the IQ until the flush occurs.

After an instruction executes, results are made available to subsequent instructions in the appropriate rename registers. The architecture-defined GPRs are updated in the write-back stage. Branch instructions that update LR or CTR write back in a similar fashion.

If a later instruction needs the result as a source operand, the result is simultaneously made available to the appropriate execution unit, which allows a data-dependent instruction to be decoded and dispatched without waiting to read the data from the architected register file. Results are then stored into the correct architected GPR during the write-back stage. Branch instructions that update either the LR or CTR write back their results in a similar fashion.

Section 4.3.1, “General Instruction Flow,” describes this process.

4.3.1 General Instruction Flow

To resolve branch instructions and improve the accuracy of branch predictions, the e500 implements a dynamic branch prediction mechanism using the 512-entry BTB, a four-way set associative cache of branch target effective addresses. A BTB entry is allocated whenever a branch resolves as taken—unallocated branches are always predicted as not taken. Each BTB entry holds a 2-bit saturating branch history counter whose value is incremented or decremented depending on whether the branch was taken. These bits can take four values: strongly taken, weakly taken, weakly not taken, and strongly not taken. This mechanism is described in Section 4.4.1.2, “BTB Branch Prediction and Resolution.”

The e500 does not implement the static branch prediction that is defined by the PowerPC architecture. The BO[*y*] prediction in branch encodings is ignored.

Dynamic branch prediction is enabled by setting BUCSR[BPEN]. Clearing BUCSR[BPEN] disables dynamic branch prediction, in which case the e500 predicts every branch as not taken.

Branch instructions are treated like any other instruction and are assigned CQ entries to ensure that the CTR and LR are updated sequentially.

The dispatch rate is affected by the serializing behavior of some instructions and the availability of issue queues and CQ entries. Instructions are dispatched in program order; an instruction in IQ1 cannot be dispatched ahead of one in IQ0.

4.3.2 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 4.3.2.1, “L1 and L2 TLB Access Times.”
- If a page translation is not in a TLB, an instruction TLB miss interrupt is taken. Section 4.3.2.2, “Interrupts Associated with Instruction Fetching,” describes other conditions that cause an instruction fetch to take an interrupt. General interrupt latency and pipeline behavior are described in Section 4.3.4, “Interrupt Latency.”
- If an L1 instruction 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 4.3.2.3, “Cache-Related Latency.”

4.3.2.1 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 5 clocks; for L1 data address translation misses, the L2 TLB latency is at least 6 clocks. These access times may be longer, depending on arbitration performed by the L2 arrays for simultaneous instruction L1 TLB misses, data L1 TLB misses, the execution of TLB instructions, and TLB snoop operations (snooping of TLBINV operations on the CCB).

Note that when a TLBINV operation is detected, the L2 MMU arrays become inaccessible due to the snooping activity caused by the TLBINV.

If the MMU is busy due to a higher priority operation, such as a **tlbivax**, instructions cannot be fetched until that operation completes.

If the page translation is in neither TLB, an instruction TLB error interrupt occurs, as described in Section 5.7.13, “Instruction TLB Error Interrupt.”

TLBs are described in detail in Chapter 12, “Memory Management Units.”

4.3.2.2 Interrupts Associated with Instruction Fetching

An instruction fetch can generate the following interrupts:

- An instruction TLB error interrupt occurs when the effective address translation for a fetch is not found in the TLBs. This interrupt is described in detail in Section 5.7.13, “Instruction TLB Error Interrupt.”

- 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).
    - – 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).
  - — A byte ordering exception occurs when the implementation cannot fetch the instruction in the byte order specified by the page's endian attribute. On the e500, accesses that cross a page boundary such that endianness changes causes a byte ordering exception.

  When an instruction storage interrupt occurs, the processor suppresses execution of the instruction causing the exception. For more information, see Section 5.7.4, "Instruction Storage Interrupt."

4.3.2.3 Cache-Related Latency

The following may happen when instructions are fetched from the instruction cache,:

- If the fetch hits the cache, it takes 2 clock cycles after the request for as many as four instructions to enter the IQ. The cache is not blocked to internal accesses during a cache reload (hits under misses).

  The cache allows a hit under one miss and is only blocked by a cache line reload for the cycle during the cache write. For example, if a cache miss is discarded by a misprediction and a new fetch hits, the cache allows instructions to come back. As many as four instructions per cycle are fetched from the cache until the original miss comes back and a cache reload is performed, which blocks the cache for 1 cycle.

  If the cache is busy due to a higher priority operation, such as an **icbi** or a cache line reload, instructions cannot be fetched until that operation completes.
- If an instruction fetch misses the on-chip instruction cache, the e500 initiates a core complex bus transaction to the non-core memory system.

To minimize the effect of bus contention, the Book E architecture defines WIM bits that define caching characteristics for the corresponding page. Accesses to caching-inhibited memory locations never update the L1 caches.

If a cache-inhibited access hits in the cache, the cache block is invalidated. If the cache block is marked modified, it is copied back to memory before being invalidated. Where caching is permitted, memory is configured as either write-back or write-through, as described in Section 11.3.4, "WIMGE Settings and Effect on L1 Caches."

4.3.3 Dispatch, Issue, and Completion Considerations

The core's ability to dispatch as many as two instructions per cycle depends on the mix of instructions and on the availability of issue queues and CQ entries. As many as two instructions can be dispatched in parallel, but an instruction in IQ1 cannot be dispatched ahead of an instruction in IQ0.

Instructions can issue out of order from GIQ0 and GIQ1. GIQ0 can issue to SU1, MU, and LSU. GIQ1 can issue to SU2, MU, and LSU. If an instruction stalls in GIQ0 (reservation station busy), an instruction in GIQ1 can issue if its reservation station is available.

Issue queues and reservation stations allow the e500 to dispatch instructions even if execution units are busy. The issue logic reads operands from register files and rename registers and routes instructions to the proper execution unit. Execution begins when all operands are available, the instruction is in the reservation station, and any execution serialization requirements are met.

Instructions pass through a single-entry reservation station associated with each execution unit. If a data dependency keeps an instruction from starting execution, that instruction is held in a reservation station. Execution begins during the same clock cycle that the rename register is updated with the data the instruction is dependent on.

The CQ maintains program order after instructions are dispatched, guaranteeing in-order completion and a precise exception model. Instruction state and other information required for completion are kept in this 14-entry FIFO. All instructions complete in order; none can retire ahead of a previous instruction. In-order completion ensures the correct architectural state when the e500 must recover from a mispredicted branch or exception.

Instructions are retired much as they are dispatched: as many as two can be retired simultaneously, but never out of order. Note the following:

- Instructions must be non-speculative to complete.
- As many as two rename registers can be updated per clock cycle. Because load and store with update instructions require two rename registers they are broken into two instructions at dispatch (**lwzu** is broken into **lwz** and **addi**). As described in Section 4.3.3.1, "GPR and CR Rename Register Operation," these two instructions are assigned two CQ entries and each is assigned CR and GPR renames at dispatch.
- Some instructions have retirement restrictions, such as retiring only out of CQ0. See Section 4.3.3.3, "Instruction Serialization."

Program-related exceptions are signaled when the instruction causing the exception reaches CQ0. Previous instructions are allowed to complete before the exception is taken, which ensures that any exceptions those instructions may cause are taken.

4.3.3.1 GPR and CR Rename Register Operation

To avoid contention for a given register file location during out-of-order execution, the e500 provides 14 rename registers for holding instruction results before the completion commits them to the architecture-defined registers. In addition to the 14 GPR renames, the e500 provides fourteen 4-bit CR field renames. Because there are 14 rename pairs and 14 CQ entries, the e500 cannot run out of renames as long as CQ entries are available.

Results from rename registers are transferred to the architecture-defined registers in the write-back stage, at which point renames are deallocated.

If branch prediction is incorrect, instructions after the branch are flushed from the CQ. Any results of those instructions are flushed from the rename registers.

4.3.3.2 LR and CTR Shadow (Speculative) Registers

The decode stage manages one speculative copy each of the LR and of the CTR. This allows one-level-deep speculation for branch-to-LR and branch-to-CTR instructions.

4.3.3.3 Instruction Serialization

Although the e500 core can dispatch and complete two instructions per cycle, some serializing instructions limit dispatch and completion to one per cycle. There are six basic types of instruction serialization:

- Presync serialization—Presync-serialized instructions are held in the instruction queue until all prior instructions have completed. They are then decoded and execute. For example, instructions such as **mfspr** that read a non-renamed status register are marked as presync-serialized.
- Postsync serialization—Postsync-serialized instructions, such as **mtspr**[XER], prevent other instructions from decoding until the serialized instruction completes. For example, instructions that modify processor state in a way that affects the handling of future instruction execution are marked with postsync-serialization. These instructions are identified in the latency tables in Section 4.6, “Instruction Latency Summary.”
- Move-from serialization—Move-from serialization is a weaker synchronization than presync serialization. A move-from serialized instruction can decode, but stalls in an execution unit’s reservation station until all prior instructions have completed. If the instruction is currently in the reservation station and is the oldest instruction, it can begin execution in the next cycle. Note that subsequent instructions can decode and execute while a move-from serialized instruction is pending. Only **mfcr** and **mfspr**[XER] are move-from serialized, so that they do not examine architectural state until all older instructions that could affect the architectural state have completed.

- Move-to serialization—A move-to serialized instruction cannot execute until the cycle after it is in CQ0, that is, the cycle after it becomes the oldest instruction. This serialization is weaker than move-from serialization in that the instruction need not spend an extra cycle in the reservation station. Move-to serializing instructions include **tlbre**, **tlbsx**, **tlbwe**, **mtmsr, wrtee**, **wrteei**, and all **mtspr** instructions.
- Refetch serialization—Refetch-serialized instructions force refetching of subsequent instructions after completion. Refetch serialization is used when an instruction has changed or may change a particular context needed by subsequent instructions. Examples include **isync**, **sc**, **rfi**, **rfci**, **rfmci**, and any instruction that toggles the summary-overflow (SO) bit.
- Store serialization (applicable to stores and some LSU instructions that access the data cache)—Store-serialized instructions are dispatched and held in the LSU's finished store queue. They are not committed to memory until all prior instructions have completed. Although a store-serialized instruction waits in the finished store queue, other load/store instructions can be freely executed. Some store-serialized instructions are further restricted to complete only from CQ0. Only one store-serialized instruction can complete per cycle, although non-serialized instructions can complete in the same cycle as a store-serialized instruction. In general, all stores and cache operation instructions are store serialized.

4.3.4 Interrupt Latency

The e500v1 flushes all instructions in the completion queue when an interrupt is taken, except for guarded load or cache-inhibited **stwcx.** instructions in CQ0.

Core complex interrupt latency (the number of core clocks between the sampling of the interrupt signal as asserted and the fetch of the first instruction in the handler) is at most 8 cycles unless a guarded load or a cache-inhibited **stwcx.** is in CQ0. This latency does not include the 2 bus cycles needed to synchronize the interrupt signal from the pad of the device. When an interrupt is detected, only guarded load and cache-inhibited **stwcx.** instructions in CQ0 are allowed to complete; in such cases, interrupt latency is affected by bus latency.

Note that a load instruction that misses in the cache may generate a bus read operation, even though the load instruction does not complete because of an interrupt. In this case, data is returned to the line fill buffer and the cache line is updated, but not the GPR specified by the load instruction. When the same load is executed again, the load is performed again, most likely from the cache or from the line fill buffer, and the GPR write back occurs after the instruction completes and is deallocated from CQ0.

On the e500v2, if an interrupt is asserted during a guarded load (that misses in the L1 cache) or a caching-inhibited **stwcx.**, the interrupt is not taken until the instruction completes. So, the interrupt latency depends on the memory latency.

- For guarded loads, the data must be returned. If a bus error occurs on a guarded load, the load is aborted and the interrupt is taken.

- For a caching-inhibited **stwcx.** instructions, the address tenure must complete on the CCB. If a bus error occurs, the **stwcx.** completes and clears CR0[EQ], indicating that the **stwcx.** did not succeed.

Guarded **lmw** and **stmw** instructions can be interrupted before the instruction completes and restarted after the interrupt is serviced.

4.3.5 Memory Synchronization Timing Considerations

This section describes the behavior of the **msync** and **mbar** instructions as they are implemented by the e500.

4.3.5.1 msync Instruction Timing Considerations

The **msync** instruction provides a memory barrier throughout the memory hierarchy. It may be used, for example, to ensure that a control bit has finally been written to its destination control register in the system before the next instruction begins execution (such as to clear a pending interrupt). By its nature, it also provides an ordering boundary for pre- and post-**msync** storage transactions.

On the e500, **msync** waits for preceding data memory accesses to reach the point of coherency (that is, visible to the entire memory hierarchy), then it is broadcast on the e500 bus. An **msync** does not finish execution until all storage transactions caused by prior instructions complete entirely in its caches and externally on the bus (address and data complete on the bus, excluding instruction fetches). No subsequent instructions and associated storage transactions are initiated until such completion.

It completes only after its successful address bus tenure (without being ARTRYed). Execution of **msync** also generates a SYNC command on the bus (if HID1[ABE] is set), which also must complete normally (without address retry) for the **msync** instruction to complete. Subsequent instructions can execute out of order, but they can complete only after **msync** completes.

It is the responsibility of the system to guarantee the intention of the SYNC command on the bus—usually by ensuring that any bus transactions received before the SYNC command from the core complex complete in its queues or at their destinations before completing the SYNC command on the CCB.

4.3.5.2 mbar Instruction Timing Considerations

The **mbar** instruction provides an ordering boundary for storage operations. Its architectural intent is to guarantee that storage operations resulting from previous instructions occur before any subsequent storage operations occur, thereby ensuring an order between pre- and post-**mbar** memory operations. It may be used, for example, to ensure that reads and writes to an I/O device or between I/O devices occur in program order or to ensure that memory updates occur before a semaphore is released.

The Book E architecture allows an implementation to support several classes of storage ordering, selected by the MO field of the **mbar** instruction. The core complex supports two classes for system flexibility.

The e500 implements two variations of **mbar**, as follows:

- When MO = 0, **mbar** behaves as defined by Book E.
- When MO = 1, **mbar** is a weaker, faster memory barrier; the e500 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, bus, and system). On the bus, this ordering barrier is issued as an ORDER command (if HID1[ABE] is set).

  mbar ensures that all data accesses caused by previous instructions complete before any caused by subsequent instructions. This order is seen by all mechanisms. However, unlike **msync** and **mbar** with MO = 0, subsequent instructions can complete without waiting for **mbar** to perform its address bus tenure. This provides a faster way to order data accesses.

4.4 Execution

The following sections describe instruction execution behavior within each of the respective execution units in the e500.

4.4.1 Branch Unit Execution

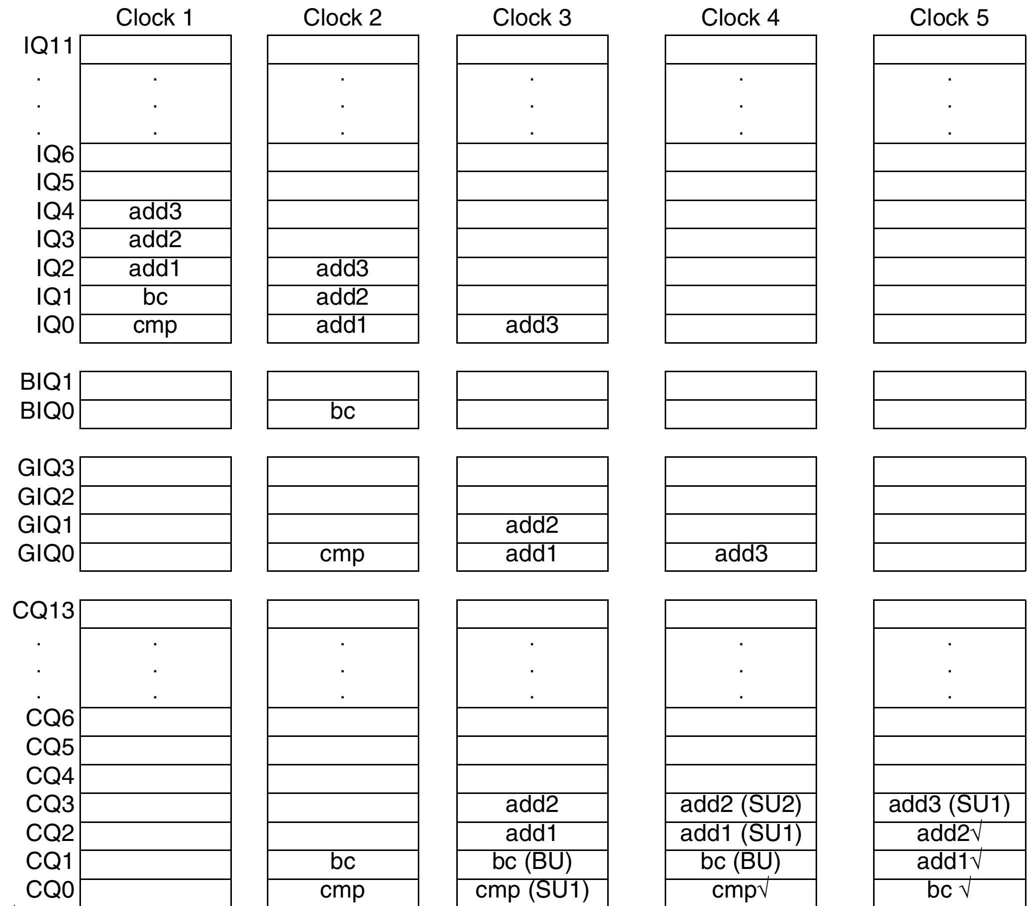
When branch or trap instructions change program flow, the IQ must be reloaded with the target instruction stream. Previously issued instructions continue executing while the new instruction stream makes its way into the IQ. Depending on whether target instructions are cached, opportunities may be missed to execute instructions.

The e500 minimizes penalties associated with flow control operations by features such as the branch target buffer (BTB), BTB locking, dynamic branch prediction, speculative link and counter registers, and nonblocking caches.

4.4.1.1 Branch Instructions and Completion

Branch instructions are not folded on the e500; all branch instructions receive a CQ entry (and CRF and GPR renames) at dispatch and must write back in program order.

Branch instructions are dispatched to the BIQ and are assigned a CQ slot, as shown in Figure 4-6.



√ indicates that the instruction has finished execution.

Figure 4-6. Branch Completion (LR/CTR Write-Back)

In this example, the **bc** depends on **cmp** and is predicted as not taken. At the end of clock cycle 1, **cmp** and **bc** are dispatched to the GIQ and BIQ, respectively, and are issued to SU1 and the BU at the end of clock 2.

In clock cycle 3, the **cmp** executes in SU1 but the **bc** cannot resolve and complete until the **cmp** results are available; add1 and add2 are dispatched to the GIQ.

In cycle 4, the **bc** resolves as correctly predicted; add1 and add2 are issued to the SUs and are marked as nonspeculative, and add3 is dispatched to the GIQ. The **cmp** is retired from the CQ at the end of cycle 4.

In cycle 5, **bc**, add1, and add2 finish execution, and **bc** and add1 retire.

4.4.1.2 BTB Branch Prediction and Resolution

The e500 dynamic branch prediction mechanism differs from its predecessors in that branches are detected and predicted earlier, in the two fetch stages. This processor-specific hardware mechanism monitors and records branch instruction behavior, from which the next occurrence of the branch instruction is predicted.

The e500 does not support static branch prediction—the BO prediction in branch instructions is ignored.

The valid bit in each BTB entry is zero (invalid) at reset. When a branch instruction first enters the instruction pipeline, it is not allocated in the BTB and so by default is predicted as not taken. If the branch is not taken, nothing is allocated in the BTB. If it is taken, the misprediction allocates a BTB entry for this branch with an initial prediction of strongly taken, as is shown in the example in Table 4-6.

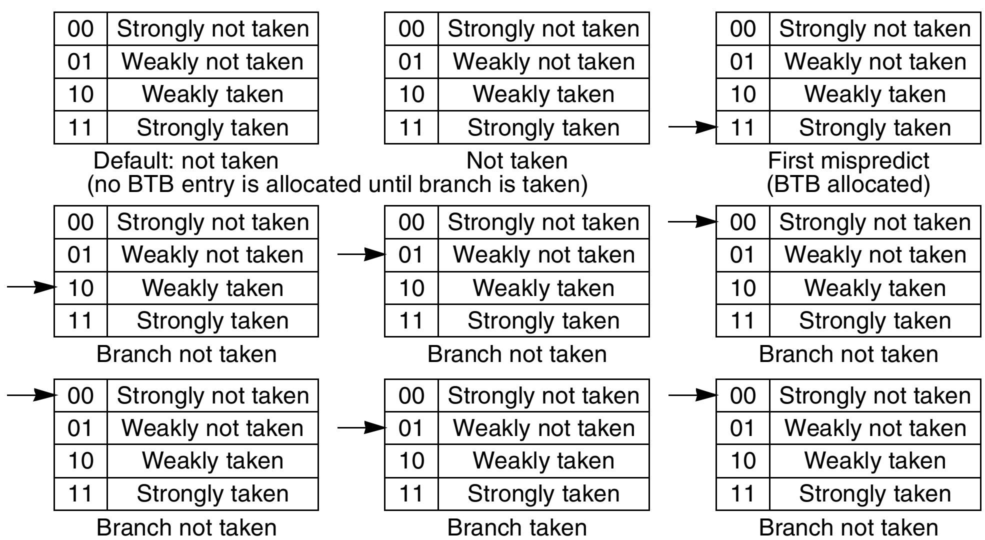


Figure 4-7. Updating Branch History

Note that unconditional branches are allocated in the BTB the first time they are encountered. This example shows how the prediction is updated depending on whether a branch is taken.

The BPU detects whether a fetch group includes any branches that hit in the BTB, and if so, determines the fetching path based on the prediction and the target address.

If the prediction is wrong, subsequent instructions and their results are purged. Instructions ahead of the predicted branch proceed normally, instruction fetching resumes along the correct path, and the history bits are revised.

The number of speculative branches that have not yet been allocated (and are predicted as not taken) is limited only by the space available in the pipeline (the branch execute unit, the BIQ, and the IQ). The presence of speculative branches allocated in the BTB slightly reduces speculation depth.

Instructions after an unresolved branch can execute speculatively, but in-order completion ensures that mispredicted speculative instructions do not complete. When misprediction occurs, the e500 easily redirects fetching and repairs its machine state because the architectural state is not updated. Any instructions dispatched after a mispredicted branch instruction are flushed from the CQ, and any results are flushed from the rename registers.

4.4.1.3 BTB Operations

Understanding how the BTB is indexed requires a discussion of the fetch mechanism. The e500 tries to fetch as many as four instructions per access. Simultaneously fetched instructions comprise a fetch group; and the address issued by the fetch unit is called a fetch group address (FGA).

A fetch group cannot straddle a cache-line boundary. As shown in Figure 4-8, if instructions in a cache line are numbered 0–7 and the fetch group address maps to the n^{th} instruction, where n = 0, 1, 2, 3, or 4, instructions n, n+1, n+2, n+3 are in the fetch group. If n ≥ 4, instructions n through 7 are the fetch group.

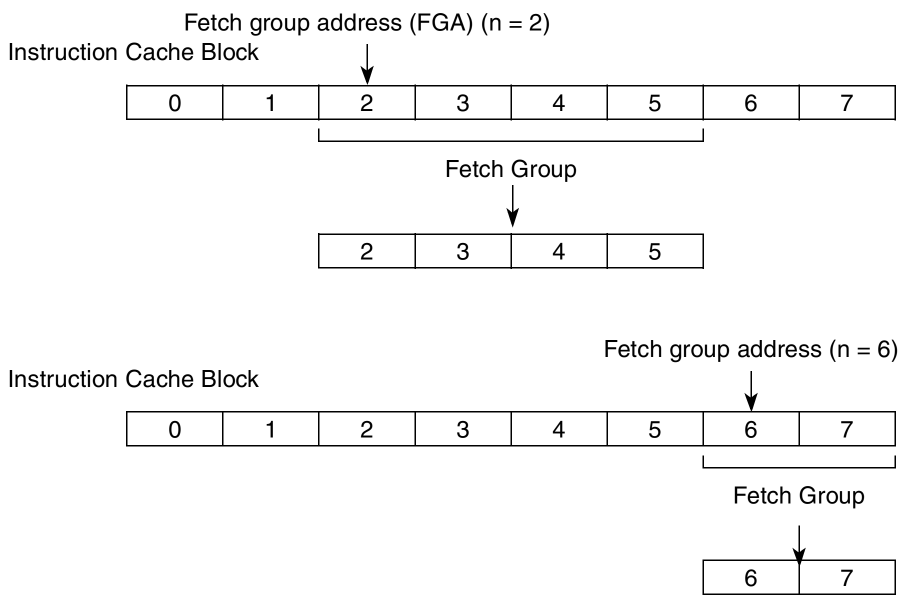


Figure 4-8. Fetch Groups and Cache Line Alignment

If the cache is disabled, instructions are loaded eight instructions at a time and placed in the eight-entry instruction line fill buffer (ILFB), from which a fetch group is delivered to the core following the pattern described in Figure 4-8.

To reduce the size and complexity of the branch predictor, the e500 indexes the BTB using the FGA to identify the first predicted branch within the fetch group. Because the same branch can be fetched at different times as a part of a different fetch group, the BTB locking APU can be used to lock all possible addresses whose fetch groups may contain the branch instruction.

The following factors affect the FGA of a branch instruction:

- The location of the branch instruction in the cache block
- A control flow that may allow multiple execution paths to reach the branch from different fetch group addresses
- The presence of other branch instructions in the fetch group that precede the branch instruction under consideration
- Interrupts taken as a result of accepting an external interrupt or exceptions in instructions preceding the branch instruction in the fetch group
- Events inside the core causing a synchronization in the pipeline during the execution of an instruction preceding the branch instruction in the fetch group
- The presence of instructions such as **isync** before the branch instruction

Figure 4-9 shows all possible fetch group addresses (FGAs) that can be associated with a branch instruction. The location of an instruction is *i* if it is the i^{th} instruction (i=0…7) from the beginning of a cache line. The address of an instruction a_i refers to the address of the i^{th} instruction in the cache block. The condition IB occurs where either a synchronizing instruction (such as **isync**) or a branch instruction whose prediction is locked in the BTB occurs at some location. The branch instruction under consideration is identified as b.

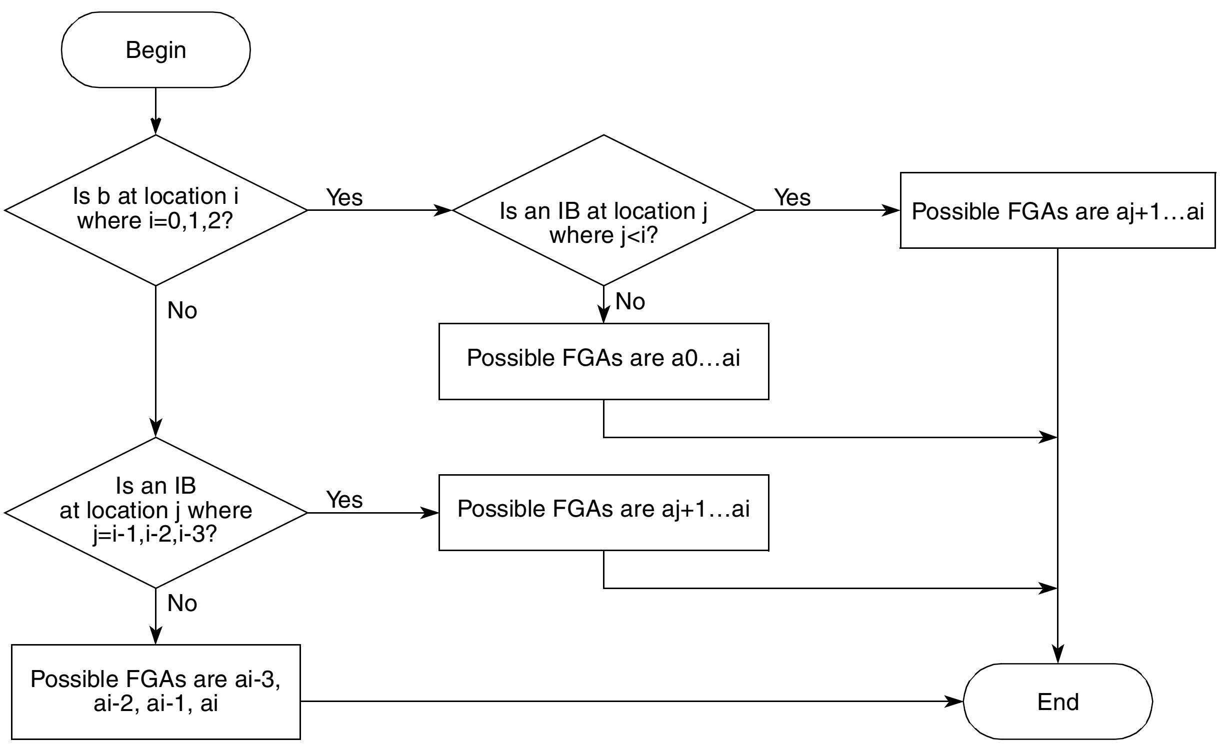


Figure 4-9. Fetch Group Addresses

Note that branch instructions that are not allocated into the BTB (either because they have never been taken or because they have been cast out of the BTB) can fall in the same fetch group. For example, the following code sequence has two branch instructions that fall into the same fetch group the first time the sequence is executed:

```
A:      add
        b1
A+8:    add
        b2
```

Assuming that fetching begins at A and that the sequence lies within a cache block, all four instructions are included in the same fetch group, including both branches because they have not been taken and therefore do not have BTB entries.

At execution, b1 is not taken, but b2 mispredicts and resolves as taken in the execute stage, at which point the branch instruction prediction (strongly taken) is allocated for b2 at A for the fetch group address of A.

Later, b1 is taken and thus mispredicted. The BTB entry for address A becomes allocated for b1, replacing the prediction for b2 for the FGA of A. If we fetch b2 again using the FGA of (A+8), it is now a BTB miss and the default prediction is used. However, if the default prediction is incorrect, a separate BTB entry is allocated for b2 (at fetch group address A+8).

Now that both branch instructions are allocated in the BTB, they can no longer be in the same fetch group.

4.4.1.3.1 BTB Locking

Note that rather than allowing branch predictions to change dynamically, the programmer can explicitly lock the predictions into the BTB.

The typical sequence of instructions to lock a branch address into a BTB entry is as follows:

```
mtspr BBEAR, rS
mtspr BBTAR, rS
bblels
```

The typical sequence of instructions to clear locked entries individually is as follows:

```
mtspr BBEAR, rS
bbelr
```

To guarantee atomicity, these instruction sequences should be protected by **lwarx** and **stwcx.** instructions.

4.4.1.3.2 BTB Locking APU Programming Model

The BTB APU programming model includes the following register resources:

- The following BTB locking APU registers.
  - — Branch buffer entry address register (BBEAR)
  - — Branch buffer target address register (BBTAR)
  - — Branch unit control and status register (BUCSR)

  These registers are described in Section 2.9, "Branch Target Buffer (BTB) Registers."
- MSR[UBLE]. The user branch locking enable bit (UBLE) is defined in the MSR. Setting MSR[UBLE] allows user mode programs to lock or unlock BTB entries. See Section 2.5.1, "Machine State Register (MSR)."

The BTB also defines the following instructions, described in Section 3.9.1, "Branch Target Buffer (BTB) Locking Instructions":

- Branch Buffer Load Entry and Lock Set (**bblels**)
- Branch Buffer Entry Lock Reset (**bbelr**)

4.4.1.3.3 BTB Operations Controlled by BUCSR

This following BTB operations are controlled through BUCSR:

- BTB disabling. BUCSR[BPEN] is used to enable or disable the BTB. The BTB is enabled when the bit is set and disabled when it is cleared.When it is disabled, BTB contents are not used to predict the branch targets and the BTB is not updated as a result of executing branch, **bblels**, or **bbelr** instructions. However, when it is disabled, the BTB maintains its contents and any locks, which can be used again when the BTB is reenabled.
- BTB overlocking. BUCSR[BBLO] is used to report an overlocking status to the program. It is a sticky bit and once set, remains set until explicitly cleared by writing a 0 to it with an **mtspr** instruction.
- BTB unable to lock. If **bblels** cannot set the BTB lock, BUCSR[BBUL] is set. It is a sticky bit.
- BTB invalidation. Flash invalidation of the BTB is accomplished by writing BUCSR[BBFI] with a 0 and then a 1 using **mtspr** instructions.
- BTB lock clearing. BUCSR[BBLFC] is used to perform a flash lock clear (unlocking) of all locked BTB entries. Writing BUCSR[BBLFC] with a 0 and then a 1 flash lock clears all locked BTB entries.

4.4.1.3.4 BTB Special Cases—Phantom Branches and Multiple Matches

The following describes special cases:

- Phantom branches. BTB entries hold effective addresses associated with a branch instruction. A process context switch might bring in another task whose MMU translations are such that it uses the same effective address for another non-branch instruction for which the BTB has an entry for a previously encountered branch. This causes the fetch unit to redirect instruction fetch to the BTB’s target address. Later, during execution of the instruction, the hardware realizes the error and evicts the BTB entry. However, locked BTB entries are not evicted. Hardware guarantees correct execution under locked phantom branches, but performance may suffer.
- Multiple matches. By ensuring that an entry is unique when it is allocated, the e500 hardware prevents multiple matches for the same fetch address.

4.4.2 Load/Store Unit Execution

The data cache supplies data to the GPRs by means of the LSU. The core complex LSU is directly coupled to the data cache with a 64-bit (8-byte) interface to allow efficient movement of data to and from the GPRs. The LSU provides all of the logic required to calculate effective addresses, handles data alignment to and from the data cache, provides sequencing for load/store multiple operations, and interfaces with the core interface unit. Write operations to the data cache can be performed on a byte, half-word, word, or double-word basis.

When free of data dependencies, cacheable loads execute in the LSU in a speculative manner with a maximum throughput of one 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.

4.4.2.1 Load/Store Unit Queueing Structures

This section describes the LSU queues that support the L1 data cache. See Section 11.3.5, “Load/Store Operations,” for more information on architectural coherency implications of load/store operations and the LSU on the core complex. Also, see Section 4.4.4, “Load/Store Execution,” for more information on other aspects of the LSU and instruction scheduling considerations.

The instruction and data caches are integrated with the LSU, instruction unit, and core interface unit in the memory subsystem of the core complex as shown in Figure 4-10.

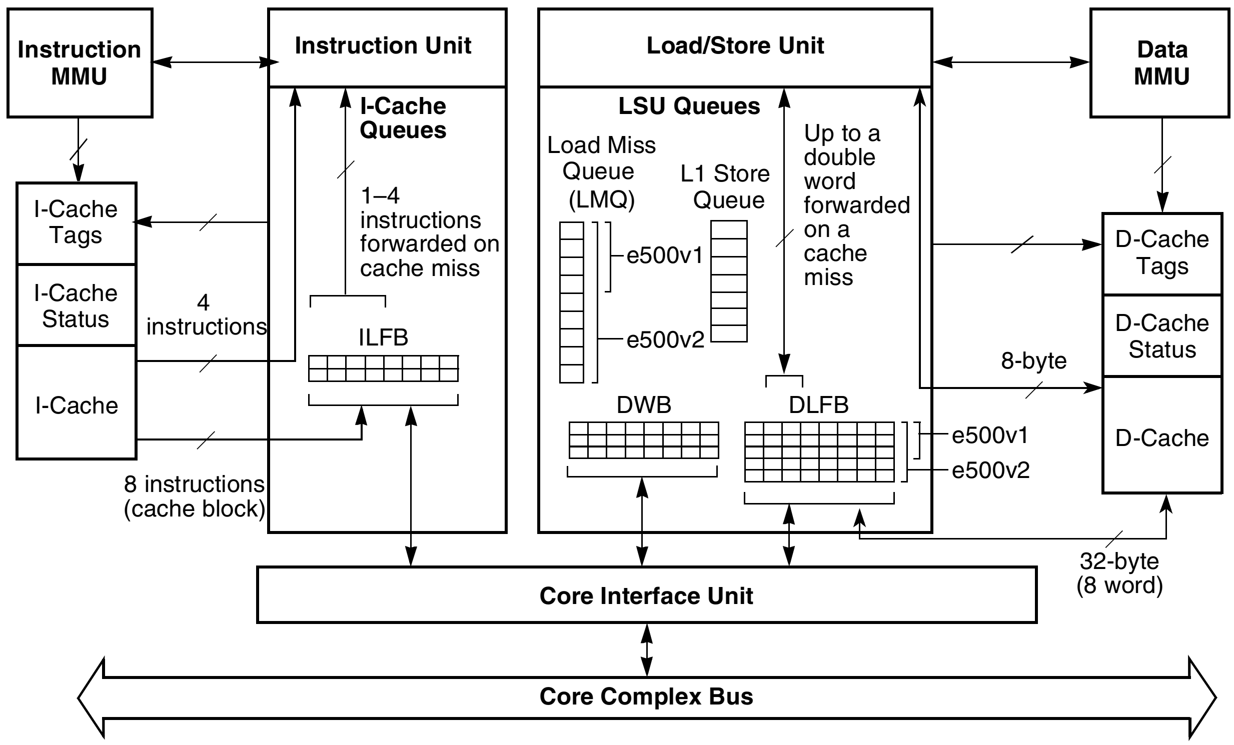


Figure 4-10. Cache/Core Interface Unit Integration

When free of data dependencies, cacheable loads execute in the LSU in a speculative manner with a maximum throughput of one 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.

Table 4-1. Load and Store Queues

Queue	Description
LSU store queue	Stores cannot execute speculatively and are held in the seven-entry store queue, shown in Figure 4-10, until completion logic indicates that the store instruction is to be committed. The store queue arbitrates for L1 data cache access. When arbitration succeeds, data is written to the data cache and the store is removed from the store queue. If a store is caching-inhibited, the operation moves through the store queue to the rest of the memory subsystem.
LSU L1 load miss queue (LMQ)	As loads reach the LSU, it tries to access the cache. On a hit, the cache returns the data. If there is a miss, the LSU allocates an LMQ entry and a DLFB entry. The LSU then queues a bus transaction to read the line. If a subsequent load hits, the cache returns the results. If a subsequent load misses, the LSU allocates a second LMQ entry and, if the load is to a different cache line than the outstanding miss, it allocates the second DLFB entry and queues a second read transaction on the bus. If the load miss is to the same cache line as an outstanding miss, the LSU need not allocate a new DLFB entry. The LSU continues processing load hits and load misses until one of the following conditions occurs: • The LMQ is full and another load miss occurs. • The LSU tries to perform a load miss, all of the DLFB entries are full, and the load is not to any of the cache lines that are represented in the DLFB.

Table 4-1. Load and Store Queues (continued)

Queue	Description
LSU data line fill buffer (DLFB)	DLFB entries are used for loads and cacheable stores. Stores are allocated in the DLFB so loads can access data from the store immediately (loads cannot access data from the L1 store queue). Also, by using the DLFB entries for stores, the LSU frees L1 store queue entries, even on store misses. Multiple cacheable store misses to the same cache line are merged in a DLFB.
LSU data write buffer (DWB)	When a full line of data is available in the DLFB, the data cache is updated. If a data cache update requires a cache line to be evicted, the line is cast out and placed in the DWB until the data has been transferred through the core interface unit to the core complex bus. If global memory's coherency needs to be maintained as a result of bus snooping, the L1 cache can also evict a line to the DWB. (This is a snoop push.) Cast-out and snoop push writes from the L1 cache are cache-line aligned (critical word is not written first), regardless of which word in a modified cache line is accessed. One DWB entry is dedicated for snoop pushes, one is for cast outs, and one can be used for either.

The core interface unit handles all bus transactions initiated by the ILFB, DLFB, and DWB. The core interface unit handles all ordering and bus protocol and is the interface between the core complex and the external memory and caches.

The core interface unit performs transactions through the core complex bus by transferring either the critical–double-word first (8 bytes) or the critical–quad-word first (16 bytes). It then forwards the transaction to the instruction or data line fill buffer critical double word first. The core complex bus also captures snoop addresses for the L1 data cache and the memory reservation (**lwarx** and **stwcx.**) operations.

4.4.3 Simple and Multiple Unit Execution

The e500 has two simple units (SU1, SU2) and one multiple unit (MU). On the e500v2, the MU has an additional six-stage subunit through which all double-precision floating-point instructions pass. The SUs execute all Book E logical and computational instructions except multiplies and divides, SPE single-cycle arithmetic, logical, shift, and splat instructions, and embedded floating-point APU arithmetic and logical instructions. The MU executes multiplies, divides, and multi-cycle arithmetic instructions defined by the SPE and embedded floating-point APUs.

Divide latency depends upon the operand data and ranges from 4 to 35 cycles, as shown in Table 4-2.

Table 4-2. The Effect of Operand Size on Divide Latency

Instruction	Condition	Latency
efsdiv*x*	**r**A or **r**B is 0.0	4
	All others	29
efddiv*x*	All double-precision floating-point divides (e500v2 only)	32
evfsdiv*x*	**r**A or **r**B are 0.0 for both upper and lower	4
	All others	29

Table 4-2. The Effect of Operand Size on Divide Latency (continued)

Instruction	Condition	Latency
divw*x*	**r**A or **r**B is 0	4
	rA representable in 8 bits	11
	rA representable in 16 bits	19
	All other cases	35
evdivw*x*	Both the lower and upper words match the criteria described above for the **divw***x* 4-cycle case.	4
	Assuming the 4-cycle **evdivw***x* case does not apply, the lower and upper words match the criteria described above for the **divw***x* 4- or 11-cycle case.	11
	Assuming neither the 4- or 11-cycle **evdivw***x* cases apply, the lower and upper words match the criteria described above for the **divw***x* 4-, 11-, or 19-cycle case.	19
	All other cases	35

4.4.3.1 MU Divide Execution

The MU provides a bypass path for divides, as shown in Figure 4-11, so the iterative portion of divide execution is performed outside of the MU pipeline, allowing subsequent instructions (except other divides) to execute in the main MU pipeline. Figure 4-11 shows the path that integer divides and both scalar and vector single-precision divide instructions take. The double-precision portion of the MU has a six-stage pipeline, but has a similar divide bypass that splits from the main path after the first stage and before the last.

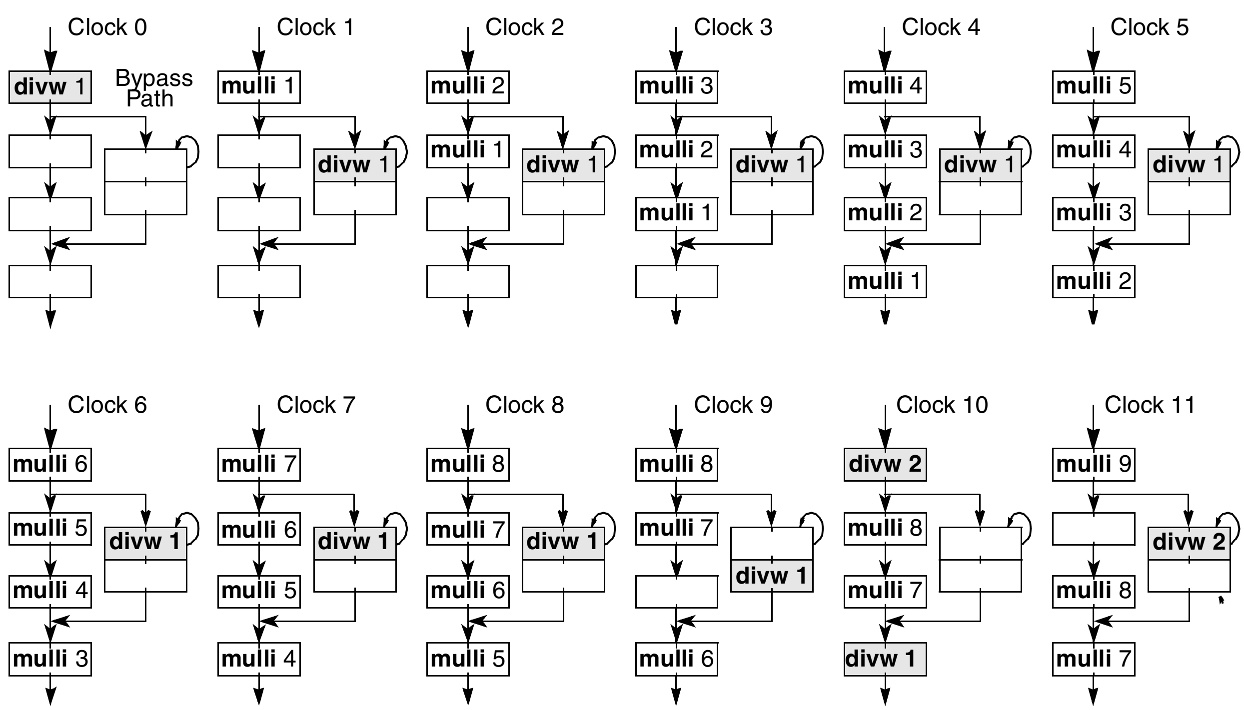


Figure 4-11. MU Divide Bypass Path (Showing an 11-Cycle Divide)

This example shows the pipeline for two **divw** instructions interspersed among **mulli** instructions (although any non-divide instructions that use the MU could have been used in place of the **mulli** instructions). The stages occupied by **divw** instructions are highlighted in grey. In clock cycle 0, the first **divw** is issued to the first stage of the MU. In clock cycle 1, the **divw** moves out of the MU main pipeline into an iterative stage in the two-stage bypass path while the first **mulli** is issued to MU stage 1.

The **divw** iterates in the first stage of the bypass path while a series of **mulli** instructions passes through the main MU pipeline. At the end of clock 4, the first of the **mulli** instructions finishes and leaves the MU pipeline. Although the **mulli** can finish out of order with respect to the **divd**, it cannot complete ahead of it.

In clock cycle 6, a signal is passed to the issue logic to indicate that **divw** 1 will reenter the main MU pipeline in 4 cycles. This creates a bubble that passes down the pipeline, making a space for the **divw** instruction to reenter the main pipeline in clock cycle 10.

A second **divw** enters the first MU stage in clock cycle 10. Had **divw** 2 been issued earlier, it would have stalled in the reservation station until **divw** 1 vacated the second stage of the bypass path. In other words, the MU can hold as many as two divide instructions only if one is in the MU fourth stage (as is the case in clock cycle 10).

Table 4-6 lists SU and MU execution latencies. As Table 4-6 shows, most instructions executed in the SU have a single-cycle execution latency.

4.4.3.2 MU Floating-Point Execution

The MU executes all floating-point arithmetic operations except **efstst***x*, **efdtst***x* and **evfstst***x*. Embedded floating-point operations largely comply with the IEEE-754 floating-point standard. Software exception handling is required to achieve full IEEE 754-compliance because the IEEE floating-point exception model is not fully implemented in hardware.

Floating-point arithmetic instructions, except for divide, execute with 4-cycle latency and 1-cycle throughput. Single-precision floating-point multiply, add, and subtract instructions execute in the four-stage pipeline MU.

If **r**A or **r**B is zero, a floating-point divide takes 4 cycles. All other cases take 29 cycles.

Table 4-8 shows floating-point instruction execution timing.

4.4.4 Load/Store Execution

The LSU executes instructions that move data between the GPRs and the memory unit of the core (made up of the L1 caches and the core interface unit buffers). Figure 4-10 shows the block diagram for the LSU.

The execution of most load instructions is pipelined in the three LSU stages, during which the effective address is calculated, MMU translations are performed, the data cache array and tags are read, and cache way selection and data alignment are performed. Cacheable loads, when free of data dependencies, execute in a speculative manner with a maximum throughput of one instruction per cycle and 3-cycle latency. Data returned from the cache is held in a rename register until the completion logic commits the value to the processor state.

Stores cannot be executed speculatively and must be held in the store queue until completion logic signals that the store instruction is to be committed, at which point the data cache array is updated.

If operands are misaligned, additional latency may be incurred either for an alignment exception or for additional cache or bus accesses. Table 4-7 gives load and store instruction execution latencies.

4.4.4.1 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 4-4.

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 Section 3.1, "Operand Conventions."

The effect of alignment on memory operation performance is the same for big- and little-endian addressing modes, including load-multiple and store-multiple operations.

In Table 4-4, optimal means that one effective address (EA) calculation occurs during the memory operation. Fair 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.

4.5 Memory Performance Considerations

Because the e500 has a maximum instruction throughput of two instructions per clock cycle, lack of memory bandwidth can affect performance. To maximize performance, the e500 must be able to read and write data efficiently. If a system has multiple bus devices, one device may experience long memory latencies while another device (for example, a direct-memory access controller) is using the external bus.

4.6 Instruction Latency Summary

Instruction timing is shown in Table 4-3 through Table 4-7. The latency tables use the following conventions:

- Pipelined load/store and floating-point instructions are shown with cycles of total latency and throughput cycles separated by a colon.
- Floating-point instructions with a single entry in the cycles column are not pipelined. Integer divide instructions are also not pipelined with other divides.

Table 4-3 through Table 4-7 list latencies associated with instructions executed by each execution unit. Figure 4-3 describes branch instruction latencies.

Table 4-3. Branch Operation Execution Latencies

Mnemonic	Cycles	Serialization
bbelr	1	Pre- and postsync
bblels	1	Pre- and postsync
bcctr[**l**]	1	—
bclr[**l**]	1	—
bc[**l**][**a**]	1	—
b[**l**][**a**]	1	—

Table 4-4 lists system operation instruction latencies. The instructions in Table 4-4 are grouped by the serialization they require. Except where otherwise noted, throughput is the same for the instructions within each serialization grouping.

Table 4-4. System Operation Instruction Execution Latencies

Mnemonic	Serialization [1]	Unit	Cycles
isync	Refetch	—[2]	0
mbar	Store	LSU	3:1
msync	Store and postsync.	LSU	Latency depends on bus response time.
mfcr	Move-from	SU1 only	4
mfspr[XER]			
mfmsr	None	SU1	4
mfpmr	None	SU1 only	4 [7]
mfspr[CTR] [3, 4]	None	SU1 or SU2	1
mfspr[LR] [3,5]			
mfspr[DBSR]	Presync, postsync	SU1 only	4
mfspr[SSCR]	Presync	SU1 only	4

Table 4-4. System Operation Instruction Execution Latencies (continued)

Mnemonic	Serialization [1]	Unit	Cycles
mfspr (all others[6])	None	SU1 only	4 [7]
mtcrf (single field)	None	SU1 or SU2	1 (one instruction per execution unit per clock cycle throughput)
mtpmr	Move-to	SU1	1
mtspr[CTR] [8]	Move-to	SU1 only	1
mtspr[LR][9]			
mtcrf (multiple field)	Move-to, presync, postsync	SU1 only	1
mtmsr			
mtspr[CSRR0]			
mtspr[DBCR0]			
mtspr[DBSR]			
mtspr[SSCR]	Move-to, postsync	SU1 only	1
mtspr[XER]			
mtspr[PID*n*]	Move-to, presync	SU1 only	1
mtspr (all others)	Move-to	SU1 only	1 (one instruction per clock cycle throughput)
msync	Store and postsync serialized	LSU	Latency depends on bus response time
rfi	Refetch	—[1]	0
rfci	Refetch	—[1]	0
rfmci	Refetch	—[1]	0
sc	Refetch	—[1]	0
tlbsync	Store	LSU	3 (1instruction per 18 cycle throughput)
wrtee	Postsync, move-to	SU1	1
wrteei	Postsync, move-to	SU1	1

1 Section 4.3.3.3, "Instruction Serialization," describes the different types of serializations listed here.

2 Refetch serialized instructions (if marked with a 0-cycle execution time) do not have an execute stage, and all refetch serialized instructions have 1 cycle between the time they are completed and the time the target/sequential instruction enters the fetch1 stage.

3 Decode out of IQ0 only

4 **mfctr** stalls in decode until any outstanding **mtctr** finishes

5 **mflr** stalls in decode until any outstanding **mtlr** finishes

6 Includes BBTAR, BBEAR, MSR, CSRR*n*, L1CFG*n*, DAC*n*, DBCR*n*, DEAR, DEC, DECAR, ESR, IVPR, IAC*n*, IVOR*n*, MAS*n*, PID*n*, TLBCFG*n*, HID*n*, L1CSR*n*, MMUSCR0, BUCSR, MMUCFG, PIR, PVR, SPRG*n*, SVR, MCSR, MCSRR*n*, SRR*n*, TBL (read and write), TBU (read and write), TCR, TSR, USPRG0,

7 This instruction take 4 cycles to execute in the single-stage SU1. It occupies SU1 for all 4 cycles, so subsequent instructions cannot enter SU1 until this instruction finishes.

8 **mtctr** stalls in decode until any other outstanding **mtctr** finishes.Throughput of 1 per 4 cycles for **mtctr** followed by **mtctr**.

9 **mtlr** stalls in decode until any other outstanding **mtlr** finishes. Throughput of 1 per 4 cycles for **mtlr** followed by **mtlr**.

Table 4-5 lists condition register logical instruction latencies.

Table 4-5. Condition Register Logical Execution Latencies

Mnemonic	Unit	Cycles	Serialization [1]
crand	BU	1	—
crandc	BU	1	—
creqv	BU	1	—
crnand	BU	1	—
crnor	BU	1	—
cror	BU	1	—
crorc	BU	1	—
crxor	BU	1	—
mcrf	BU	1	—
mcrxr	BU	1	Presync, postsync
mfcr	SU1	1	Move-from
mtcrf (single field)	SU1	1	—
mtcrf (multiple fields)	BU	2	Move-to, presync, postsync

[1] Section 4.3.3.3, "Instruction Serialization," describes the different types of serializations listed here.

Table 4-6 lists integer instruction latencies.

Table 4-6. SU and MU PowerPC Instruction Execution Latencies

Mnemonic	Unit	Cycles
addc[**o**][.]	SU1 or SU2	1 [1]
adde[**o**][.]	SU1 or SU2	1 [1]
addi	SU1 or SU2	1
addic	SU1 or SU2	1
addic.	SU1 or SU2	1 [1]
addis	SU1 or SU2	1
addme[**o**][.]	SU1 or SU2	1 [1]
addze[**o**][.]	SU1 or SU2	1 [1]
add[**o**][.]	SU1 or SU2	1 [1]
andc[.]	SU1 or SU2	1 [1]
andi.	SU1 or SU2	1 [1]
andis.	SU1 or SU2	1 [1]
and[.]	SU1 or SU2	1 [1]
cmp	SU1 or SU2	1
cmpi	SU1 or SU2	1

Table 4-6. SU and MU PowerPC Instruction Execution Latencies (continued)

Mnemonic	Unit	Cycles
cmpl	SU1 or SU2	1
cmpli	SU1 or SU2	1
cntlzw[.]	SU1	1 [1]
divwu[**o**][.] **divw**[**o**][.]	MU	4 (**r**A or **r**B = 0, minint/-1) [1,2]
		11 (**r**A can be represented as an 8-bit value within context (signed or unsigned)) [1, 2]
		19 (**r**A operand can be represented as a 16-bit value within context (signed or unsigned)) [1, 2]
		35 (all others) [1, 2]
eqv[.]	SU1 or SU2	1 [1]
extsb[.]	SU1 or SU2	1 [1]
extsh[.]	SU1 or SU2	1 [1]
isel	SU1 or SU2	1
mulhwu[.]	MU	4:1 [1, 3]
mulhw[.]	MU	4:1 [1, 3]
mulli	MU	4:1 [3]
mullw[**o**][.]	MU	4:1 [1, 3]
nand[.]	SU1 or SU2	1 [1]
neg[**o**][.]	SU1 or SU2	1 [1]
nor[.]	SU1 or SU2	1 [1]
orc[.]	SU1 or SU2	1 [1]
ori	SU1 or SU2	1
oris	SU1 or SU2	1
or[.]	SU1 or SU2	1 [1]
rlwimi[.]	SU1 or SU2	1 [1]
rlwinm[.]	SU1 or SU2	1 [1]
rlwnm[.]	SU1 or SU2	1 [1]
slw[.]	SU1 or SU2	1 [1]
srawi[.]	SU1 or SU2	1 [1]
sraw[.]	SU1 or SU2	1 [1]
srw[.]	SU1 or SU2	1 [1]
subfc[**o**][.]	SU1 or SU2	1 [1]
subfe[**o**][.]	SU1 or SU2	1 [1]
subfic	SU1 or SU2	1
subfme[**o**][.]	SU1 or SU2	1 [1]
subfze[**o**][.]	SU1 or SU2	1 [1]

Table 4-6. SU and MU PowerPC Instruction Execution Latencies (continued)

Mnemonic	Unit	Cycles
subf[**o**][.]	SU1 or SU2	1 [1]
tw	SU1 or SU2	1
twi	SU1 or SU2	1
xori	SU1 or SU2	1
xoris	SU1 or SU2	1
xor[.]	SU1 or SU2	1 [1]

[1] If the record bit is set, CR results are not available until after one more cycle. A subsequent instruction can execute while CR results are generated.

[2] The MU provides a bypass path that allows divide instructions to perform the iterative operations necessary for division without blocking the MU pipeline (except to other divide instructions). Therefore, multiply instructions than come after a divide instruction can finish execution ahead of the divide.

[3] 4:1 indicates 4-cycle latency. Once the pipeline is full, throughput is 1 instruction per clock cycle).

Table 4-7 shows load and store instruction latencies. Load/store multiple instruction cycles are represented as a fixed number of cycles plus a variable number of cycles, where *n* represents the number of words accessed by the instruction. Pipelined load/store instructions are shown with total latency and throughput separated by a colon (latency:throughput).

Table 4-7. LSU Instruction Latencies

Mnemonic	Cycles (Latency:Throughput) [1]	Serialization [2]
dcba	3:1	Store
dcbf	3:1	Store
dcbi	3:1	—
dcblc	3:1	—
dcbst	3:1	Store
dcbt	3:1	—
dcbtls	3:1	—
dcbtst	3:1	—
dcbtstls	3:1	—
dcbz	3:1	Store
evldd	3:1	—
evlddx	3:1	—
evldh	3:1	—
evldhx	3:1	—
evldw	3:1	—
evldwx	3:1	—
evlhhesplat	3:1	—

Table 4-7. LSU Instruction Latencies (continued)

Mnemonic	Cycles (Latency:Throughput) [1]	Serialization [2]
evlhhesplatx	3:1	—
evlhhossplat	3:1	—
evlhhossplatx	3:1	—
evlhhousplat	3:1	—
evlhhousplatx	3:1	—
evlwhe	3:1	—
evlwhex	3:1	—
evlwhos	3:1	—
evlwhosx	3:1	—
evlwhou	3:1	—
evlwhoux	3:1	—
evlwhsplat	3:1	—
evlwhsplatx	3:1	—
evlwwsplat	3:1	—
evlwwsplatx	3:1	—
evstdd	3:1	Store
evstddx	3:1	Store
evstdh	3:1	Store
evstdhx	3:1	Store
evstdw	3:1	Store
evstdwx	3:1	Store
evstwhe	3:1	Store
evstwhex	3:1	Store
evstwho	3:1	Store
evstwhox	3:1	Store
evstwwe	3:1	Store
evstwwex	3:1	Store
evstwwo	3:1	Store
evstwwox	3:1	Store
icbi	3:1	Store
icblc	3:1	Store serialized,
icbt CT=0	0 (no-op)	—
icbt CT=1	3:1	—
icbtls	Latency is long and depends on memory latency, as well as other resource availability.	Pre- and postsync serialized.

Table 4-7. LSU Instruction Latencies (continued)

Mnemonic	Cycles (Latency:Throughput) [1]	Serialization [2]
lbz	3:1	—
lbzu	3:1 [3]	—
lbzux	3:1 [3]	—
lbzx	3:1	—
lha	3:1	—
lhau	3:1 [3]	—
lhaux	3:1 [3]	—
lhax	3:1	—
lhbrx	3:1	—
lhz	3:1	—
lhzu	3:1 [3]	—
lhzux	3:1 [3]	—
lhzx	3:1	—
lmw	2 + n	—
lwarx	3	Presync
lwbrx	3:1	—
lwz	3:1	—
lwzu	3:1 [3]	—
lwzux	3:1 [3]	—
lwzx	3:1	—
mbar	3:1	Store serialized
msync	Latency depends on bus response time.	Store and postsync serialized.
stb	3:1	Store
stbu	3:1 [3]	Store
stbux	3:1 [3]	Store
stbx	3:1	Store
sth	3:1	Store
sthbrx		Store
sthu	3:1 [3]	Store
sthux	3:1 [3]	Store
sthx	3:1	Store
stmw	3 + n	Store
stw	3:1	Store
stwbrx	3:1	Store

Table 4-7. LSU Instruction Latencies (continued)

Mnemonic	Cycles (Latency:Throughput) [1]	Serialization [2]
stwcx.	3:1	Store, presync, postsync
stwu	3:1 [3]	Store
stwux	3:1 [3]	Store
stwx	3:1	Store
tlbivax	3:1	—
tlbre	3:1	Presync, postsync, move-to
tlbsx	3:1	Presync, postsync, move-to
tlbwe	3:1	Presync, postsync, move-to

[1] For cache operations, the first number indicates the latency for finishing a single instruction; the second indicates the throughput for a large number of back-to-back cache operations. The throughput cycle may be larger than the initial latency because more cycles may be needed for the data to reach the cache. If the cache remains busy, subsequent cache operations cannot execute.

[2] Section 4.3.3.3, "Instruction Serialization," describes the different types of serializations listed here.

[3] Load and store update instructions are broken into two instructions at dispatch, a load or store instruction that executes in the LSU and an **addi** that executes in either SU. See Section 4.3.3.1, "GPR and CR Rename Register Operation."

Table 4-8 lists instruction latencies for SPE and embedded floating-point computational and logical instructions. SPE loads and stores are executed by the LSU and are described in Table 4-7.

Table 4-8. SPE and Embedded Floating-Point APU Instruction Latencies

Mnemonic	Unit	Cycles (Latency:Throughput)
brinc	SU1 or SU2	1
efdabs	MU	6:1
efdadd	MU	6:1
efdcfsf	MU	6:1
efdcfsi	MU	6:1
efdcfuf	MU	6:1
efdcfui	MU	6:1
efdcmpeq	MU	6:1
efdcmpgt	MU	6:1
efdcmplt	MU	6:1
efdctsf	MU	6:1
efdctsi	MU	6:1
efdctsiz	MU	6:1
efdctuf	MU	6:1
efdctui	MU	6:1
efdctuiz	MU	6:1
efddiv	MU [1]	32
efdmul	MU	6:1

Table 4-8. SPE and Embedded Floating-Point APU Instruction Latencies (continued)

Mnemonic	Unit	Cycles (Latency:Throughput)
efdnabs	MU	6:1
efdneg	MU	6:1
efdsub	MU	6:1
efdtsteq	MU	6:1
efdtstgt	MU	6:1
efdtstlt	MU	6:1
efsabs	SU1 or SU2	1
efsadd	MU	4:1
efscfsf	MU	4:1
efscfsi	MU	4:1
efscfuf	MU	4:1
efscfui	MU	4:1
efscmpeq	SU1	4:1
efscmpgt	SU1	1
efscmplt	SU1	1
efsctsf	MU	4:1
efsctsi	MU	4:1
efsctsiz	MU	4:1
efsctuf	MU	4:1
efsctui	MU	4:1
efsctuiz	MU	4:1
efsdiv	MU [1]	4 (if either **r**A or **r**B is 0.0)
		29 (all other cases)
efsmul	MU	4:1
efsnabs	SU1 or SU2	4:1
efsneg	SU1 or SU2	1
efssub	MU	4:1
efststeq	SU1 or SU2	4:1
efststgt	SU1 or SU2	1
efststlt	SU1 or SU2	1
evabs	SU1	1
evaddiw	SU1	1
evaddsmiaaw	MU	4:1
evaddssiaaw	MU	4:1
evaddumiaaw	MU	4:1
evaddusiaaw	MU	4:1
evaddw	SU1	1
evand	SU1	1

Table 4-8. SPE and Embedded Floating-Point APU Instruction Latencies (continued)

Mnemonic	Unit	Cycles (Latency:Throughput)
evandc	SU1	1
evcmpeq	SU1	1
evcmpgts	SU1	1
evcmpgtu	SU1	1
evcmplts	SU1	1
evcmpltu	SU1	1
evcntlsw	SU1	1
evcntlzw	SU1	1
evdivws **evdivwu**	MU	Both the lower and upper words match the criteria described for the **divw***x* 4-cycle case. [1]
		Assuming the 4-cycle **evdivw***x* case does not apply, the lower and upper words match the criteria described for the **divw***x* 4- or 11-cycle case. [1]
		Assuming neither the 4- or 11-cycle **evdivw***x* cases apply, the lower and upper words match the criteria described for the **divw***x* 4-, 11-, or 19-cycle case. [1]
		All other cases[1]
eveqv	SU1	1
evextsb	SU1	1
evextsh	SU1	1
evfsabs	SU1	1
evfsadd	MU	4:1
evfscfsf	MU	4:1
evfscfsi	MU	4:1
evfscfuf	MU	4:1
evfscfui	MU	4:1
evfscmpeq	MU	4:1
evfscmpgt	MU	4:1
evfscmplt	MU	4:1
evfsctsf	MU	4:1
evfsctsi	MU	4:1
evfsctsiz	MU	4:1
evfsctuf	MU	4:1
evfsctui	MU	4:1
evfsctuiz	MU	4:1
evfsdiv	MU	4 (if either **r**A or **r**B is 0.0)
		29 (all other cases)
evfsmul	MU	4:1
evfsnabs	SU1	1
evfsneg	SU1	1
evfssub	MU	4:1

Table 4-8. SPE and Embedded Floating-Point APU Instruction Latencies (continued)

Mnemonic	Unit	Cycles (Latency:Throughput)
evfststeq	SU1	1
evfststgt	SU1	1
evfststlt	SU1	1
evmergehi	SU1	1
evmergehilo	SU1	1
evmergelo	SU1	1
evmergelohi	SU1	1
evmhegsmfaa	MU	4:1
evmhegsmfan	MU	4:1
evmhegsmiaa	MU	4:1
evmhegsmian	MU	4:1
evmhegumiaa	MU	4:1
evmhegumian	MU	4:1
evmhesmf	MU	4:1
evmhesmfa	MU	4:1
evmhesmfaaw	MU	4:1
evmhesmfanw	MU	4:1
evmhesmi	MU	4:1
evmhesmia	MU	4:1
evmhesmiaaw	MU	4:1
evmhesmianw	MU	4:1
evmhessf	MU	4:1
evmhessfa	MU	4:1
evmhessfaaw	MU	4:1
evmhessfanw	MU	4:1
evmhessiaaw	MU	4:1
evmhessianw	MU	4:1
evmheumi	MU	4:1
evmheumia	MU	4:1
evmheumiaaw	MU	4:1
evmheumianw	MU	4:1
evmheusiaaw	MU	4:1
evmheusianw	MU	4:1
evmhogsmfaa	MU	4:1
evmhogsmfan	MU	4:1
evmhogsmiaa	MU	4:1
evmhogsmian	MU	4:1
evmhogumiaa	MU	4:1

Table 4-8. SPE and Embedded Floating-Point APU Instruction Latencies (continued)

Mnemonic	Unit	Cycles (Latency:Throughput)
evmhogumian	MU	4:1
evmhosmf	MU	4:1
evmhosmfa	MU	4:1
evmhosmfaaw	MU	4:1
evmhosmfanw	MU	4:1
evmhosmi	MU	4:1
evmhosmia	MU	4:1
evmhosmiaaw	MU	4:1
evmhosmianw	MU	4:1
evmhossf	MU	4:1
evmhossfa	MU	4:1
evmhossfaaw	MU	4:1
evmhossfanw	MU	4:1
evmhossiaaw	MU	4:1
evmhossianw	MU	4:1
evmhoumi	MU	4:1
evmhoumia	MU	4:1
evmhoumiaaw	MU	4:1
evmhoumianw	MU	4:1
evmhousiaaw	MU	4:1
evmhousianw	MU	4:1
evmra	MU	4:1
evmwhsmf	MU	4:1
evmwhsmfa	MU	4:1
evmwhsmi	MU	4:1
evmwhsmia	MU	4:1
evmwhssf	MU	4:1
evmwhssfa	MU	4:1
evmwhumi	MU	4:1
evmwhumia	MU	4:1
evmwlsmiaaw	MU	4:1
evmwlsmianw	MU	4:1
evmwlssiaaw	MU	4:1
evmwlssianw	MU	4:1
evmwlumi	MU	4:1
evmwlumia	MU	4:1
evmwlumiaaw	MU	4:1
evmwlumianw	MU	4:1

Table 4-8. SPE and Embedded Floating-Point APU Instruction Latencies (continued)

Mnemonic	Unit	Cycles (Latency:Throughput)
evmwlusiaaw	MU	4:1
evmwlusianw	MU	4:1
evmwsmf	MU	4:1
evmwsmfa	MU	4:1
evmwsmfaa	MU	4:1
evmwsmfan	MU	4:1
evmwsmi	MU	4:1
evmwsmia	MU	4:1
evmwsmiaa	MU	4:1
evmwsmian	MU	4:1
evmwssf	MU	4:1
evmwssfa	MU	4:1
evmwssfaa	MU	4:1
evmwssfan	MU	4:1
evmwumi	MU	4:1
evmwumia	MU	4:1
evmwumiaa	MU	4:1
evmwumian	MU	4:1
evnand	SU1	1
evneg	SU1	1
evnor	SU1	1
evor	SU1	1
evorc	SU1	1
evrlw	SU1	1
evrlwi	SU1	1
evrndw	SU1	1
evsel	SU1	1
evslw	SU1	1
evslwi	SU1	1
evsplatfi	SU1	1
evsplati	SU1	1
evsrwis	SU1	1
evsrwiu	SU1	1
evsrws	SU1	1
evsrwu	SU1	1
evsubfsmiaaw	MU	4:1
evsubfssiaaw	MU	4:1
evsubfumiaaw	MU	4:1

Table 4-8. SPE and Embedded Floating-Point APU Instruction Latencies (continued)

Mnemonic	Unit	Cycles (Latency:Throughput)
evsubfusiaaw	MU	4:1
evsubfw	SU1	1
evsubifw	SU1	1
evxor	SU1	1

[1] The MU bypass path allows divide instructions to perform the iterative operations necessary for division without blocking the MU pipeline (except to other divide instructions). Therefore, multiply instructions than follow a divide instruction can finish execution ahead of the divide. See Section 4.4.3, "Simple and Multiple Unit Execution."

4.7 Instruction Scheduling Guidelines

This section provides an overview of instruction scheduling guidelines, followed by detailed examples showing how to optimize scheduling with respect to various pipeline stages. Performance can be improved by avoiding resource conflicts and scheduling instructions to take fullest advantage of the parallel execution units. Instruction scheduling can be improved by observing the following guidelines:

- To reduce branch mispredictions, separate the instruction that sets CR bits from the branch instruction that evaluates them. Because there can be no more than 26 instructions in the processor (with the instruction that sets CR in CQ0 and the dependent branch instruction in IQ11), there is no advantage to having more than 24 instructions between them.
- When branching to a location specified by the CTR or LR, separate the **mtspr** instruction that initializes the CTR or LR from the dependent branch instruction. This ensures the register values are immediately available to the branch instruction.
- Schedule instructions so two can be dispatched at a time.
- Schedule instructions to minimize stalls due to busy execution units.
- Avoid scheduling high-latency instructions close together. Interspersing single-cycle latency instructions between longer-latency instructions minimizes the effect that instructions such as integer divide can have on throughput.
- Avoid using serializing instructions.
- Schedule instructions to avoid dispatch stalls. As many as 14 instructions can be assigned CR and GPR renames and can be assigned CQ entries; therefore, 14 instructions can be in the execute stages at any one time. (However, note the exception of load or store with update instructions, which are broken into two instructions at dispatch.)
- Avoid branches where possible; favor not-taken branches over taken branches.

The following sections give detailed information on optimizing code for e500 pipeline stages.

4.7.1 Fetch/Branch Considerations

The following lists the resources required to avoid stalling the fetch unit in the course of branch resolution:

- The **bclr** instruction requires LR availability for resolution.
- The branch conditional on counter decrement and the CR condition requires CTR availability or the CR condition must be false.

4.7.1.1 Dynamic Prediction versus No Branch Prediction

No branch prediction (BUCSR[BPEN] = 0) means that the e500 predicts every branch as not taken. The dynamic predictor is ignored. Sometimes this simplistic prediction is superior, either through informed guessing or through available profile-directed feedback. Run time for code with no prediction is more nearly deterministic, which can be useful in embedded systems.

Note that disabling and enabling the BTB (by clearing and setting BPEN) do not affect the BTB's contents or locks.

4.7.1.1.1 Position-Independent Code

Position-independent code is used when not all addresses are known at compile time or link time. Because performance is typically not good, position-independent code should be avoided when possible.

4.7.2 Dispatch Unit Resource Requirements

The following is a general list of the most common reasons that instructions may stall in the dispatch unit:

- Presync serializing instructions cannot decode until all previous instructions have completed.
- Postsync serializing instructions inhibit the decoding of any further instructions until they have completed.
- Decode stalls if there is no room in the CQ for two instructions, regardless of how many are eligible for decode.
- When an unconditional branch misses in the BTB, the decoder stalls any further decode until it receives an indication that the unconditional branch executed and redirected fetch.
- A branch-class instruction cannot be decoded if there is no room in the BIQ. Although **mtctr** and **mtlr** do not go to the BIQ, they are also affected by this stall.
- The decode stage cannot decode a second branch-class instruction in a single cycle. This applies only to IQ1.
- Decoding stops if there are no free entries in the GIQ, even if the next instruction to decode is to the BU or does not require an issues queue slot.

Additional conditions are described in the *e500 Software Optimization Guide*. The following sections describe how to optimize code for dispatch.

4.7.2.1 Dispatch Groupings

Maximum dispatch throughput is two instructions per cycle. The dispatch process includes checking for availability of CQ and issue queue entries and a branch ready check.

The dispatcher can send two instructions to the two issues queues, with a maximum of two to the GIQ and one to the BIQ.

The dispatcher can rename as many as two GPRs per cycle, so a two-instruction dispatch window composed of **add** and **mulli** could be dispatched in one cycle.

Note that a load/store update form (for example, **lwzu**), requires a rename register for the update. This means an **lwzu** needs two GPR renames. The restriction to two GPR renames in a dispatch group means that the sequence, **lwzu**, **add**, cannot be dispatched in one cycle.

4.7.3 Issue Queue Resource Requirements

Instructions cannot be issued unless the specified execution unit is available. The following sections describe how to optimize use of the issue queues.

4.7.3.1 General Issue Queue (GIQ)

As many as two instructions can be dispatched to the four-entry general issue queue (GIQ) per cycle. As many as two instructions can be issued in any order from GIQ0 and GIQ1 to the LSU, MU, SU1, and SU2 reservation stations.

Issuing instructions out-of-order can help in a number of situations. For example, if the MU is busy and a multiply is stalled at the bottom GIQ entry, the instruction in the next GIQ entry can be issued to LSU or SU1, bypassing that multiply.

4.7.3.2 Branch Issue Queue (BIQ)

One instruction per clock cycle can be dispatched to the BIQ. One instruction can be issued to the branch execution unit out of BIQ0.

4.7.4 Completion Unit Resource Requirements

The e500 completion queue has 14 entries, so as many as 14 instructions can be in execution. The following resources are required to avoid stalls in the completion unit; note that the two completion entries are described as CQ0–CQ1, where CQ0 is located at the end of the CQ (see

Figure 4-2). The following list describes some of the common conditions that may cause instructions to stall in the completion stage:

- A refetch-serialized instruction has generated a pending flush of the instruction pipeline.
- There are no finished instructions in the CQ.
- Only one store instruction can complete per cycle.
- A store cannot complete out of CQ1 if the instruction producing its data value is completing out of CQ0 at the same time.
- Some instructions must complete out of CQ0.
- All refetch-serialized instructions except for **isync** must stall an extra cycle before completing. This includes phantom branches, as described in Section 4.4.1.3.4, "BTB Special Cases—Phantom Branches and Multiple Matches."
- If the instruction in CQ0 is a refetch-serialized instruction, the entry in CQ1 should not be considered valid.
- If the instruction in CQ0 is a mispredicted branch, the entry in CQ1 should not be considered valid.

These and other less common conditions are described in the *e500 Software Optimization Guide*.

4.7.4.1 Completion Groupings

The e500 can retire as many as two instructions per cycle. Only two renames can be retired per cycle. For example, an **lwzu**, **add** sequence has three GPR rename targets so both instructions cannot retire in the same cycle. The **lwzu** is broken during decode into two parts, each of which updates one rename. Both halves of the **lwzu** instruction can retire in one cycle. The **add** retires 1 cycle later.

4.7.5 Serialization Effects

The e500 supports the serialization described in Section 4.3.3.3, "Instruction Serialization."

Tables in Section 4.6, "Instruction Latency Summary," indicate which instructions require serialization.

4.7.6 Execution Unit Considerations

The following sections describe how to optimize use of the execution units.

4.7.6.1 SU Considerations

Each SU has one reservation station in which instructions are held until operands are available. Also note that some SU1 instructions take more than one cycle and that some are not fully pipelined. A new instruction cannot begin execution if the previous instruction is still executing. Although the majority of instructions executed by the SUs require only a single cycle, **mfcr** and many **mfspr** instructions require several cycles and can cause stalls.

A new instruction cannot execute if one of its operands is not yet available. A new instruction that is marked as completion-serialized cannot begin execution until it is signalled from the completion unit that it is the oldest instruction.

4.7.6.2 MU Considerations

The MU is similar to the SUs. The MU has one reservation station. The bypass unit, described in Section 4.4.3, "Simple and Multiple Unit Execution," allows divide instructions to execute in parallel with other MU instructions. Note the following:

- A new instruction cannot execute if one of its operands is not yet available.
- A new instruction that is marked as completion-serialized cannot begin execution until it is signaled from the completion unit that it is the oldest instruction.
- A new divide instruction cannot begin execution if the previous divide instruction is still executing.
- A new instruction cannot begin execution if it would finish execution at the same time as an executing divide instruction. As shown in Figure 4-1 and Figure 4-1, the MU consists of a multiply subunit and a divide subunit. These subunits share the same reservation station and result bus. In general, when a divide is in progress (which could take up to 35 cycles), new multiply instructions can proceed down the four-stage multiply subunit. However, because there is only one result bus, the processor ensures that a divide and a multiply do not collide on the result bus, with both attempting to write results at the same time. When a divide is 4 cycles away from providing its result, it blocks a new 4-cycle multiply from beginning execution (inserting a bubble in the multiply subunit) so that when the divide provides its result, no multiply will collide with it.

4.7.6.3 LSU Considerations

The following sections describe situations that can affect LSU timing.

4.7.6.3.1 Load/Store Interaction

When loads and stores are intermixed, stores normally lose arbitration to the cache. A store that repeatedly loses arbitration can stay in the core interface unit store queue much longer than 3 cycles, which is not normally a performance problem because a store in this queue is effectively part of the architecture-defined state. However, sometimes—including if the store queue fills up or if a store causes a pipeline stall (as in a partial address alias case of store to load)—the arbiter gives higher priority to the store, guaranteeing forward progress.

4.7.6.3.2 Misalignment Effects

Misalignment, particularly the back-to-back misalignment of loads, can cause strange performance effects. The e500 splits misaligned transactions into two transactions, so misaligned load latency is at least 1 cycle greater than the default latency.

For loads that hit in the cache, the throughput of the LSU degrades to one misaligned load every 3 cycles. Similarly, stores can be translated at a rate of one store per 3 cycles. Additionally, after translation, each misaligned store is treated as two separate store queue entries, each requiring a cache access.

A word or half-word storage access requires multiple accesses if it crosses a double-word boundary. Extended vector loads and stores cause alignment exceptions if they cross their natural alignment boundaries (as show in Figure 4-9).

Table 4-9. Natural Alignment Boundaries for Extended Vector Instructions

Instruction	Boundary
evld{**d,w,h**} **evld**{**d,w,h**}**x** **evstd**{**d,w,h**} **evstd**{**d,w,h**}**x**	Double-word
evlwwsplat{**x**} **evlwhe**{**x**} **evlwhou**{**x**} **evlwhos**{**x**} **evlwhsplat**{**x**} **evstwwe**{**x**} **evstwwo**{**x**} **evstwhe**{**x**} **evstwho**{**x**}	Word
evlhhesplat{**x**} **evlhhousplat**{**x**} **evlhhossplat**{**x**}	Half

Frequent unaligned accesses are discouraged because of the impact on performance.

Note the following:

- Accesses that cross a translation boundary may be restarted—that is, a misaligned access that crosses a page boundary is entirely restarted if the second portion of the access causes a TLB miss. This may result in the first portion being accessed twice.
- Accesses that cross a translation boundary where the endianness changes cause a byte-ordering DSI exception.
- Future generations of high-performance microprocessors that implement the PowerPC architecture may experience greater misalignment penalties.

If a load misses in the L1 data cache, critical data forwarding occurs, followed shortly by the rest of the cache line.

4.7.6.3.3 Load Miss Pipeline

As shown in Figure 4-10, the e500v1 supports as many as four outstanding load misses in the load miss queue (LMQ); the e500v2 LMQ supports as many as nine. Table 4-10 shows a load followed by a dependent **add**. Here, the load misses in the data cache and the full line is reloaded into the data cache.

Table 4-10. Data Cache Miss, L2 Cache Hit Timing

Instruction	0	1	2	3	4	5	6
lwz r4,0x0(**r**9)	E0	E1	Miss	LMQ0	LMQ0/E2	C	
add r5,**r**4,**r**3	—	—	—	—	—	E	C

Chapter 5
Interrupts and Exceptions

This chapter provides a general description of the PowerPC Book E interrupt and exception model as it is implemented in the e500 core complex. It identifies and describes the portions of the interrupt model that are defined by the Book E architecture and by the Freescale implementation standards (EIS).

5.1 Overview

A note on terminology:

The Book E architecture has defined additional resources for interrupt handling. As a result, the terms 'interrupt' and 'exception' differ somewhat from their use in previous Freescale documentation, such as the *Programming Environments Manual*. Use of these terms in this document are 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, causes the processor to take an interrupt. Book E describes exceptions as being generated by signals from internal and external peripherals, instructions, the internal timer facility, debug events, or error conditions.

There are three categories of interrupts, described as follows:

- Noncritical interrupts—First-level interrupts that let the processor change program flow to handle conditions generated by external signals, errors, or unusual conditions arising from program execution, or from programmable timer-related events.

  These interrupts are largely identical to those defined by the OEA portion of the PowerPC architecture. They use save and restore registers (SRR0/SRR1) to save state when they are taken, and they use the **rfi** instruction to restore state. Asynchronous noncritical interrupts can be masked by the external interrupt enable bit, MSR[EE].
- Critical interrupts—Critical interrupts (critical input, watchdog timer, and debug interrupts) can be taken during a noncritical interrupt or during regular program flow. They use the critical save and restore registers (CSRR0/CSRR1) to save state when they are taken, and they use the **rfci** instruction to restore state.

  Critical input and watchdog timer critical interrupts can be masked by the critical enable bit, MSR[CE]. Debug events can be masked by the debug enable bit MSR[DE]. Book E defines the critical input, watchdog timer, debug, and machine check interrupts as critical interrupts, but the EIS defines a third set of resources for the machine check interrupt, as described below.

- Machine check interrupt—The EIS defines a separate set of resources for the machine check interrupt, which is similar to the Book E–defined critical interrupt type. Machine check interrupts on an EIS device use the machine check save and restore registers (MCSRR0/MCSRR1) to save state when they are taken, and they use the **rfmci** instruction to restore state. These interrupts can be masked by the machine check enable bit, MSR[ME].

All interrupts except the machine check interrupt are ordered within the two categories of noncritical and critical, such that only one interrupt of each category is reported, and when it is processed (taken), no program state is lost. Because save/restore register pairs are serially reusable, program state may be lost when an unordered interrupt is taken. (See Section 5.10, "Interrupt Ordering and Masking".)

All interrupts except the machine check interrupt are context synchronizing as defined in the instruction model chapter of the EREF. A machine check interrupt acts like a context-synchronizing operation with respect to subsequent instructions.

5.2 e500 Interrupt Definitions

This section gives an overview of additions and modifications to the Book E interrupt model defined by the EIS and implemented on the e500. Specific details are also provided throughout this chapter. Except for the following, the core complex reports exceptions as specified in Book E:

- The machine check exception differs as follows:
  - It is not processed as a critical interrupt, but uses MCSRR0 and MCSRR1 for saving the return address and the MSR in case the machine check is recoverable.
  - Return From Machine Check Interrupt instruction (**rfmci**) is implemented to support the return to the address saved in MCSRR0.
  - A machine check syndrome register, MCSR, is used to log the cause of the machine check (instead of ESR). See Section 2.7.2.4, "Machine Check Syndrome Register (MCSR)," for a description of the MCSR.

  The core complex reports the machine check exception as described in Section 5.7.2, "Machine Check Interrupt."

- The following interrupts are defined for use with the embedded floating-point and signal-processing (SPE) APUs:
  - — SPE/embedded floating-point unavailable interrupt. IVOR32 (SPR 528) contains the vector offset. See Section 5.7.15.1, "SPE/Embedded Floating-Point APU Unavailable Interrupt."
  - — Embedded floating-point data interrupt. IVOR33 (SPR 529) contains the vector offset. See Section 5.7.15.2, "Embedded Floating-Point Data Interrupt."
  - — Embedded floating-point round interrupt. IVOR34 (SPR 530) contains the vector offset. See Section 5.7.15.3, "Embedded Floating-Point Round Interrupt."

  The following additional bits are defined to support SPE and SPFP exceptions:
  - — MSR[38] is defined as the vector available bit (SPE). If this bit is clear and software attempts to execute any of the SPE instructions, the SPE unavailable interrupt is taken. If this bit is set, software can execute any SPE instructions.

NOTE

On the e500v1, all SPFP instructions also require MSR[SPE] to be set. Any attempt to execute a vector or scalar SPFP instruction when MSR[SPE] is 0 causes an SPE APU unavailable interrupt. On the e500v2, when MSR[SPE] is 0, this interrupt is caused by DPFP instructions and SPFP vector instructions, but not by SPFP scalar instructions (in other words, only those instructions that access the upper half of the GPRs).

Table 5-1 presents this information in table form.

Table 5-1. SPE APU Unavailable Interrupt Generation When MSR[SPE] = 0

APU	e500v1	e500v2
SPE	x	x
Single-Precision Floating-Point Vector	x	x
Single-Precision Floating-Point Scalar	x	—
Double-Precision Floating-Point	N/A	x

For more information, see the "Embedded Vector and Scalar Single-Precision Floating-Point APU Instructions," section of the "Instruction Model" chapter of the EREF.

— ESR[SPE], the SPE exception bit, is set when the processor reports an exception related to the execution of SPFP or SPE instructions.

NOTE

The SPE APU and embedded floating-point APU functionality is implemented in all PowerQUICC III devices. However, these instructions will not be supported in devices subsequent to PowerQUICC III. Freescale Semiconductor strongly recommends that use of these instructions be confined to libraries and device drivers. Customer software that uses SPE or embedded floating-point APU instructions at the assembly level or that uses SPE intrinsics will require rewriting for upward compatibility with next-generation PowerQUICC devices.

Freescale Semiconductor offers a libmoto_e500 library that uses SPE instructions. Freescale will also provide libraries to support next-generation PowerQUICC devices.

- The debug exception implementation does not support IAC3, IAC4, DAC3, and DAC4 comparisons.
- The core complex supports instruction address compare (IAC1 and IAC2) and data address compare (DAC1 and DAC2) for effective addresses only. Real-address support is not provided.
- The e500 does not support the Book E–defined floating-point unavailable and auxiliary processor unavailable interrupts.
- Data value compare (DVC) debug exceptions are not supported.
- The interrupt priorities differ from those specified in Book E as described in Section 5.11.1, “e500 Exception Priorities.”
- Alignment exceptions. Vector operations can cause alignment exceptions as described in Section 5.7.6, “Alignment Interrupt.”
- Book E and the machine check APU define sources of externally generated interrupts.

5.2.1 Recoverability from Interrupts

All interrupts except some machine check interrupts are recoverable. The state of the core complex (return address and MSR contents) is saved when a machine check interrupt is taken. The conditions that cause a machine check may or may not prohibit recovery. Section 5.7.2.1, “Core Complex Bus (CCB) and L1 Cache Machine Check Errors,” provides additional information about machine check recoverability.

5.3 Interrupt Registers

Table 5-2 summarizes registers used for interrupt handling.

Table 5-2. Interrupt Registers Defined by the PowerPC Architecture

Register	Description
Book E Interrupt Registers	
Save/restore register 0 (SRR0)	On a noncritical interrupt, SRR0 is set to the current or next instruction address. When **rfi** is executed, instruction execution continues at the address in SRR0. In general, SRR0 contains the address of the instruction that caused the noncritical interrupt or the address of the instruction to return to after a noncritical interrupt is serviced.
Save/restore register 1 (SRR1)	When a noncritical interrupt is taken, MSR contents are placed into SRR1. When **rfi** is executed, SRR1 contents are placed into the MSR. SRR1 bits that correspond to reserved MSR bits are also reserved. Note that an MSR bit that is reserved may be altered by **rfi**.
Critical save/restore register 0 (CSRR0)	When a critical interrupt is taken, CSRR0 is set to the current or next instruction address. When **rfci** is executed, instruction execution continues at the address in CSRR0. In general, CSRR0 contains the address of the instruction that caused the critical interrupt, or the address of the instruction to return to after a critical interrupt is serviced.
Critical save/restore register 1 (CSRR1)	When a critical interrupt is taken, MSR contents are placed into CSRR1. When **rfci** is executed, CSRR1 contents are placed into the MSR. CSRR1 bits that correspond to reserved MSR bits are also reserved. Note that an MSR bit that is reserved may be altered by **rfci**.
Data exception address register (DEAR)	DEAR contains the address referenced by a load, store, or cache management instruction that caused an alignment, data TLB miss, or data storage interrupt.
Interrupt vector prefix register (IVPR)	IVPR[32–47] 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 IVPR to form the address of the interrupt handling routine. IVPR[48–63] are reserved.
Exception syndrome register (ESR)	Provides a syndrome to differentiate between exceptions that can generate the same interrupt type. When one of these types of interrupts is generated, bits corresponding to the specific exception that generated the interrupt are set and all other ESR bits are cleared. Other interrupt types do not affect the ESR. ESR does not need to be cleared by software. Table 5-3 shows ESR bit definitions. The EIS defines ESR[56] as the SPE exception bit (SPE). It is set when the processor reports an exception related to the execution of an embedded floating-point or SPE instruction. Note that the EIS definition of the machine check interrupt uses the machine check syndrome register (MCSR) rather than the ESR.
Interrupt vector offset registers (IVORs)	Holds the quad-word index from the base address provided by the IVPR for each interrupt type. IVOR0–IVOR15 are provided for defined interrupt types. SPR numbers corresponding to IVOR16–IVOR31 are reserved. IVOR[32–47,60–63] are reserved. SPR numbers for IVOR32–IVOR63 are allocated for implementation-dependent use. (IVOR32–IVOR34 (SPR 528–530) are used by interrupts defined by the EIS.) IVOR assignments are shown below. <u>IVOR Number</u> <u>Interrupt Type</u> — <u>IVOR Number</u> <u>Interrupt Type</u> IVOR0 Critical input — IVOR11 Fixed-interval timer interrupt IVOR1 Machine check — IVOR12 Watchdog timer interrupt IVOR2 Data storage — IVOR13 Data TLB error IVOR3 Instruction storage — IVOR14 Instruction TLB error IVOR4 External input — IVOR15 Debug IVOR5 Alignment — VOR32 SPE APU unavailable IVOR6 Program — IVOR33 Embedded floating-point data IVOR8 System call — IVOR34 Embedded floating-point round IVOR10 Decrementer — IVOR35 Performance monitor

Table 5-2. Interrupt Registers Defined by the PowerPC Architecture (continued)

Register	Description
Machine state register (MSR)	MSR[38] is defined as the vector available bit (SPE). It functions as follows: 0 For the e500v2, if software attempts to execute an instruction that tries to access the upper word of a 64-bit GPR, an SPE APU unavailable interrupt is taken. For the e500v1, the interrupt is also taken if an attempt is made to execute an embedded SPFP scalar instruction. 1 Software can execute any embedded floating-point or SPE instructions.
EIS-Specific Interrupt Registers	
Machine check save/restore register 0 (MCSRR0)	When a machine check interrupt is taken, MCSRR0 is set to the current or next instruction address. When **rfmci** is executed, instruction execution continues at the address in MCSRR0. In general, MCSRR0 contains the address of the instruction that caused the machine check interrupt, or the address of the instruction to return to after a machine check interrupt is serviced.
Machine check save/restore register 1 (MCSRR1)	When a machine check interrupt is taken, MSR contents are placed into MCSRR1. When **rfmci** is executed, MCSRR1 contents are restored to MSR. MCSRR1 bits that correspond to reserved MSR bits are also reserved. Note that an MSR bit that is reserved may be altered by **rfmci**.
Machine check syndrome register (MCSR)	When a machine check interrupt is taken, the MCSR is updated to differentiate between machine check conditions. Table 5-4 lists e500 bit assignments. The MCSR also indicates whether a machine check condition is recoverable. ABIST status is logged in MCSR[48–54]. These read-only bits do not initiate machine check (or any other interrupt). An ABIST bit being set indicates an error being detected in the corresponding module. Note that processors that do not implement the machine check APU use the Book E–defined ESR for this purpose.
Machine check address register (MCAR)	When a machine check interrupt is taken, MCAR is updated with the address of the data associated with the machine check. Note that if a machine check interrupt is caused by a signal, the MCAR contents are not meaningful. See Section 2.7.2.3, "Machine Check Address Register (MCAR)."

Table 5-3 shows ESR bit definitions.

Table 5-3. Exception Syndrome Register (ESR) Definition

Bits	Name	Syndrome	Interrupt Types
32–35	—	Allocated	—
36	PIL	Illegal instruction exception	Program
37	PPR	Privileged instruction exception	Program
38	PTR	Trap exception	Program
39	—	Reserved, should be cleared.	—
40	ST	Store operation	Alignment, data storage, data TLB error
41	—	Reserved, should be cleared.	—
42	DLK	Cache locking. Settings are implementation-dependent. On the e500, DLK is set when a DSI occurs because **dcbtls**, **dcbtstls**, or **dcblc** is executed in user mode and MSR[UCLE] = 0.	Data storage
43	ILK	(EIS) Set when a DSI occurs because **icbtl** or **icblc** is executed in user mode (MSR[PR] = 1) and MSR[UCLE] = 0	Data storage
44–45	—	Reserved, should be cleared. [1]	—

Table 5-3. Exception Syndrome Register (ESR) Definition (continued)

Bits	Name	Syndrome	Interrupt Types
46	BO	Byte-ordering exception	Data storage, instruction storage
47–55	—	Reserved, should be cleared.	—
56	SPE	SPE exception bit. Book E allocates this bit for implementation-dependent use, so it may have different functions on other implementations.	—
57–63	—	Allocated for implementation-dependent use. Reserved, should be cleared.	—

[1] Book E defines bit 45 as PUO (unimplemented operation exception). On the e500, unimplemented instructions are handled as illegal instructions.

An implementation may define additional ESR bits to identify implementation-specific or architected interrupt types; the EIS defines ESR[ILK] and ESR[SPE].

NOTE

System software may need to identify the type of instruction that caused the interrupt and examine the TLB entry and ESR to fully identify the exception or exceptions. For example, because both protection violation and byte-ordering exception conditions may be present, and either causes a data storage interrupt, system software would have to look beyond ESR[BO], such as the state of MSR[PR] in SRR1 and the TLB entry page protection bits, to determine if a protection violation also occurred.

Table 5-4 shows MCSR bit definitions. Section 5.7.2.1, "Core Complex Bus (CCB) and L1 Cache Machine Check Errors," provides information about machine check recoverability.

Table 5-4. Machine Check Syndrome Register (MCSR) Field Descriptions

Bits	Name	Description
32	MCP	Machine check input signal
33	ICPERR	Instruction cache parity error
34	DCP_PERR	Data cache push parity error
35	DCPERR	Data cache parity error
36–55	—	Reserved, should be cleared.
56	BUS_IAERR	Bus instruction address error
57	BUS_RAERR	Bus read address error
58	BUS_WAERR	Bus write address error
59	BUS_IBERR	Bus instruction data bus error
60	BUS_RBERR	Bus read data bus error
61	BUS_WBERR	Bus write bus error
62	BUS_IPERR	Bus instruction parity error
63	BUS_RPERR	Bus read parity error

5.4 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)
- In general, an attempt by an application program to access a nonexistent SPR (unimplemented operation instruction exception-type program interrupt). Note the following behavior defined by the EIS:
  — If MSR[PR] = 1 (user mode), SPR bit 5 = 0 (user-accessible SPR), and the SPR number is invalid, an illegal instruction exception is taken.
  — If MSR[PR] = 0 (supervisor mode) and the SPR number is invalid, an illegal instruction exception is taken.
  — If MSR[PR] = 1, SPR bit 5 = 1, and invalid SPR address (supervisor-only SPR), a privileged instruction exception-type program interrupt is taken.
- Execution of a defined instruction using an invalid form (illegal instruction exception-type program interrupt, unimplemented operation exception-type program interrupt, or privileged instruction exception-type program interrupt). The e500 does not support unimplemented operation exceptions. Such conditions are processed as illegal instruction exceptions.
- 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 interrupt)
- Execution of a **trap** instruction whose trap condition is met (trap interrupt type)
- Execution of a defined instruction that is not implemented (illegal instruction exception or unimplemented operation exception-type program interrupt)
- Execution of an allocated instruction that is not implemented (illegal instruction exception or unimplemented operation exception-type program interrupt)

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).

5.5 Interrupt Classes

All interrupts except machine check are categorized by two independent characteristics:

- Critical/noncritical. Some interrupt types demand immediate attention even if other interrupt types being processed have not had the opportunity to save the machine state (that is, return address and captured state of the MSR). To enable taking a critical interrupt immediately after a noncritical interrupt is taken (that is, before the machine state is saved), two sets of save/restore register pairs are provided. Critical interrupts use CSRR0/CSRR1, and noncritical interrupts use SRR0/SRR1.
- Asynchronous/synchronous. Asynchronous interrupts are caused by events external to instruction execution; synchronous interrupts are caused by instruction execution and are either precise or imprecise.

Table 5-5 describes asynchronous and synchronous interrupts.

Table 5-5. Asynchronous and Synchronous Interrupts

Class	Description
Asynchronous	Caused by events independent from instruction execution. For asynchronous interrupts, the address reported to the interrupt handling routine is the address of the instruction that would have executed next, had the asynchronous interrupt not occurred.
Synchronous, Precise	Caused directly by instruction execution. Synchronous interrupts are precise or imprecise. These interrupts precisely indicate the address of the instruction causing the exception or, for certain synchronous, precise interrupt types, the address of the immediately following instruction. When the execution or attempted execution of an instruction causes a synchronous, precise interrupt, the following conditions exist at the interrupt point: • Whether SRR0 or CSRR0 addresses the instruction causing the exception or the next instruction is determined by the interrupt type and status bits. • An interrupt is generated such that all instructions before the instruction causing the exception appear to have completed with respect to the executing processor. However, some accesses associated with these preceding instructions may not have been performed with respect to other processors and mechanisms. • The exception-causing instruction may appear not to have begun execution (except for causing the exception), may be partially executed, or may have completed, depending on the interrupt type. See Section 5.9, “Partially Executed Instructions.” • Architecturally, no instruction beyond the exception-causing instruction executed.

Table 5-5. Asynchronous and Synchronous Interrupts (continued)

Class	Description
Synchronous, Imprecise	Imprecise interrupts may indicate the address of the instruction causing the exception that generated the interrupt or some instruction after that instruction. When execution or attempted execution of an instruction causes an imprecise interrupt, the following conditions exist at the interrupt point. • SRR0 or CSRR0 addresses either the exception-causing instruction or some instruction following the exception-causing instruction that generated the interrupt. • An interrupt is generated such that all instructions preceding the instruction addressed by SRR0 or CSRR0 appear to have completed with respect to the executing processor. • If context synchronization forces the imprecise interrupt due to an instruction that causes another exception that generates an interrupt (for example, alignment or data storage interrupt), SRR0 addresses the interrupt-forcing instruction, which may have partially executed (see Section 5.9, "Partially Executed Instructions"). • If execution synchronization forces an imprecise interrupt due to an execution-synchronizing instruction other than **msync** or **isync**, SRR0 or CSRR0 addresses the interrupt-forcing instruction, which appears not to have begun execution (except for its forcing the imprecise interrupt). If the interrupt is forced by **msync** or **isync**, SRR0 or CSRR0 may address **msync** or **isync**, or the following instruction. • If context or execution synchronization forces an imprecise interrupt, the instruction addressed by SRR0 or CSRR0 may have partially executed (see Section 5.9, "Partially Executed Instructions"). No instruction following the instruction addressed by SRR0 or CSRR0 has executed.

5.5.1 Requirements for System Reset Generation

Book E does not specify a system reset interrupt as was defined in the AIM version of the PowerPC architecture. On the e500, a system reset is initiated in one of the following ways:

- By asserting $\overline{hreset}$, which resets the internal state of the core complex
- By writing a 1 to DBCR0[34], if MSR[DE] = 1

5.6 Interrupt Processing

Associated with each kind of interrupt is an interrupt vector, the address of the initial instruction that is executed when an interrupt occurs.

Interrupt processing consists of saving a small part of the processor's state in certain registers, identifying the cause of the interrupt in another register, and continuing execution at the corresponding interrupt vector location. When an exception exists that causes an interrupt to be generated and it has been determined that the interrupt can be taken, the following steps are performed:

1. SRR0 (for noncritical class interrupts) or CSRR0 (for critical class interrupts) or MCSRR0 for machine check interrupts is loaded with an instruction address that depends on the type of interrupt; see the specific interrupt description for details.
2. The ESR or MCSR is loaded with information specific to the exception type. Note that many interrupt types can only be caused by a single type of exception event, and thus do not need nor use an ESR setting to indicate the cause of the interrupt.

3. SRR1 (for noncritical class interrupts) or CSRR1 (for critical class interrupts) or MCSRR1 for machine check interrupts is loaded with a copy of the MSR contents.
4. New MSR values take effect beginning with the first instruction following the interrupt. The MSR is updated as follows:
   — MSR[SPE,WE,EE,PR,FP,FE0,FE1,IS,DS] are cleared by all interrupts.
   — MSR[CE,DE] are cleared by critical class interrupts and unchanged by noncritical class interrupts.
   — MSR[ME] is cleared by machine check interrupts and unchanged by other interrupts.
   — Other defined MSR bits are unchanged by all interrupts.

   MSR fields are described in Section 2.5.1, “Machine State Register (MSR).”
5. Instruction fetching and execution resumes, using the new MSR value, at a location specific to the interrupt type (IVPR[32–47] || IVOR*n*[48–59] || 0b0000)

   The IVOR*n* for the interrupt type is described in Table 5-6. IVPR and IVOR contents are indeterminate upon reset and must be initialized by system software.

Interrupts do not clear reservations obtained with load and reserve instructions. The operating system should do so at appropriate points, such as at process switch.

At the end of a noncritical interrupt handling routine, executing **rfi** causes the MSR to be restored from SRR1 and instruction execution to resume at the address contained in SRR0. Likewise, **rfci** and **rfmci** perform the same function at the end of critical and machine check interrupt handling routines respectively, using the critical and machine check save/restore registers.

NOTE

In general, at process switch, due to possible process interlocks and possible data availability requirements, the operating system needs to consider executing the following:

- **stwcx.**—Clear outstanding reservations to prevent pairing a **lwarx** in the old process with a **stwcx.** in the new one
- **msync**—Ensure that memory operations of an interrupted process complete with respect to other processors before that process begins executing on another processor
- **rfi**, **rfci**, **rfmci**, or **isync**—Ensure that instructions in the new process execute in the new context

5.7 Interrupt Definitions

Table 5-6 summarizes each interrupt type, the various exception types that may cause that interrupt, the interrupt classification, which ESR bits can be set, which MSR bits can mask the interrupt type, and which IVOR is used to specify the vector address.

Table 5-6. Interrupt and Exception Types

IVOR	Interrupt Type	Exception Type	Exception Class[1]	ESR[2]	Mask Bits	Notes	Page
IVOR0	Critical input	Critical input	A, C	—	MSR[CE]	3	5-13
IVOR1	Machine check	Machine check	C	—	MSR[ME]	4,5	5-14
IVOR2	Data storage (DSI)	Access	SP	[SPE],[ST]	—	6	5-19
		Load reserve or store conditional to write- through required location (W = 1)	SP	[ST]	—	6	
		Cache locking	SP	[DLK,ILK],[ST]	—	7	
		Byte ordering	SP	[ST],BO	—	—	
IVOR3	Instruction storage (ISI)	Access	SP	—	—	—	5-20
		Byte ordering	SP	BO	—	—	
IVOR4	External input		A	—	MSR[EE]	3	5-21
IVOR5	Alignment		SP	[ST],[SPE,ST]	—	—	5-22
IVOR6	Program	Illegal	SP	PIL	—	—	5-24
		Privileged	SP	PPR	—	—	
		Trap	SP	PTR	—	—	
IVOR8	System call		SP	—	—	—	5-25
IVOR10	Decrementer		A	—	MSR[EE], TCR[DIE]	—	5-25
IVOR11	Fixed interval timer		A	—	MSR[EE], TCR[FIE]	—	5-26
IVOR12	Watchdog		A, C	—	MSR[CE], TCR[WIE]	—	5-27
IVOR13	Data TLB error	Data TLB miss	SP	[SPE],[ST]	—	—	5-27
IVOR14	Instruction TLB error	Instruction TLB miss	SP	—	—	—	5-29
IVOR15	Debug	Trap (synchronous)	A, SP, C	—	MSR[DE], DBCR0[IDM]	—	5-30
		Instruction address compare (synchronous)	A, SP, C	—	MSR[DE], DBCR0[IDM]	—	
		Data address compare (synchronous)	A, SP, C	—	MSR[DE], DBCR0[IDM]	—	
		Instruction complete	SP, C	—	MSR[DE], DBCR0[IDM]	8	
		Branch taken	SP, C	—	MSR[DE], DBCR0[IDM]	8	
		Return from interrupt	SP, C	—	MSR[DE], DBCR0[IDM]	—	
		Interrupt taken	SI, C	—	MSR[DE], DBCR0[IDM]	—	
		Unconditional debug event	SI, C	—	MSR[DE], DBCR0[IDM]	—	

Table 5-6. Interrupt and Exception Types (continued)

IVOR	Interrupt Type	Exception Type	Exception Class[1]	ESR[2]	Mask Bits	Notes	Page
IVOR32	SPE/ embedded floating-point APU unavailable	SPE/embedded floating-point APU unavailable	SP	—	—	9	5-31
IVOR33	Embedded floating-point data	Embedded floating-point data exception	SP	—	—	9	5-32
IVOR34	Embedded floating-point round	Embedded floating-point round exception	SP	—	—	9	5-32

1 A = asynchronous, C = critical, SI = synchronous, imprecise, SP = synchronous, precise

2 In general, when an interrupt causes an ESR bit or bits to be set (or cleared) as indicated in the table, it also causes all other ESR bits to be cleared. Special rules may apply for implementation-specific ESR bits

Legend:

xxx (no brackets) means ESR[xxx] is set.
[xxx] means ESR[xxx] could be set.
[xxx,yyy] means either ESR[xxx] or ESR[yyy] may be set, but never both.
{xxx,yyy} means either ESR[xxx] or ESR[yyy] may be set, or possibly both.

3 Although not part of Book E, system interrupt controllers commonly provide independent mask and status bits for critical input and external input interrupt sources.

4 Machine check interrupts are not asynchronous or synchronous. See Section 5.7.2, “Machine Check Interrupt.”

5 Machine check status information is commonly provided as part of the system implementation but is not part of Book E.

6 Software must examine the instruction and the subject TLB entry to determine the exact cause of the interrupt.

7 Cache locking and cache locking exceptions are implementation-dependent.

8 Instruction complete and branch taken debug events are defined only for MSR[DE] = 1 for internal debug mode (DBCR0[IDM] = 1). In other words, for internal debug mode with MSR[DE] = 0, instruction complete and branch taken debug events cannot occur, no DBSR status bits are set, and no subsequent imprecise debug interrupt can occur.

9 EIS-defined exception

5.7.1 Critical Input Interrupt

A critical input interrupt occurs when no higher priority exception exists, a critical input exception is presented to the interrupt mechanism, and MSR[CE] = 1. The specific definition of a critical input exception is implementation-dependent but is typically caused by assertion of an asynchronous signal that is part of the system. In addition to MSR[CE], implementations may provide other ways to mask the critical input interrupt.

CSRR0, CSRR1, and MSR are updated as shown in Table 5-7.

Table 5-7. 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 is unchanged. All other MSR bits are cleared.

Instruction execution resumes at address IVPR[32–47] || IVOR0[48–59] || 0b0000.

On the e500, to guarantee that the core complex can take a critical input interrupt, the critical input interrupt signal must be asserted until the interrupt is taken. Otherwise, whether the core complex takes an external interrupt depends on whether MSR[CE] is set when the critical interrupt signal is asserted.

NOTE

To avoid redundant critical input interrupts, software must take any actions required by the implementation to clear any critical input exception status before reenabling MSR[CE].

5.7.2 Machine Check Interrupt

The EIS defines the machine check APU, which differs from the Book E definition of the machine check interrupt as follows:

- Book E defines machine check interrupts as critical interrupts, but the machine check APU treats them as a distinct interrupt type.
- Machine check is no longer a critical interrupt but uses MCSRR0 and MCSRR1 to save the return address and the MSR in case the machine check is recoverable.
- Return From Machine Check Interrupt instruction (**rfmci**) is implemented to support the return to the address saved in MCSRR0.
- An address related to the machine check may be stored in MCAR, according to Table 5-10.
- A machine check syndrome register, MCSR, is used to log the cause of the machine check (instead of ESR). The MCSR is described in Table 5-4.

The following general information applies to both the Book E and EIS definitions. A machine check interrupt occurs when no higher priority exception exists, a machine check exception is presented to the interrupt mechanism, and MSR[ME] = 1. Specific causes of machine check exceptions are implementation-dependent, as are the details of the actions taken on a machine check interrupt.

Machine check interrupts are typically caused by a hardware or memory subsystem failure or by an attempt to access an invalid address. They may be caused indirectly by execution of an instruction, but may not be recognized or reported until long after the processor has executed past the instruction that caused the machine check. As such, machine check interrupts are not thought of as synchronous or asynchronous nor as precise or imprecise.

The following general rules apply:

- No instruction after the one whose address is reported to the machine check interrupt handler in MCSRR0 has begun execution.
- The instruction whose address is reported to the machine check interrupt handler in MCSRR0 and all prior instructions may or may not have completed successfully. All instructions certain to complete appear to have done so within the context existing before the machine check interrupt. No further interrupts (other than possible additional machine check interrupts) occur as a result of those instructions.

e500 machine check exceptions are specified in Table 5-8.

Table 5-8. e500 Machine Check Exception Sources

Source	Signal	Additional Enable Bits
Negative edge on machine check signal ($\overline{mcp}$)	$\overline{mcp}$	HID0[EMCP]
Data cache parity error	*dcperr*	L1CSR0[CPE]
Instruction cache parity error	*icperr*	L1CSR1[ICPE]
Data cache push parity error	*dcp_perr*	L1CSR0[CPE]
Bus instruction address error	*bus_iaerr*	No enable bit
Bus read address error	*bus_raerr*	No enable bit
Bus write address error	*bus_waerr*	No enable bit
Bus instruction data bus error	*bus_iberr*	No enable bit
Read data bus error	*bus_rberr*	No enable bit
Write bus error	*bus_wberr*	No enable bit
Instruction parity error	*bus_iperr*	HID1[R1DPE], HID1[R2DPE] (depending on which bus the instruction fetch arrived)
Read parity error	*bus_rperr*	HID1[R1DPE], HID1[R2DPE] (on whichever bus the data read arrived)
Bus fault	*core_fault_in*	HID1[RFXE] = 1. This interrupt should not occur during normal operation because RFXE should be zero and such errors shold be reported instead by peripherals as external interrupts or critical interrupts. For information about bus faults, see Section 13.8, "Proper Reporting of Bus Faults." For additional information, see Section 2.10.2, "Hardware Implementation-Dependent Register 1 (HID1)."

If MSR[ME] is cleared, the processor enters checkstop state immediately on detecting the machine check condition.

When a machine check interrupt is taken, registers are updated as shown in Table 5-9.

Table 5-9. Machine Check Interrupt Settings

Register	Setting
MCSRR0	On a best-effort basis, the core complex sets this to an effective address of some instruction that was executing or about to be executing when the machine check condition occurred.
MCSRR1	MSR[37–38,46–55,57–59,61–63] are loaded with equivalent MSR bits. All other bits are reserved.
MCAR	When a machine check interrupt is taken, the machine check address register is updated with the address of the data associated with the machine check. Note that if a machine check interrupt is caused by a signal, the MCAR contents are not meaningful. See Section 2.7.2.3, “Machine Check Address Register (MCAR).”
MCSR	Set according to the machine check condition. See Table 5-4.

Instruction execution resumes at address IVPR[32–47] || IVOR1[48–59] || 0b0000.

NOTES

If a machine check interrupt is caused by a memory subsystem error, the subsystem may return incorrect data, which may be placed into registers or on-chip caches.

For implementations on which a machine check interrupt is caused by referring to an invalid physical address, executing **dcbz** or **dcba** can cause a delayed machine check interrupt by establishing a data cache block associated with an invalid physical address. A machine check interrupt can occur later if and when an attempt is made to write that block to main memory, for example as the result of executing an instruction that causes a cache miss for which the block is the target for replacement or as the result of executing **dcbst** or **dcbf**.

5.7.2.1 Core Complex Bus (CCB) and L1 Cache Machine Check Errors

This section describes machine checks caused by bus and L1 cache errors. It describes error signaling and detection, and it contains information about error recoverability.

The L1 caches in the e500 core complex are protected by parity. Parity information is written into the L1 caches when one of the following occurs:

- A store instruction, **dcbz**, or **dcba** modifies the data cache.
- A line fill occurs into the instruction or data cache.

L1 cache parity is checked when one of the following occurs:

- A load instruction hits in the L1 data cache.
- An instruction fetch hits in the L1 instruction cache.
- A line is cast out of the L1 data cache.

For loads that hit in the cache, parity is enforced at a double-word granularity. So, if a byte load lies within a double word that contains a parity error, an interrupt is generated. These interrupts do not occur if the load is on a speculative path and never completes.

L1 cache parity checking is disabled by default and can be enabled by setting L1CSR0[CPE] and L1CSR1[ICPE].

The e500's core complex bus (CCB) is also protected by parity. Parity is checked whenever data is read on either of the two CCB read buses; a machine check interrupt is generated if errors occur. Parity is also generated whenever data is written on the CCB write bus, giving an opportunity to identify and report errors when data is cast out of the cache or written with a cache-inhibited or write-through store. For cache pushes (or castouts), a parity error is generated if there is any bad parity on the cache line.

For bus reads, a parity error occurs whenever bad data is read on the bus, regardless of whether the data is ever used. CCB read bus parity checking is disabled by default and is enabled by setting HID1[R1DPE] and HID1[R2DPE].

Table 5-10 is an expanded list of the scenarios listed above. For each scenario, there is a list of what kind of machine check error can occur as indicated by the MCSR bit that is set. For each condition, the table provides comments about recoverability, whether the MCAR has the address of the bad data, whether the exception is precise, and how far corrupted data can go into the GPRs, cache, or memory.

Table 5-10. Parity Error Exception Scenarios

Scenario	MCSR Bit	Description	MCSRR0 and MCAR Values	Comments
Load (cache hit)	DCPERR	Detection of a data cache parity error	MCSRR0 has the instruction address of the failing load instruction. MCAR is not set.	Data does not get into GPR.
Store (cache hit)	No cases to consider			
Load (cache miss or cache inhibited)	BUS_RAERR	Address bus error	MCSRR0 points to some instruction near the failing load. MCAR is set to an address on the cache line with the error.	Data does not get into GPR. Line-fill data does not get into L1 cache (if cacheable).
	BUS_RBERR	Read data bus error		
	BUS_RPERR	Detection of a read data bus parity error		
Store (cache miss)	BUS_RAERR	Address bus error	MCSRR0 points to some instruction after the failing store. (It is not particularly meaningful.) MCAR is set to an address on the cache line with the error.	Line-fill data does not get into L1 cache. (Stores to that line may be lost.)
	BUS_RBERR	Read data bus error		
	BUS_RPERR	Detection of read data bus parity error		
Store (cache-inhibited or write-through)	BUS_WAERR	Address bus error	MCSRR0 points to some instruction near the failing store. (It is not particularly meaningful.) MCAR is set to an address on the cache line with the error.	The system has enough information to prevent memory corruption.
	BUS_WBERR	Write data bus error signaled by assertion of *core_wr_errin_b* input		

Table 5-10. Parity Error Exception Scenarios (continued)

Scenario	MCSR Bit	Description	MCSRR0 and MCAR Values	Comments
Castout or snoop push	BUS_WAERR	Address bus error	MCSRR0 is not meaningful. MCAR is set to an address on the cache line with the error.	The system has enough information to prevent memory corruption. A front-side L2 does not cache the bad data.
	BUS_WBERR	Write data bus error signaled by assertion of *core_wr_errin_b* input		
	DCP_PERR	Detection of an L1 data cache parity error in the data being pushed		A front-side L2 does not cache the bad data. The system has enough information to prevent memory corruption.
Instruction fetch (cache hit)	ICPERR	Detection of an L1 instruction cache parity error	MCSRR0 has an address on the line of the failing instruction. MCAR is not set.	The instruction that causes the exception is not executed.
Instruction fetch (cache miss or cache inhibited)	BUS_IAERR	Address bus error	MCSRR0 has an address on the line of the failing instruction. MCAR is set to an address on the cache line with the error.	The instruction that causes the exception is not executed. Line-fill data does not get into the L1 cache.
	BUS_IBERR	Read data bus error		
	BUS_IPERR	Detection of a read data bus parity error		

5.7.2.2 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[ICPI] is set, any instruction cache line fill has all of its parity bits inverted in the instruction cache.
- If L1CSR0[CPI] is set, any data line fill has all of its 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**, and **dcba**.

NOTE

L1 cache parity checking for the instruction cache must be enabled (L1CSR1[ICPE] = 1,) when L1CSR1[ICPI] is set. Similarly for the data cache, L1CSR0[CPE] must be set if L1CSR0[CPI] is set. If the programmer attempts to set the field L1CSR0[CPI] (using **mtspr**) without setting the field L1CSR0[CPE], then the field L1CSR0[CPI] will not be set. If the programmer attempts to set the field L1CSR1[ICPI] without setting the field L1CSR1[ICPE], then the field L1CSR1[ICPI] will not be set.

5.7.3 Data Storage Interrupt

A data storage interrupt (DSI) occurs when no higher priority exception exists and a data storage exception is presented to the interrupt mechanism. Table 5-11 describes exception conditions for a data storage interrupt as defined by Book E.

Table 5-11. Data Storage Interrupt Exception Conditions

Exception	Cause
Read access control exception	Occurs when either of the following conditions exists: • In user mode (MSR[PR] = 1), a load or load-class cache management instruction attempts to access a memory location that is not user-mode read enabled (page access control bit UR = 0). • In supervisor mode (MSR[PR] = 0), a load or load-class cache management instruction attempts to access a location that is not supervisor-mode read enabled (page access control bit SR = 0).
Write access control exception	Occurs when either of the following conditions exists: • In user mode (MSR[PR] = 1), a store or store-class cache management instruction attempts to access a location that is not user-mode write enabled (page access control bit UW = 0). • In supervisor mode (MSR[PR] = 0), a store or store-class cache management instruction attempts to access a location that is not supervisor-mode write enabled (page access control bit SW = 0).
Byte-ordering exception	The implementation cannot access data in the byte order specified by the page's endian attribute. Note: The byte-ordering 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 e500, load/store accesses that cross a page boundary such that endianness changes cause a byte-ordering exception.
Cache locking exception	(EIS) The locked state of one or more cache lines has the potential to be altered. This exception is implementation-dependent. A cache locking exception occurs with the execution of **icbtls**, **icblc**, **dcbtls**, **dcbtstls**, or **dcblc** when (MSR[PR] = 1) and (MSR[UCLE] = 0). ESR is set as follows: • For **icbtls** and **icblc**, ESR[ILK] is set. • For **dcbtls**, **dcbtstls**, or **dcblc**, ESR[DLK] is set. Book E refers to this as a cache-locking exception.
Storage synchronization exception	Occurs when either of the following conditions exists: • An attempt is made to execute a load and reserve or store conditional instruction from or to a location that is write-through required or caching inhibited. (If the interrupt does not occur, the instruction executes correctly.) • A store conditional instruction produces an effective address for which a normal store would cause a data storage interrupt but the processor does not have the reservation from a load and reserve instruction. Book E states that it is implementation-dependent whether a data storage interrupt occurs. The EIS defines that the data storage interrupt is taken. See the section, "Atomic Update Primitives Using **lwarx** and **stwcx.**," in the "Instruction Model" chapter of the EREF.

icbt, **dcbt**, **dcbtst**, and **dcba** instructions cannot cause a data storage interrupt, regardless of the effective address.

NOTE

icbi and **icbt** are treated as loads from the addressed byte with respect to address translation and protection. They use MSR[DS], not MSR[IS], to determine translation for their operands. Instruction storage interrupts and instruction TLB error interrupts are associated with instruction fetching and not execution. Data storage interrupts and data TLB error interrupts are associated with the execution of instruction cache management instructions.

When a data storage interrupt occurs, the processor suppresses execution of the instruction causing the data storage exception. SRR0, SRR1, ESR, MSR, and DEAR, are updated as follows:

Table 5-12. Data Storage Interrupt Register Settings

Register	Setting
SRR0	Set to the effective address of the instruction causing the interrupt
SRR1	Set to the MSR contents at the time of the interrupt
ESR	ST Set if the instruction causing the interrupt is a store or store-class cache management instruction; otherwise cleared DLK DLK is set when a DSI occurs because **dcbtls**, **dcbtstls**, or **dcblc** is executed in user mode and MSR[UCLE] = 0. BO Set if the instruction caused a byte-ordering exception; otherwise cleared All other defined ESR bits are cleared.
MSR	CE, ME, and DE are unchanged. All other MSR bits are cleared.
DEAR	Set to the effective address 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 IVPR[32–47] || IVOR2[48–59] || 0b0000.

5.7.4 Instruction Storage Interrupt

An instruction storage interrupt occurs when no higher priority exception exists and an instruction storage exception is presented to the interrupt mechanism. Instruction storage exception conditions are described in Table 5-13.

Table 5-13. Instruction Storage Interrupt Exception Conditions

Exception	Cause
Execute access control exception	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). 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).
Byte-ordering exception	The implementation cannot fetch the instruction in the byte order specified by the page's endian attribute. The EIS defines that accesses that cross a page boundary such that endianness changes cause a byte-ordering exception.

Note that Book E provides this exception to assist implementations that cannot dynamically switch byte ordering between consecutive accesses, do not support the byte order for a class of accesses, or do not support misaligned accesses using a specific byte order.

When an instruction storage interrupt occurs, the processor suppresses execution of the instruction causing the exception.

SRR0, SRR1, MSR, and ESR are updated as shown in Table 5-14.

Table 5-14. Instruction Storage Interrupt Register Settings

Register	Setting
SRR0	Set to the effective address of the instruction causing the instruction storage interrupt
SRR1	Set to the MSR contents at the time of the interrupt
MSR	CE, ME, and DE are unchanged. All other MSR bits are cleared.
ESR	BO is set if the instruction fetch caused a byte-ordering exception; otherwise cleared. All other defined ESR bits are cleared.

NOTE

Permissions violations and byte-ordering exceptions are not mutually exclusive. Even if ESR[BO] is set, system software must examine the TLB entry accessed by the fetch to determine whether a permissions violation also may have occurred.

Instruction execution resumes at address IVPR[32–47] || IVOR3[48–59] || 0b0000.

5.7.5 External Input Interrupt

An external input interrupt occurs when no higher priority exception exists, an external input exception is presented to the interrupt mechanism, and MSR[EE] = 1. The specific definition of an external input exception is implementation-dependent and is typically caused by assertion of an asynchronous signal that is part of the processing system. On the e500, this is the external interrupt signal.

To guarantee that the core complex can take an external interrupt, the external interrupt pin must be asserted until the interrupt is taken. Otherwise, whether the external interrupt is taken depends on whether MSR[EE] is set when the external interrupt signal is asserted.

In addition to MSR[EE], implementations may provide other ways to mask this interrupt. The e500 does not support additional masking mechanisms.

SRR0, SRR1, and MSR are updated as shown in Table 5-15.

Table 5-15. External Input 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	CE, ME, and DE are unchanged. All other MSR bits are cleared.

Instruction execution resumes at address IVPR[32–47] || IVOR4[48–59] || 0b0000.

NOTE

To avoid redundant external input interrupts, software must take any actions required to clear any external input exception status before reenabling MSR[EE].

5.7.6 Alignment Interrupt

An alignment interrupt occurs when no higher priority exception exists and an alignment exception is presented to the interrupt mechanism. An alignment exception may occur when an implementation cannot perform a data access for one of the following reasons:

- The operand of a load or store is not aligned.
- The instruction is a move assist, load multiple, or store multiple.
- A **dcbz** operand is in write-through-required or caching-inhibited memory, or **dcbz** is executed in an implementation with no data cache or a write-through data cache.
- The operand of a store, except store conditional, is in write-through required memory.

The EIS defines the following alignment exception conditions:

- Execution of a **dcbz** references a page marked as write-through or cache inhibited.
- A load multiple word instruction (**lmw**) reads an address that is not a multiple of four.
- A **lwarx** or **stwcx**. instruction references an address that is not a multiple of four.
- SPFP and SPE APU instructions are not aligned on a natural boundary. A natural boundary is defined by the size of the data element being accessed.
- A vector operation reports an exception if the physical address of the following instructions is not aligned to the 64-bit boundary: **evldd**, **evlddx**, **evldw**, **evldwx**, **evldh**, **evldhx**, **evstdd**, **evstddx**, **evstdw**, **evstdwx**, **evstdh**, and **evstdhx**. Table 5-16 describes additional ESR settings.

For **lmw** and **stmw** with a non–word-aligned operand and for load and reserve and store conditional instructions with an misaligned operand, an implementation may yield boundedly undefined results instead of causing an alignment interrupt. A store conditional to a write-through-required location may either cause an alignment or data storage interrupt or may correctly execute the instruction. For all other cases listed above, an implementation may execute the instruction correctly instead of causing an alignment interrupt. For **dcbz**, correct execution means clearing each byte of the block in main memory.

NOTE

Book E does not support use of a misaligned effective address by load and reserve and store conditional instructions. If a misaligned effective address is specified, the alignment interrupt handler should treat the instruction as a programming error and must not attempt to emulate the instruction.

When an alignment interrupt occurs, the processor suppresses the execution of the instruction causing the alignment exception.

SRR0, SRR1, MSR, DEAR, and ESR are updated as shown in Table 5-16.

Table 5-16. Alignment Interrupt Register Settings

Register	Setting
SRR0	Set to the effective address of the instruction causing the alignment interrupt
SRR1	Set to the MSR contents at the time of the interrupt
MSR	CE, ME, and DE are unchanged. All other MSR bits are cleared.
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 alignment exception
ESR	The following bits may be affected for vector alignment exception conditions: SPE Set ST Set only if the instruction causing the exception is a store and is cleared for a load All other defined ESR bits are cleared.

Instruction execution resumes at address IVPR[32–47] || IVOR5[48–59] || 0b0000.

5.7.7 Program Interrupt

A program interrupt occurs when no higher priority exception exists and a program exception is presented to the interrupt mechanism. A program interrupt is caused when any of the following exceptions occurs during execution of an instruction.

Table 5-17. Program Interrupt Exception Conditions

Exception	Cause
Illegal instruction exception	An illegal instruction exception always occurs when execution of any of the following kinds of instructions is attempted. • A reserved-illegal instruction • In user mode, an **mtspr** or **mfspr** that specifies an SPRN value with SPRN[5] = 0 (user-mode accessible) that represents an unimplemented SPR • (EIS) If an invalid SPR address is accessible only in supervisor mode and the processor is in supervisor mode (MSR[PR] = 0), results are undefined. • (EIS) If the invalid SPR address is accessible only in the supervisor mode and the processor is in user mode (MSR[PR] = 1), a privileged instruction exception is taken. An illegal instruction exception may occur when execution is attempted of any of the following kinds of instructions. If the exception does not occur, the alternative is shown in parentheses. • An instruction that is in invalid form (boundedly undefined results). On the e500, all instructions have invalid forms cause boundedly undefined results. • A reserved no-op instruction (no-operation performed is preferred). There are no reserved no-ops for the e500. • A defined or allocated instruction that is not implemented (unimplemented operation exception). Unimplemented Book E instructions such as **mfapidi**, **mfdcr**, and **mtdcr** take an illegal instruction exception. • The EIS defines that an attempt to execute a 64-bit Book E instruction causes an illegal instruction exception.
Privileged instruction exception	Occurs when MSR[PR] = 1 and execution is attempted of any of the following: • A privileged instruction • An **mtspr** or **mfspr** instruction that specifies a privileged SPR (SPRN[5] = 1) • (EIS) An **mtpmr** or **mfpmr** instruction that specifies a privileged PMR (PMRN[5] = 1)
Trap exception	A trap exception occurs when any of the conditions specified in a trap instruction are met.
Unimplemented operation exception	An unimplemented operation exception may occur when a defined or allocated instruction is encountered that is not implemented. Otherwise an illegal instruction exception occurs. On the e500, these instructions are **mfapidi**, **mfdcr**, and **mtdcr** and they take an illegal instruction exception.

SRR0, SRR1, MSR, and ESR are updated as shown in Table 5-18.

Table 5-18. Program Interrupt Register Settings

Register	Description
SRR0	For all program interrupts except an enabled exception when in an imprecise mode (see Table 5-19), set to the EA of the instruction that caused the interrupt.
SRR1	Set to the MSR contents at the time of the interrupt.

Table 5-18. Program Interrupt Register Settings (continued)

Register	Description
MSR	CE, ME, and DE are unchanged. All other MSR bits are cleared.
ESR	PIL Set if an illegal instruction exception-type program interrupt; otherwise cleared. PPR Set if a privileged instruction exception-type program interrupt; otherwise cleared. PTR Set if a trap exception-type program interrupt; otherwise cleared. All other defined ESR bits are cleared.

Instruction execution resumes at address IVPR[32–47] || IVOR6[48–59] || 0b0000.

5.7.8 System Call Interrupt

A system call interrupt occurs when no higher priority exception exists and a System Call (**sc**) instruction is executed. SRR0, SRR1, and MSR are updated as shown in Table 5-19.

Table 5-19. System Call Interrupt Register Settings

Register	Description
SRR0	Set to the effective address of the instruction after the **sc** instruction.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	CE, ME, and DE are unchanged. All other MSR bits are cleared.

Instruction execution resumes at address IVPR[32–47] || IVOR8[48–59] || 0b0000.

5.7.9 Decrementer Interrupt

A decrementer interrupt occurs when no higher priority exception exists, a decrementer exception exists (TSR[DIS] = 1), and the interrupt is enabled (TCR[DIE] = 1 and MSR[EE] = 1).

NOTE

MSR[EE] also enables external input and fixed-interval timer interrupts.

SRR0, SRR1, MSR, and TSR are updated as shown in Table 5-20.

Table 5-20. 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	CE, ME, and DE are unchanged. All other MSR bits are cleared.
TSR	DIS is set.

Instruction execution resumes at address IVPR[32–47] || IVOR10[48–59] || 0b0000.

NOTE

To avoid redundant decrementer interrupts, before reenabling MSR[EE], the interrupt handling routine must clear TSR[DIS] by writing a word to TSR using **mtspr** with a 1 in any bit position to be cleared and 0 in all others. The data written to the TSR is not direct data, but a mask. Writing a 1 to this bit causes it to be cleared; writing a 0 has no effect.

5.7.10 Fixed-Interval Timer Interrupt

A fixed-interval timer interrupt occurs when no higher priority exception exists, a fixed-interval timer exception exists (TSR[FIS] = 1), and the interrupt is enabled (TCR[FIE] = 1 and MSR[EE] = 1). The "Timers" chapter in the EREF describes Book E and EIS aspects 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],TCR[FP] = 000000 selects TBU[32]. TCR[FPEXT],TCR[FP] = 111111 selects TBL[63].

NOTE

MSR[EE] also enables external input and decrementer interrupts.

SRR0, SRR1, MSR, and TSR are updated as shown in Table 5-21.

Table 5-21. Fixed-Interval Timer 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	CE, ME, and DE are unchanged. All other MSR bits are cleared.
TSR	FIS is set.

Instruction execution resumes at address IVPR[32–47] || IVOR11[48–59] || 0b0000.

NOTE

To avoid redundant fixed-interval timer interrupts, before reenabling MSR[EE], the interrupt handling routine must clear TSR[FIS] by writing a word to TSR using **mtspr** with a 1 in any bit position to be cleared and 0 in all others. The data written 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.

5.7.11 Watchdog Timer Interrupt

A watchdog timer interrupt occurs when no higher priority exception exists, a watchdog timer exception exists (TSR[WIS] = 1), and the interrupt is enabled (TCR[WIE] = 1 and MSR[CE] = 1). The "Timers" chapter in the EREF describes Book E and EIS aspects of the watchdog timer.

NOTE

MSR[CE] also enables the critical input interrupt.

CSRR0, CSRR1, MSR, and TSR are updated as shown in Table 5-22.

Table 5-22. Watchdog Timer 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 is unchanged; all other MSR bits are cleared.
TSR	WIS is set.

Instruction execution resumes at address IVPR[32–47] || IVOR12[48–59] || 0b0000.

NOTE

To avoid redundant watchdog timer interrupts, before reenabling MSR[CE], the interrupt handling routine must clear TSR[WIS] by writing a word to TSR using **mtspr** with a 1 in any bit position to be cleared and 0 in all others. The data written to the TSR is not direct data, but a mask. Writing a 1 to this bit causes it to be cleared; writing a 0 has no effect.

5.7.12 Data TLB Error Interrupt

A data TLB error interrupt occurs when no higher priority exception exists and the exception described in Table 5-23 is presented to the interrupt mechanism.

Table 5-23. Data TLB Error Interrupt Exception Conditions

Exception	Description
Data TLB miss exception	Virtual addresses associated with a data fetch do not match any valid TLB entry.

If a store conditional instruction produces an effective address 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, Book E defines it as implementation-dependent whether a data TLB error interrupt occurs. The EIS defines that the interrupt is taken.

When a data TLB error interrupt occurs, the processor suppresses execution of the instruction causing the data TLB error exception.

SRR0, SRR1, MSR, DEAR, and ESR are updated as shown in Table 5-24.

Table 5-24. Data TLB Error Interrupt Register Settings

Register	Setting
SRR0	Set to the effective address of the instruction causing the data TLB error interrupt.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	CE, ME, and DE are unchanged. All other MSR bits are cleared.
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 data TLB error exception.
ESR	ST Set if the instruction causing the interrupt is a store, **dcbi**, or **dcbz** instruction; otherwise cleared. All other defined ESR bits are cleared.
MAS*n*	See Table 5-25.

Table 5-25 shows MAS register settings for data and instruction TLB error interrupts as implemented on the e500. The "Cache and MMU Background" chapter of the EREF describes how these values are set as defined by the EIS.

Table 5-25. MMU Assist Register Field Updates for TLB Error Interrupts

MAS Register Bit/Field	Value Loaded for Each Case
TLBSEL	TLBSELD
ESEL	if TLBSELD = 0: TLB0[NV] else, undefined
NV	if TLBSELD = 0: ¬TLB0[NV] else, undefined
V	1
IPROT	0
TID[0–7]	Value of PID register selected by TIDSELD
TS	MSR[IS/DS]
TSIZE[0–3]	TSIZED
EPN[32–51]	EPN of access
X0, X1 W, I, M, G, E	X0D, X1D WD, ID, MD, GD, ED
RPN[32–51]	Zeros
PERMIS	Zeros
TLBSELD	—
TIDSELD[0–1]	—
TSIZED[0–3]	—

Table 5-25. MMU Assist Register Field Updates for TLB Error Interrupts (continued)

MAS Register Bit/Field	Value Loaded for Each Case
WD, ID, MD, GD, ED	—
SPID0	PID0
SAS	MSR[IS] for instruction access; MSR[DS] for data access

Instruction execution resumes at address IVPR[32–47] || IVOR13[48–59] || 0b0000.

5.7.13 Instruction TLB Error Interrupt

An instruction TLB error interrupt occurs when no higher priority exception exists and the exception described in Table 5-26 is presented to the interrupt mechanism.

Table 5-26. Instruction TLB Error Interrupt Exception Conditions

Exception	Description
Instruction TLB miss exception	Virtual addresses associated with an instruction fetch do not match any valid TLB entry.

When an instruction TLB error interrupt occurs, the processor suppresses execution of the instruction causing the instruction TLB miss exception.

SRR0, SRR1, and MSR are updated as shown in Table 5-27.

Table 5-27. Instruction TLB Error Interrupt Register Settings

Register	Setting
SRR0	Set to the effective address of the instruction causing the instruction TLB error interrupt.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	CE, ME, and DE are unchanged. All other MSR bits are cleared.
MAS*n*	See Table 5-25.

Table 5-25 shows MAS register settings for data and instruction TLB error interrupts as implemented on the e500. The “Cache and MMU Background” chapter of the EREF describes how these values are set as defined by the EIS.

Instruction execution resumes at address IVPR[32–47] || IVOR14[48–59] || 0b0000.

5.7.14 Debug Interrupt

A debug interrupt occurs when no higher priority interrupt exists, a debug exception exists 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 “Debug Support” chapter of the EREF describes Book E and EIS aspects of the debug interrupt.

Table 5-28. Debug Interrupt Register Settings

Register	Setting
CSRR0	For debug exceptions that occur while debug interrupts are enabled (DBCR0[IDM] = 1 and MSR[DE] = 1), CSRR0 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 address of the instruction causing the debug interrupt. • For instruction complete (ICMP) debug exceptions, set to the address of the instruction that would have executed after the one that caused the debug interrupt. • For unconditional debug event (UDE) debug exceptions, set to the address of the instruction that would have executed next if the debug interrupt had not occurred. • For interrupt taken (IRPT) debug exceptions, set to the interrupt vector value of the interrupt that caused the interrupt taken debug event. • For return from interrupt (RET) debug exceptions, set to the address of the instruction that would have executed after the **rfi**, **rfci**, or **rfmci** that caused the debug interrupt. • For debug exceptions that occur while debug interrupts are disabled (DBCR0[IDM] = 0 or MSR[DE] = 0), a debug interrupt occurs at the next synchronizing event if DBCR0[IDM] and MSR[DE] are modified such that they are both set and if the debug exception status is still set in the DBSR. When this occurs, CSRR0 holds the address of the instruction that would have executed next, not the address of the instruction that modified DBCR0 or MSR and thus caused the interrupt.
CSRR1	Set to the MSR contents at the time of the interrupt.
MSR	ME is unchanged. All other MSR bits are cleared.
DBSR	Set to indicate type of debug event. (See Section 2.13.2, “Debug Status Register (DBSR).”)

Note that on the e500, if DBCR0[IDM] is cleared, no debug events occur. That is, irrespective of MSR, DBCR0, DBCR1, and DBCR2 settings, no debug events are logged in DBSR and no debug interrupts are taken. If DBCR0[IDM] is set, Book E debug mode functions as specified in Book E (according to the value of MSR[DE] and the values of DBCR0, DBCR1, and DBCR2).

The e500 core complex complies with the Book E 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.
- DVC is not supported.

CSRR0, CSRR1, MSR, and DBSR are updated as shown in Table 5-28.

Instruction execution resumes at address IVPR[32–47] || IVOR15[48–59] || 0b0000.

5.7.15 EIS-Defined Interrupts

The interrupts in this section are defined by the EIS and supported by the e500.

NOTE

The SPE APU and embedded floating-point APU functionality is implemented in all PowerQUICC III devices. However, these instructions will not be supported in devices subsequent to PowerQUICC III. Freescale Semiconductor strongly recommends that use of these instructions be confined to libraries and device drivers. Customer software that uses SPE or embedded floating-point APU instructions at the assembly level or that uses SPE intrinsics will require rewriting for upward compatibility with next-generation PowerQUICC devices.

Freescale Semiconductor offers a libmoto_e500 library that uses SPE instructions. Freescale will also provide libraries to support next-generation PowerQUICC devices.

5.7.15.1 SPE/Embedded Floating-Point APU Unavailable Interrupt

As defined by the EIS, an SPE APU unavailable interrupt is taken if MSR[SPE] is cleared and an SPE, embedded scalar double-precision (e500v2 only), or embedded vector single-precision floating-point instruction is executed. It is not used by the embedded scalar single-precision floating-point APU. However, on the e500v1, MSR[SPE] affects the SPE and both the vector and scalar single-precision floating-point APUs.

On the e500v2, MSR[SPE] affects only instructions that affect the upper and lower portions of the 64-bit GPRs, that is, instructions defined by the SPE, the vector single-precision floating-point APU, and the double-precision floating-point APUs. It does not affect scalar single-precision floating-point APU instructions.

When an SPE unavailable interrupt occurs, the processor suppresses execution of the instruction causing the interrupt. The SRR0, SRR1, MSR, and ESR registers are modified as shown in Table 5-29.

Table 5-29. SPE/Embedded Floating-Point APU Unavailable Interrupt Register Settings

Register	Setting
SRR0	Set to the effective address of the instruction causing the interrupt.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	CE, ME, and DE are unchanged. All other bits are cleared.
ESR	SPE (bit 24) is set. All other ESR bits are cleared.

Instruction execution resumes at address IVPR[32–47] || IVOR32[48–59] || 0b0000.

5.7.15.2 Embedded Floating-Point Data Interrupt

An embedded floating-point data interrupt is generated in the following cases:

- SPEFSCR[FINVE] = 1 and either SPEFSCR[FINVH,FINV] = 1
- SPEFSCR[FDBZE] = 1and either SPEFSCR[FDBZH,FDBZ] = 1
- SPEFSCR[FUNFE] = 1 and either SPEFSCR[FUNFH,FUNF] = 1
- SPEFSCR[FOVFE] = 1 and either SPEFSCR[FOVFH,FOVF] = 1

Note that although SPEFSCR status bits can be updated by using **mtspr**, interrupts occur only if they are set as the result of an arithmetic operation.

When an embedded floating-point data interrupt occurs, the processor suppresses execution of the instruction causing the interrupt. Table 5-30 shows register settings.

Table 5-30. Embedded Floating-Point Data Interrupt Register Settings

Register	Setting
SRR0	Set to the effective address of the instruction causing the interrupt.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	CE, ME, and DE are unchanged. All other bits are cleared.
ESR	SPE (bit 24) is set. All other ESR bits are cleared.
SPEFSCR	One or more of the FINVH, FINV, FDBZH, FDBZ, FUNFH, FUNF, FOVFH, or FOVF bits are set to indicate the interrupt type.

Instruction execution resumes at address IVPR[32–47] || IVOR33[48–59] || 0b0000.

5.7.15.3 Embedded Floating-Point Round Interrupt

The embedded floating-point round interrupt is taken on any of the following conditions:

- SPEFSCR[FINXE] = 1 and any of the SPEFSCR[FGH,FXH,FG,FX] bits = 1
- SPEFSCR[FRMC] = 0b10 (+∞)
- SPEFSCR[FRMC] = 0b11 (–∞)

Note that although these SPEFSCR status bits can be updated by using an **mtspr**[SPEFSCR], interrupts occur only if they are set as the result of an arithmetic operation.

When an embedded floating-point round interrupt occurs, the unrounded (truncated) result is placed in the target register. Table 5-31 describes register settings.

Table 5-31. Embedded Floating-Point Round Interrupt Register Settings

Register	Setting
SRR0	Set to the effective address of the instruction following the instruction causing the interrupt.
SRR1	Set to the MSR contents at the time of the interrupt.
MSR	CE, ME, and DE are unchanged. All other MSR bits are cleared.
ESR	SPE (bit 24) is set. All other ESR bits are cleared.
SPEFSCR	FGH, FXH, FG, FX, and FRMC are set appropriately to indicate the interrupt type.

Instruction execution resumes at address IVPR[32–47] || IVOR34[48–59] || 0b0000.

5.8 Performance Monitor Interrupt

The performance monitor provides a performance monitor interrupt that is triggered by an enabled condition or event. An enabled condition or event is as follows:

A PMC*n* register overflow condition occurs with the following settings:

- PMLCa*n*[CE] = 1; that is, for the given counter the overflow condition is enabled.
- PMC*n*[OV] = 1; that is, the given counter indicates an overflow.

For a performance monitor interrupt to be signaled on an enabled condition or event, PMGC0[PMIE] must be set.

The performance monitor can also freeze the performance monitor counters triggered by an enabled condition or event. For the performance monitor counters to freeze on an enabled condition or event, PMGC0[FCECE] must be set.

Although the interrupt condition could occur with MSR[EE] = 0, the interrupt cannot be taken until MSR[EE] = 1. If a counter overflows while PMGC0[FCECE] = 0, PMLCa*n*[CE] = 1, and MSR[EE] = 0, it is possible for the counter to wrap around to all zeros again without the performance monitor interrupt being taken.

The priority of the performance monitor interrupt is below that of the fixed-interval interrupt and above that of the decrementer interrupt.

The APUs chapter of the EREF describes Book E and EIS aspects of the debug interrupt.

5.9 Partially Executed Instructions

In general, the PowerPC 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-string or 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 has been altered.
- For update forms of load or store, the update register, **r**A, will not have been 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 or not.
- 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 or load string, some registers in the range to be loaded may have been altered. Including the addressing registers **r**A and possibly **r**B in the range to be loaded is a programming error, and thus the rules for partial execution do not protect these registers against overwriting.

In no case is access control violated.

As previously stated, elementary, aligned, guarded loads and stores are the only access instructions guaranteed not to be interrupted after being partially executed. The following list identifies the specific instruction types for which interruption after partial execution may occur, as well as the specific interrupt types that could cause the interruption:

1. Any load or store (except elementary, aligned, or guarded):
   — Any asynchronous interrupt
   — Machine check
   — Decrementer
   — Fixed-interval timer
   — Watchdog timer
   — Debug (unconditional debug event)
2. Misaligned elementary load or store, or any multiple or string:
   All of the above listed under item 1, plus the following:
   — Alignment
   — Data storage (if the access crosses a protection boundary)
   — Debug (data address compare)

5.10 Interrupt Ordering and Masking

Multiple exceptions that can each generate an interrupt can exist simultaneously. However, the PowerPC architecture does not provide for reporting multiple simultaneous interrupts of the same class (critical or noncritical). Therefore, the PowerPC architecture defines that interrupts must be ordered with one another and provides a way to mask certain persistent interrupt types.

When an interrupt type is masked (disabled) and an event causes an exception that would normally generate an interrupt of that type, the exception persists as a status bit in a register (which register depends upon the exception type) but no interrupt is generated. Later, if the interrupt type is enabled (unmasked) and the exception status has not been cleared by software, the interrupt due to the original exception event is finally generated. (The e500 only has such a mechanism for certain debug events. A signal that triggers an asynchronous interrupt, such as external input, must be asserted until they are taken. There is no mechanism for saving the external interrupt if the signal is negated before the interrupt is taken. All interrupts are level-sensitive except for machine check, which is edge-triggered.)

All asynchronous interrupt types and some synchronous interrupt types can be masked. The PowerPC architecture allows implementations to avoid situations in which an interrupt would cause state information (saved in save/restore registers) from a previous interrupt to be overwritten and lost. As a first step, upon any noncritical class interrupt, hardware automatically disables further asynchronous, noncritical class interrupts (external input) by clearing MSR[EE]. Likewise, upon any critical class interrupt, hardware automatically disables further asynchronous interrupts, both critical and noncritical, by clearing MSR[CE] and MSR[EE]. Critical input, watchdog timer, and debug interrupts are disabled by clearing MSR[CE,DE]. Note that machine check interrupts, while considered neither asynchronous nor synchronous, are not maskable by MSR[CE,DE,EE] and could be presented in a situation that could cause loss of state information.

This first step of clearing MSR[EE] (and MSR[CE,DE] for critical class interrupts) prevents subsequent asynchronous interrupts from overwriting save/restore registers before software can save their contents. On any interrupt, hardware also clears MSR[WE,PR,FP,FE0,FE1,IS,DS] automatically, which helps avoid subsequent interrupts of certain other types. However, guaranteeing that these interrupt types do not occur also requires system software to avoid executing instructions that could cause (or enable) a subsequent interrupt, if SRR1 contents have not been saved.

5.10.1 Guidelines for System Software

Table 5-32 lists actions system software must avoid before saving save/restore register contents.

Table 5-32. Operations to Avoid

Operation	Reason
Reenabling MSR[EE] (or MSR[CE,DE] in critical class interrupt handlers)	Prevents any asynchronous interrupts, as well as (in the case of MSR[DE]) any debug interrupts, including synchronous and asynchronous types
Branching (or sequential execution) to addresses not mapped by the TLB, mapped without UX = 1 or SX = 1 permission, or causing large address or instruction address overflow exceptions.	Prevents instruction storage, instruction TLB error, and instruction address overflow interrupts
Load, store, or cache management instructions to addresses not mapped by the TLB or not having required access permissions.	Prevents data storage and data TLB error interrupts
Execution of System Call (**sc**) or trap (**tw**, **twi**, **td**, **tdi**) instructions	Prevents system call and trap exception-type program interrupts
Reenabling of MSR[PR]	Prevents privileged instruction exception-type program interrupts. Alternatively, software could reenable MSR[PR] but avoid executing any privileged instructions.
Execution of any illegal instructions	Prevents illegal instruction exception-type program interrupts
Execution of any instruction that could cause an alignment interrupt	Prevents alignment interrupts, including string or multiple instructions and misaligned elementary load or store instructions. Section 5.7.6, "Alignment Interrupt," lists instructions that cause alignment interrupts.

It is unnecessary for hardware or software to avoid critical-class interrupts from within noncritical-class interrupt handlers (hence hardware does not automatically clear MSR[CE,ME,DE] on a noncritical interrupt), since the two interrupt classes use different save/restore registers. However, because a critical-class interrupt can occur within a noncritical handler before the noncritical handler saves SRR0/SRR1, hardware and software must cooperate to avoid both critical and noncritical-class interrupts from within critical class-interrupt handlers. Therefore, within the critical-class interrupt handler, both pairs of save/restore registers may contain data necessary to system software.

5.10.2 Interrupt Order

Enabled interrupt types for which simultaneous exceptions can exist are prioritized as follows:

1. Synchronous (non-debug) interrupts:
   - Data storage
   - Instruction storage
   - Alignment
   - Program
   - System call
   - Data TLB error
   - Instruction TLB error

   Only one of the above synchronous interrupt types may have an existing exception generating it at a given time. This is guaranteed by the exception priority mechanism (see Section 5.11, "Exception Priorities") and the sequential execution model.
2. Machine check
3. Debug
4. Critical input
5. Watchdog timer
6. External input
7. Fixed-interval timer
8. Decrementer

Although, as indicated above, noncritical, synchronous exception types listed under item 1 are generated with higher priority than critical interrupt types in items 2–5, noncritical interrupts are immediately followed by the highest priority existing critical interrupt type, without executing any instructions at the noncritical interrupt handler. This is because noncritical interrupt types do not automatically disable MSR mask bits for critical interrupt types (CE and ME). In all other cases, a particular interrupt type listed above automatically disables subsequent interrupts of the same type, as well as all lower priority interrupt types.

5.11 Exception Priorities

Book E requires all synchronous (precise and imprecise) interrupts to be reported in program order, as required by the sequential execution model. The one exception to this rule is the case of multiple synchronous imprecise interrupts. Upon a synchronizing event, all previously executed instructions are required to report any synchronous imprecise interrupt-generating exceptions, and the interrupt is then generated with all of those exception types reported cumulatively in the ESR and in any status registers associated with the particular exception type (such as the SPEFSCR).

For any single instruction attempting to cause multiple exceptions for which the corresponding synchronous interrupt types are enabled, this section defines the priority order by which the instruction is permitted to cause a single enabled exception, thus generating a particular synchronous interrupt. Note that it is this exception priority mechanism, along with the requirement that synchronous interrupts be generated in program order, that guarantees that at any given time there exists for consideration only one of the synchronous interrupt types listed in item 1 of Section 5.10.2, “Interrupt Order.” 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

Some exception types may even be mutually exclusive of each other and could otherwise be considered the same priority. In these cases, the exceptions are listed in the order suggested by the sequential execution model.

Exception priorities within each instruction type are listed in the following sections. Priority is shown highest to lowest.

5.11.1 e500 Exception Priorities

The following is a prioritized listing of e500 exceptions:

1. HRESET (Note that hard reset is not defined as a true interrupt in Book E, but is included here to show its relationship to the interrupt structure.)
2. Machine_check
3. Debug_ude_exc
4. Critical input
5. Debug interrupt
6. External input
7. Debug—trap | instruction address compare
8. ITLB miss
9. ISI
10. SPE/embedded floating-point APU unavailable
11. Program
12. DTLB miss
13. DSI
14. Alignment
15. Embedded floating-point data interrupt
16. Embedded floating-point round interrupt
17. System call
18. Debug—data address compare | branch taken | instruction compare | return from interrupt
19. Watchdog
20. Fixed interval timer
21. Performance monitor
22. Decrementer

5.12 e500 Interrupt Latency

Interrupt latency of the core complex is 8 cycles or less unless a guarded load or a cache-inhibited **stwcx.** instruction is in the last completion queue entry (CQ0). For specific information, see Section 4.3.4, “Interrupt Latency.”

5.13 Guarded Load and Cache-Inhibited stwcx. Instructions

The e500v2 does not service an interrupt (including machine check) if a guarded load or cache-inhibited **stwcx.** is pending, but if bus errors occur, the load or **stwcx.** instruction may never complete.

If a guarded load gets a bus error, the guarded attribute is cleared on the load. Note that a guarded load cannot go out on the bus until it reaches the bottom of the completion queue (CQ), so only a guarded load in the bottom of the completion queue (CQ0) can get a bus error. When a load hits bad data in the line-fill buffer, *lac_ldst_finish* is squashed (as described above), but *lac_clear_guarded* is asserted in its place (along with the tag). If the tag of CQ0 matches the load/store tag when *lac_clear_guarded* is asserted, the guarded attribute in CQ0 is cleared.

This process allows the completion unit to take an interrupt. If a cache-inhibited **stwcx.** gets an address error, the action taken is effectively the same as what happens if a snoop causes the loss of the reservation. The reservation is cleared, and the cache-inhibited **stwcx.** finishes and reports CR = 0, indicating that the **stwcx.** did not succeed. This allows the **stwcx.** to complete and the completion unit can then take an interrupt.

Note the following:

- This implementation does not make address errors precache-inhibited for cache-inhibited **stwcx.**, as they are for loads. However, if the **stwcx.** failed due to an was an address error, the software is likely to spin in the **lwarx**/**stwcx.** loop until an interrupt occurs. Bus errors on other stores are not precise either.
- Because a cache-inhibited **stwcx.** finishes as soon as the address tenure completes, there is no concern about hanging a cache-inhibited **stwcx.** in completion due to a write bus data error.

Chapter 6
Power Management

This chapter describes the power management facilities as they are defined by Book E and implemented in devices that contain the e500 core. The scope of this chapter is limited to the features of the core complex only. Additional power management capabilities associated with a device that integrates this core (referenced as the integrated device throughout the chapter) are documented separately.

6.1 Overview

A complete power management scheme for a system using the core complex requires the support of logic in the integrated device. The core complex provides software a way to signal a need for power management to the integrated device. It also provides a signal interface that the integrated device can use to transition the core complex into its different power management states.

6.2 Power Management Signals

Table 6-1 summarizes the power management signals of the core complex.

Table 6-1. Power Management Signals of Core Complex

Signal	I/O	Description
halt	I	Asserted by integrated device logic to initiate actions that cause the core complex to enter core-halted state, as follows: • Suspend instruction fetching. • Complete all previously fetched instructions. • When the instruction pipeline is empty, the core asserts the *halted* output. The core clock continues running. Negating *halt* returns the core complex to full-on state. If it is negated before the core complex has entered core-halted state, the negation may not be recognized.
halted	O	Asserted by the core complex when it reaches core-halted state. Indicates to the integrated device logic that it is safe to power-down; that is, no data is lost on transition to the core-stopped state.
stop	I	Asserted by integrated device logic to initiate the required actions that cause the core complex to go from core-halted into core-stopped state (as described in Table 6-2). Negating *stop* returns the core complex to core-halted state. Once asserted, *stop* must not be negated until after the core complex has entered the core-stopped state; otherwise the negation may not be recognized. For power management purposes, *stop* must be asserted only while the core complex is in the core-halted state.
stopped	O	Asserted by the core anytime the internal functional clocks of the core complex are stopped (for example after integrated device logic asserts *stop*).

Table 6-1. Power Management Signals of Core Complex (continued)

Signal	I/O	Description
tben	I	Asserted by the integrated device logic to enable the time base.
tbint	O	Asserted when a time base interrupt is signaled. This ordinarily prompts logic in the integrated device to bring the core out of core-stopped state to service the interrupt.
doze	O	Reflect the state of corresponding HID0[DOZE,NAP,SLEEP] bits (if MSR[WE] = 1); both must be set for the respective output to assert. These signals do not affect the core's power-down state, but indicate to the integrated device of power management requests made by software. Integrated device logic may use these signals to affect device-level power state, which in turn may affect the core complex power state (signaled through the *halt*, *stop*, and *tben*).
nap	O	
sleep	O	

6.3 Core and Integrated Device Power Management States

The notion of nap, doze, and sleep modes (or states) pertains to the integrated device as a whole. As shown in Figure 6-1, an integrated device may interpret the assertion of *nap*, *doze*, and *sleep* to trigger actions that affect the device-level power state, which in turn may use the *halt*, *stop*, and *tben* inputs to determine how the core transitions between the core-specific power states: full on, core halted, and core stopped.

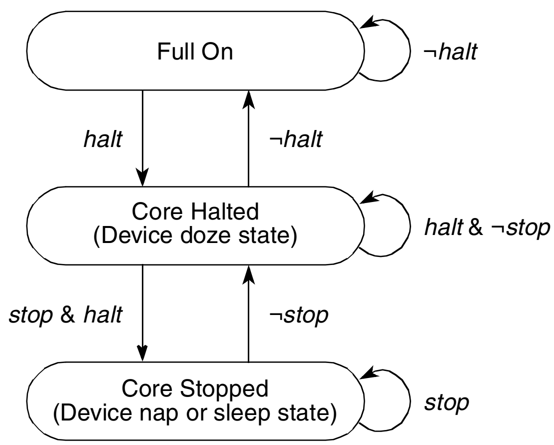


Figure 6-1. Core Power Management State Diagram

In addition to the power-management states, dynamic power management automatically stops clocking individual internal functional units whenever they are idle. The integrated logic may similarly stop clocking to idle device-level blocks.

Table 6-2 describes the core power management states.

Table 6-2. Core Power States

State	Descriptions
Full on (default)	Default. All internal units are operating at the full clock speed defined at power-up. Dynamic power management automatically stops clocking individual internal functional units that are idle.
Core halted	Initiated by asserting the *halt* input. The core complex responds by stopping instruction execution. It then asserts the *halted* output to indicate that it is in the core-halted state. Core complex clocks continue running, and bus snooping continues to maintain L1 cache coherency. As Figure 6-1 shows, the core complex is in core-halted state when the integrated device is in doze state.
Core stopped	Initiated entered when *stop* is asserted to the core while it is in core-halted state. The core responds by inhibiting clock distribution to most of its functional units (after the CCB interface idles), and then asserting the *stopped* output. Internal PLL clock generation is maintained to allow quick recovery to core-halted or full-on state. Although snooping cannot occur in core-stopped state, cache coherency can be maintained by allowing the core to temporarily return to core-halted state, as described below.
	Disabling the timer facility and PLL. Additional power reduction is achieved by negating the time base enable (*tben*) input, which suspends timer facility operations. Note that *tben* controls the time base (and decrementer) in all power management states. Timer operation is independent of power management except for software considerations required for processing timer interrupts that occur during core-stopped state. For example, if the timer facility is stopped, software ordinarily uses an external time reference to update the various timing counters upon restart. Core power can be further reduced by stopping the internal PLL unit (through the *pll_cfg[0:5]* inputs) and optionally by stopping *pll_clk*. To recover from this complete shutdown, the system must first restart the PLL (through *pll_cfg[0:5]*, and *pll_clk* if it was stopped) and allow time for the PLL to lock before any external interrupt is signaled to the core. This state is unsuitable for dynamic snooping because of the PLL's long start-up and lock time. Refer to Table 13-1 for the encodings of the PLL_CFG[0:5] inputs.
	Dynamic bus snooping. To maintain L1 cache coherency, the core complex can be momentarily restored to core-halted state (by negating *stop*; *halt* remains asserted) to perform snoop operations. After the core complex exits core-stopped state (*stopped* negated), the core complex can recognize snoop transactions on the CCB. While the core is in core-halted state and *stop* and *stopped* are negated, snoops are issued only to the core complex. The core returns to core-stopped state when snooping (and any required snoop response and snoop copy-back transactions on the CCB) completes.

6.4 Power Management Control Bits

Although the core can signal power management through the bits shown in Table 6-3, core power management is controlled by the integrated device, which may provide additional ways to put the core into a power-saving state. Interlocks between the core and the integrated device prevent data loss that could occur if one part of the system powered down before the other had time to prepare.

Table 6-3. Core Power Management Control Bits

Bit	Description
MSR[WE]	Must be set for HID0[DOZE,NAP,SLEEP] to cause assertion of *doze*, *nap*, and *sleep* to system logic.
HID0[DOZE]	If MSR[WE] = 1, signals power management logic to initiate device-level doze state. The core complex enters core-halted state after integrated device logic asserts *halt*.
HID0[NAP]	If MSR[WE] = 1, signals power management logic to initiate device nap mode. The core complex enters core-stopped state (with its time base enabled) after integrated device logic asserts *stop*.

Table 6-3. Core Power Management Control Bits (continued)

Bit	Description
HID0[SLEEP]	If MSR[WE] = 1, signals power management logic to initiate device sleep mode. The core complex remains in core-stopped state and stops its time base after integrated device logic negates *tben*.
HID0[TBEN]	Time base and decrementer enable 0 Time base disabled (no counting) • 1Time base enabled

NOTE

The e500 does not implement its own doze, nap, and sleep modes. The core-halted and core-stopped states may correlate to the integrated device's doze, nap, and sleep modes, but the e500 cannot be put into core-halted or core-stopped state without interaction with system integration logic.

6.4.1 Software Considerations for Power Management

Setting MSR[WE] generates a request to the power management logic of the integrated device (external to the core complex) to enter a power-saving state. It is assumed that the desired power-saving state (doze, nap, or sleep) has been previously set up by setting the appropriate HID0 bit, typically at system start-up time. Setting MSR[WE] does not directly affect instruction execution, but it is reflected on the core *doze*, *nap*, and *sleep* signals, depending on the HID0[DOZE,NAP,SLEEP] settings.

To ensure a clean transition into and out of a power-saving mode, the following program sequence is recommended:

```
        msync
        mtmsr (WE)
        isync
loop:   br loop
```

6.5 Power Management Protocol

The e500 outputs the *doze*, *nap*, and *sleep* signals to the integrated device logic, which controls power states both for the device as a whole and for the core (namely the core-halted and core-stopped states). Figure 6-2 shows how device integration logic would typically respond to *doze*, *nap*, and *sleep* and control the core's power state through the *halt*/*halted*, *stop*/*stopped*, and *tben*/*tbint* signal pairs.

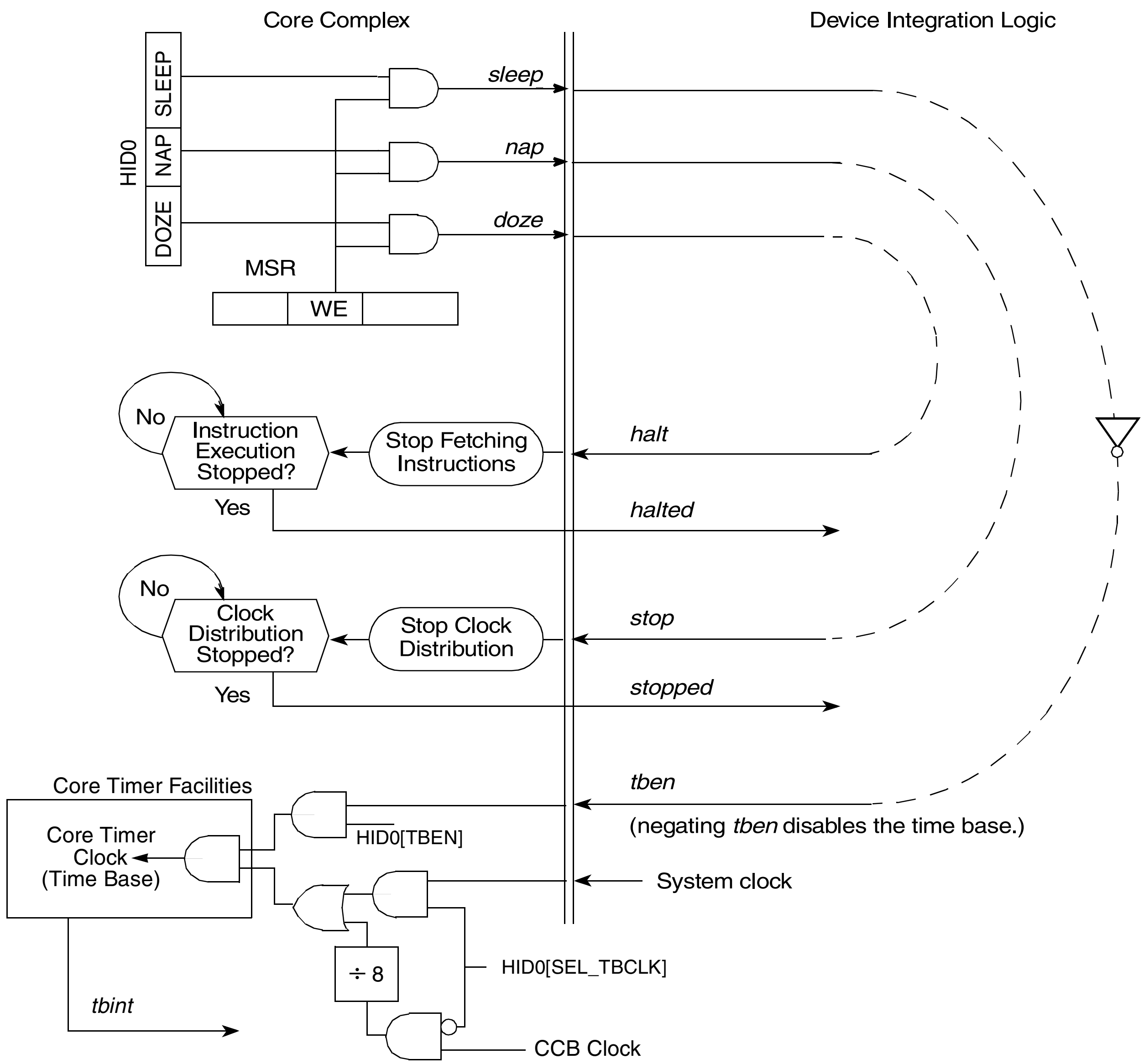


Figure 6-2. Example Core Power Management Handshaking

6.6 Interrupts and Power Management

In core-halted or core-stopped state, the core complex does not recognize interrupts. The power management logic of the integrated device must monitor all external interrupt requests (as well as the e500 *tbint* output) to detect interrupt requests. Upon sensing an interrupt request, the integrated device ordinarily negates *stop* and *halt* to restore the core to full-on state, allowing it to service the interrupt request.

MSR[WE], which gates the *doze*, *nap*, and *sleep* power management outputs from the core complex, is always saved to save/restore register (SRR1, CSRR1, or MCSRR1, depending on the interrupt) when an interrupt is taken and restored to the MSR when the handler issues an **rfi**, **rfci**, or **rfmci**. As a result, *doze*, *nap*, and *sleep* outputs negate to the external power management logic on entry to the interrupt service routine and then return to their previous state on return from the interrupt when MSR[WE] value is restored. This function of MSR[WE] has the following implications for the design of power management software:

- In previous devices, when the processor exits a low-power state, MSR[POW], which enables power-down requests, is cleared without being automatically restored, unlike Book E implementations which restore WE.
- Assuming that the system entered a low-power state in response to the assertion of *doze*, *nap*, or *sleep*, the integrated device's power management logic must recognize that these outputs remain asserted for some time after the core complex is restored to full-on state (due to the normal latency of restarting internal clock distribution and initiating the interrupt request), and then negate as the interrupt is serviced.

Chapter 7 Performance Monitor

This chapter describes the performance monitor, which is generally defined by the Freescale Book E implementation standards (EIS) and implemented as an APU on the e500 core. Although the programming model is defined by the EIS, some features are defined by the e500 implementation, in particular, the events that can be counted.

References to e500 apply to both e500v1 and e500v2.

7.1 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 in the MSR (MSR[PMM]). This bit controls which programs are monitored.
- The move to/from performance monitor registers (PMR) instructions, **mtpmr** and **mfpmr**.
- The external input, *pm_event*.

- PMRs:
  - — The performance monitor counter registers (PMC0–PMC3) are 32-bit counters used to count software-selectable events. Each counter counts up to 128 events. UPMC0–UPMC3 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 7-10.
  - — The performance monitor global control register (PMGC0) controls the counting of performance monitor events. It takes priority over all other performance monitor control registers. UPMGC0 provides user-level read access to PMGC0.
  - — The performance monitor local control registers (PMLCa0–PMLCa3, PMLCb0–PMLCb3) control each individual performance monitor counter. Each counter has a corresponding PMLCa and PMLCb register. UPMLCa0–UPMLCa3 and UPMLCb0–UPMLCb3 provide user-level read access to PMLCa0–PMLCa3, PMLCb0–PMLCb3).
- The performance monitor interrupt follows the Book E interrupt model and is assigned to interrupt vector offset register 35 (IVOR35). Its priority is less than the fixed-interval interrupt and greater than the decrementer interrupt.

Software communication with the performance monitor APU is achieved through PMRs rather than SPRs. The PMRs are used for enabling conditions that can trigger a APU-defined performance monitor interrupt.

7.2 Performance Monitor APU Registers

The performance monitor APU provides a set of PMRs for defining, enabling, and counting conditions that trigger the performance interrupt. It also defines IVOR35 (SPR 531) for indicating the address of the performance monitor interrupt vector. IVOR35 is described in Section 2.7.1.5, “Interrupt Vector Offset Registers (IVORs).”

The supervisor-level performance monitor registers in Table 7-1 are accessed with **mtpmr** and **mfpmr**. Attempting to read or write supervisor-level registers in user-mode causes a privilege exception.

Table 7-1. Performance Monitor Registers–Supervisor Level

Number	PMR[0–4]	PMR[5–9]	Name	Abbreviation
16	00000	10000	Performance monitor counter 0	PMC0
17	00000	10001	Performance monitor counter 1	PMC1
18	00000	10010	Performance monitor counter 2	PMC2
19	00000	10011	Performance monitor counter 3	PMC3
144	00100	10000	Performance monitor local control a0	PMLCa0

Table 7-1. Performance Monitor Registers–Supervisor Level (continued)

Number	PMR[0–4]	PMR[5–9]	Name	Abbreviation
145	00100	10001	Performance monitor local control a1	PMLCa1
146	00100	10010	Performance monitor local control a2	PMLCa2
147	00100	10011	Performance monitor local control a3	PMLCa3
272	01000	10000	Performance monitor local control b0	PMLCb0
273	01000	10001	Performance monitor local control b1	PMLCb1
274	01000	10010	Performance monitor local control b2	PMLCb2
275	01000	10011	Performance monitor local control b3	PMLCb3
400	01100	10000	Performance monitor global control 0	PMGC0

The user-level performance monitor registers in Table 7-2 are read-only and are accessed with the **mfpmr** instruction. Attempting to write these user-level registers in either supervisor or user mode causes an illegal instruction exception.

Table 7-2. Performance Monitor Registers–User Level (Read-Only)

Number	PMR[0–4]	PMR[5–9]	Name	Abbreviation
0	00000	00000	Performance monitor counter 0	UPMC0
1	00000	00001	Performance monitor counter 1	UPMC1
2	00000	00010	Performance monitor counter 2	UPMC2
3	00000	00011	Performance monitor counter 3	UPMC3
128	00100	00000	Performance monitor local control a0	UPMLCa0
129	00100	00001	Performance monitor local control a1	UPMLCa1
130	00100	00010	Performance monitor local control a2	UPMLCa2
131	00100	00011	Performance monitor local control a3	UPMLCa3
256	01000	00000	Performance monitor local control b0	UPMLCb0
257	01000	00001	Performance monitor local control b1	UPMLCb1
258	01000	00010	Performance monitor local control b2	UPMLCb2
259	01000	00011	Performance monitor local control b3	UPMLCb3
384	01100	00000	Performance monitor global control 0	UPMGC0

7.2.1 Global Control Register 0 (PMGC0)

The performance monitor global control register (PMGC0), shown in Figure 7-1, controls all performance monitor counters.

PMGC0 (PMR400)
UPMGC0 (PMR384)

Access: PMGC0: Supervisor-only
UPMGC0: Supervisor/user read-only

	32	33	34	35–50	51–52	53–54	55	56–63
R/W	FAC	PMIE	FCECE	—	TBSEL[1]	—	TBEE[1]	—
Reset	All zeros							

[1] e500v2 only

Figure 7-1. Performance Monitor Global Control Register 0 (PMGC0)/ User Performance Monitor Global Control Register 0 (UPMGC0)

PMGC0 is cleared by a hard reset. Reading this register does not change its contents. Table 7-3 describes PMGC0 fields.

Table 7-3. PMGC0 Field Descriptions

Bits	Name	Description
32	FAC	Freeze all counters. When FAC is set by hardware or software, PMLCx[FC] maintains its current value until it is changed by software. 0 The PMCs are incremented (if permitted by other PM control bits). 1 The PMCs are not incremented.
33	PMIE	Performance monitor interrupt enable 0 Performance monitor interrupts are disabled. 1 Performance monitor interrupts are enabled and occur when an enabled condition or event occurs, at which time PMGC0[PMIE] is cleared Software can clear PMIE to prevent performance monitor interrupts. Performance monitor interrupts are caused by time base events or PMCx overflow.
34	FCECE	Freeze counters on enabled condition or event 0 The PMCs can be incremented (if permitted by other PM control bits). 1 The PMCs can be incremented (if permitted by other PM control bits) only until an enabled condition or event occurs. When an enabled condition or event occurs, PMGC0[FAC] is set. It is up to software to clear FAC. An enabled condition or event is defined as one of the following: • When the msb = 1 in PMC*x* and PMLCa*x*[CE] = 1. • When the time-base bit specified by TBSEL=1 and TBEE=1.
35–50	—	Reserved, should be cleared.

Table 7-3. PMGC0 Field Descriptions (continued)

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 multiprocessor systems in which TB registers are synchronized across processors, these events can be used to correlate performance monitor data obtained by the several processors. For this use, software must specify the same TBSEL value for all processors in the system. Time-base frequency is implementation-dependent, so software should invoke a system service program to obtain the frequency before choosing a TBSEL value.
53–54	—	Reserved, should be cleared.
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 signalled 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 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.
56–63	—	Reserved, should be cleared.

7.2.2 User Global Control Register 0 (UPMGC0)

The contents of PMGC0 are reflected to UPMGC0, which can be read by user-level software. UPMGC0 can be read with the **mfpmr** instruction using PMR384.

7.2.3 Local Control A Registers (PMLCa0–PMLCa3)

The local control A registers (PMLCa0–PMLCa3) function as event selectors and give local control for the corresponding performance monitor counters. PMLCa works with the corresponding PMLCb register. PMLCa registers are shown in Figure 7-2.

PMLCa0 (PMR144)
PMLCa1 (PMR145)
PMLCa2 (PMR146)
PMLCa3 (PMR147)

UPMLCa0 (PMR128)
UPMLCa1 (PMR129)
UPMLCa2 (PMR130)
UPMLCa3 (PMR131)

Access: PMLCa0–PMLCa3: Supervisor-only
UPMLCa0–UPMLCa3: Supervisor/user read-only

	32	33	34	35	36	37	38–40	41–47	48–63
R W	FC	FCS	FCU	FCM1	FCM0	CE	—	EVENT	—
Reset	All zeros								

Figure 7-2. Local Control A Registers (PMLCa0–PMLCa3)/ User Local Control A Registers (UPMLCa0–UPMLCa3)

PMLCa registers are cleared by a hard reset. Table 7-4 describes PMLCa fields.

Table 7-4. PMLCa0–PMLCa3 Field Descriptions

Bits	Name	Description
32	FC	Freeze counter. 0 The PMC can be incremented (if enabled by other performance monitor control fields). 1 The PMC cannot be incremented.
33	FCS	Freeze counter in supervisor state. 0 The PMC can be incremented (if enabled by other performance monitor control fields). 1 The PMC cannot be incremented if MSR[PR] is cleared.
34	FCU	Freeze counter in user state. 0 The PMC can be incremented (if enabled by other performance monitor control fields). 1 The PMC cannot be incremented if MSR[PR] is set.
35	FCM1	Freeze counter while mark is set. 0 The PMC can be incremented (if enabled by other performance monitor control fields). 1 The PMC cannot be incremented if MSR[PMM] is set.
36	FCM0	Freeze counter while mark is cleared. 0 The PMC can be incremented (if enabled by other performance monitor control fields). 1 The PMC cannot be incremented if MSR[PMM] is cleared.
37	CE	Condition enable. 0 Overflow conditions for PMC*n* cannot occur (PMC*n* cannot cause interrupts or freeze counters) 1 Overflow conditions occur when the most-significant-bit of PMC*n* is equal to 1. It is recommended that CE be cleared when counter PMC*n* is selected for chaining.
38–40	—	Reserved, should be cleared.
41–47	EVENT	Event selector. Up to 128 events selectable. See Section 7.7, “Event Selection”
48–63	—	Reserved, should be cleared.

7.2.4 User Local Control A Registers (UPMLCa0–UPMLCa3)

The PMLCa contents are reflected to UPMLCa0–UPMLCa3, which can be read by user-level software with **mfpmr** using PMR numbers in Table 7-2.

7.2.5 Local Control B Registers (PMLCb0–PMLCb3)

Local control B registers 0 through 3 (PMLCb0–PMLCb3) specify a threshold value and a multiple to apply to a threshold event selected for the corresponding performance monitor counter. For the e500, thresholding is supported only for PMC0 and PMC1. PMLCb works with the corresponding PMLCa register.

PMLCb registers are shown in Figure 7-3.

PMLCb0 (PMR272) UPMLCb0 (PMR256)
PMLCb1 (PMR273) UPMLCb1 (PMR257)
PMLCb2 (PMR274) UPMLCb2 (PMR258)
PMLCb3 (PMR275) UPMLCb3 (PMR259)

Access: PMLCb0–PMLCb3: Supervisor-only
UPMLCb0–UPMLCb3: Supervisor/user read-only

	32–52	53–55	56–57	58–63
R/W	—	THRESHMUL	—	THRESHOLD
Reset	All zeros			

Figure 7-3. Local Control B Registers (PMLCb0–PMLCb3)/ User Local Control B Registers (UPMLCb0–UPMLCb3)

PMLCb is cleared by a hard reset. Table 7-5 describes PMLCb fields.

Table 7-5. PMLCb0–PMLCb3 Field Descriptions

Bits	Name	Description
32–52	—	Reserved, should be cleared.
53–55	THRESHMUL	Threshold multiple. 000 Threshold field is multiplied by 1 (PMLCb*n*[THRESHOLD] × 1) 001 Threshold field is multiplied by 2 (PMLCb*n*[THRESHOLD] × 2) 010 Threshold field is multiplied by 4 (PMLCb*n*[THRESHOLD] × 4) 011 Threshold field is multiplied by 8 (PMLCb*n*[THRESHOLD] × 8) 100 Threshold field is multiplied by 16 (PMLCb*n*[THRESHOLD] × 16) 101 Threshold field is multiplied by 32 (PMLCb*n*[THRESHOLD] × 32) 110 Threshold field is multiplied by 64 (PMLCb*n*[THRESHOLD] × 64) 111 Threshold field is multiplied by 128 (PMLCb*n*[THRESHOLD] × 128)
56–57	—	Reserved, should be cleared.
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 unit (for example duration in cycles) and the granularity with which the threshold value is interpreted. By varying the threshold value, software can obtain a profile of the event characteristics subject to thresholding. For example, if PMC1 is configured to count cache misses that exceed the threshold value, software can measure the distribution of cache miss durations for a given program by monitoring the program repeatedly using a different threshold value each time.

7.2.6 User Local Control B Registers (UPMLCb0–UPMLCb3)

The contents of PMLCb0–PMLCb3 are reflected to UPMLCb0–UPMLCb3, which can be read by user-level software with **mfpmr** using PMR numbers in Table 7-2.

7.2.7 Performance Monitor Counter Registers (PMC0–PMC3)

The performance monitor counter registers (PMC0–PMC3), shown in Figure 7-4, are 32-bit counters that can be programmed to generate interrupt signals when they overflow. Each counter is enabled to count up to 128 events.

PMC0 (PMR16) UPMC0 (PMR0)
PMC1 (PMR17) UPMC1 (PMR1)
PMC2 (PMR18) UPMC2 (PMR2)
PMC3 (PMR19) UPMC3 (PMR3)

Access: PMC0–PMC3: Supervisor-only
UPMC0–UPMC3: Supervisor/user read-only

	32	33–63
R / W	OV	Counter value
Reset	All zeros	

Figure 7-4. Performance Monitor Counter Registers (PMC0–PMC3)/ User Performance Monitor Counter Registers (UPMC0–UPMC3)

PMCs are cleared by a hard reset. Table 7-6 describes PMC register fields.

Table 7-6. PMC0–PMC3 Field Descriptions

Bits	Name	Description
32	OV	Overflow. 0 Counter has not reached an overflow state. 1 Counter has reached an overflow state.
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 wraps to the minimum value.

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

NOTE

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

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

- If PMLCa*n*[CE] is clear, no special actions occur on overflow: the counter continues incrementing, and no exception is signaled.
- If PMLCa*n*[CE] and PMGC0[FCECE] are set, all counters are frozen when PMC*n* overflows.
- If PMLCa*n*[CE] and PMGC0[PMIE] are set, an exception is signaled when PMC*n* reaches overflow. Interrupts are masked by clearing MSR[EE]. An exception may be signaled while EE is zero, but the interrupt is not taken until it is set 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 EE remains clear until after the counter leaves the overflow state (msb becomes 0), or if EE remains clear until after PMLCa*n*[CE] or PMGC0[PMIE] cleared, the exception is not signaled.

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

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

7.2.8 User Performance Monitor Counter Registers (UPMC0–UPMC3)

The contents of PMC0–PMC3 are reflected to UPMC0–UPMC3, which can be read by user-level software with the **mfpmr** instruction using PMR numbers in Table 7-2.

7.3 Performance Monitor APU Instructions

The APU defines instructions for reading and writing the PMRs as shown in Table 7-7.

Table 7-7. Performance Monitor APU Instructions

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

7.4 Performance Monitor Interrupt

The performance monitor interrupt is triggered by an enabled condition or event. The only enabled condition or event defined for the e500 is the following:

- A PMC*n* overflow condition occurs when both of the following are true:
  — The counter's overflow condition is enabled; PMLCa*n*[CE] is set.
  — The counter indicates an overflow; PMC*n*[OV] is set.

If PMGC0[PMIE] is set, an enabled condition or event triggers the signaling of a performance monitor exception.

If PMGC0[FCECE] is set, an enabled condition or event also triggers all performance monitor counters to freeze.

Although the performance monitor exception condition could occur with MSR[EE] cleared, the interrupt cannot be taken until MSR[EE] is set. If PMC*n* overflows and would signal an exception (PMLCa*n*[CE] and PMGC0[PMIE] are set) while interrupts are disabled (MSR[EE] is clear), and freezing of the counters is not enabled (PMGC0[FCECE] is clear), PMC*n* can wrap around to all zeros again without the performance monitor interrupt being taken.

7.5 Event Counting

This section describes configurability and specific unconditional counting modes.

7.5.1 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 the PMLCa*n*[FCS,FCU,FCM1,FCM0] fields, the state for which monitoring is enabled, counting is enabled for PMC*n*.

The processor states and the settings of the FCS, FCU, FCM1, and FCM0 fields in PMLCa*n* necessary to enable monitoring of each processor state are shown in Table 7-8.

Table 7-8. Processor States and PMLCa0–PMLCa3 Bit Settings

Processor State	FCS	FCU	FCM1	FCM0
Marked	0	0	0	1
Not marked	0	0	1	0
Supervisor	0	1	0	0
User	1	0	0	0
Marked and supervisor	0	1	0	1
Marked and user	1	0	0	1
Not marked and supervisor	0	1	1	0
Not mark and user	1	0	1	0
All	0	0	0	0
None	X	X	1	1
None	1	1	X	X

Two unconditional counting modes may be specified:

- Counting is unconditionally enabled regardless of the states of MSR[PMM] and MSR[PR]. This can be accomplished by clearing PMLCa*n*[FCS], PMLCa*n*[FCU], PMLCa*n*[FCM1], and PMLCa*n*[FCM0] for each counter control.
- Counting is unconditionally disabled regardless of the states of MSR[PMM] and MSR[PR]. This can be accomplished by setting PMGC0[FAC] or by setting PMLCa*n*[FC] for each counter control. Alternatively, this can be accomplished by setting PMLCa*n*[FCM1] and PMLCa*n*[FCM0] for each counter control or by setting PMLCa*n*[FCS] and PMLCa*n*[FCU] for each counter control.

7.6 Examples

The following sections provide examples of how to use the performance monitor facility:

7.6.1 Chaining Counters

The counter chaining feature can be used to decrease the processing pollution caused by performance monitor interrupts (such as cache contamination and pipeline effects) by allowing a higher event count than is possible with a single counter. Chaining two counters together effectively adds 32 bits to a counter register where the first counter's overflow event acts like a carry out feeding the second counter. By defining the event of interest to be another PMC's overflow generation, the chained counter increments each time the first counter rolls over to zero. Multiple counters may be chained together.

Because the entire chained value cannot be read in a single instruction, an overflow may occur between counter reads, producing an inaccurate value. A sequence like the following is necessary to read the complete chained value when it spans multiple counters and the counters are not frozen. The example shown is for a two-counter case.

```
loop:   mfpmr           Rx,pmctr1       #load from upper counter
        mfpmr           Ry,pmctr0       #load from lower counter
        mfpmr           Rz,pmctr1       #load from upper counter
        cmp             cr0,0,Rz,Rx     #see if 'old' = 'new'
        bc              4,2,loop        #loop if carry occurred between reads
```

The comparison and loop are necessary to ensure that a consistent set of values has been obtained. The above sequence is not necessary if the counters are frozen.

7.6.2 Thresholding

Threshold event measurement enables the counting of duration and usage events. For example, data line fill buffer (DLFB) load miss cycles (event C0:76 and C1:76) require a threshold value. A DLFB load miss cycles event is counted only when the number of cycles spent recovering from the miss is greater than the threshold. Because this event is counted on two counters and each counter has an individual threshold, one execution of a performance monitor program can sample two different threshold values. Measuring code performance with multiple concurrent thresholds expedites code profiling significantly.

7.7 Event Selection

Event selection is specified through the PMLCa*n* registers described in Section 7.2.3, "Local Control A Registers (PMLCa0–PMLCa3)." The event-select fields in PMLCa*n*[EVENT] are described in Table 7-10, which lists encodings for the selectable events to be monitored. Table 7-10 establishes a correlation between each counter, events to be traced, and the pattern required for the desired selection.

The Spec/Nonspec column indicates whether the event count includes any occurrences due to processing that was not architecturally required by the PowerPC sequential execution model (speculative processing).

- Speculative counts include speculative instructions that were later flushed.
- Nonspeculative counts do not include speculative operations, which are flushed.

Table 7-9 describes how event types are indicated in Table 7-10.

Table 7-9. Event Types

Event Type	Label	Description
Reference	Ref:#	Shared across counters PMC0—PMC3. Applicable to most microprocessors.
Common	Com:#	Shared across counters PMC0–PMC3. Fairly specific to e500 microarchitectures.
Counter-specific	C[0–3]:#	Counted only on one or more specific counters. The notation indicates the counter to which an event is assigned. For example, an event assigned to counter PMC2 is shown as C2:#.

Table 7-10 describes performance monitor events. Pipeline events in Table 7-10 are defined in Chapter 4, "Execution Timing."

Table 7-10. Performance Monitor Event Selection

Number	Event	Spec/ Nonspec	Count Description
General Events			
Ref:0	Nothing	Nonspec	Register counter holds current value
Ref:1	Processor cycles	Nonspec	Every processor cycle
Ref:2	Instructions completed	Nonspec	Completed instructions. 0, 1, or 2 per cycle.
Com:3	Micro-ops completed [1]	Nonspec	Completed micro-ops. 0, 1, or 2 per cycle. (1 for each standard instruction, 2 for load/store-with-update. 1–32 for load or store multiple instructions)
Com:4	Instructions fetched	Spec	Fetched instructions. 0, 1, 2, 3, or 4 per cycle. (instructions written to the IQ.)
Com:5	Micro-ops decoded [1]	Spec	Micro-ops decoded. 0, 1, or 2 per cycle. (2 for load/store-with-update)
Com:6	PM_EVENT transitions	Spec	0 to 1 transitions on the *pm_event* input.
Com:7	PM_EVENT cycles	Spec	Processor cycles that occur when the *pm_event* input is asserted.
Instruction Types Completed			
Com:8	Branch instructions completed	Nonspec	Completed branch instructions.
Com:9	Load micro-ops completed [1]	Nonspec	Completed load micro-ops. (**l***, **evl***, load-update (1 load micro-op), load-multiple (1–32 micro-ops), **dcbt**(L1, CT = 0), and **dcbtst**(L1, CT = 0)
Com:10	Store micro-ops completed [1]	Nonspec	Completed store micro-ops. (**st***, **evst***, store-update (1 store micro-op), store-multiple (1–32 micro-ops), **tlbivax**, **icbi**, **icblc**, **icbtls**, **dcba**, **dcbf**, **dcblc**, **dcbst**, **dcbt**(L2, CT = 1), **dcbtls**, **dcbtst**(L2, CT = 1), **dcbtstls**, **dcbz**, **icbt**(L2, CT = 1), **mbar**, and **msync**)
Com:11	Number of CQ redirects	Nonspec	Fetch redirects initiated from the completion unit. (for example, resulting from **sc**, **rfi**, **rfci**, **rfmci**, **isync**, and interrupts)
Branch Prediction and Execution Events			
Com:12	Branches finished	Spec	Includes all branch instructions
Com:13	Taken branches finished	Spec	Includes all taken branch instructions

Table 7-10. Performance Monitor Event Selection (continued)

Number	Event	Spec/ Nonspec	Count Description
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.
Pipeline Stalls			
Com:18	Cycles decode stalled	Spec	Cycles the IQ is not empty but 0 instructions decoded
Com:19	Cycles issue stalled	Spec	Cycles the issue buffer is not empty but 0 instructions issued
Com:20	Cycles branch issue stalled	Spec	Cycles the branch buffer is not empty but 0 instructions issued
Com:21	Cycles SU1 schedule stalled	Spec	Cycles SU1 is not empty but 0 instructions scheduled
Com:22	Cycles SU2 schedule stalled	Spec	Cycles SU2 is not empty but 0 instructions scheduled
Com:23	Cycles MU schedule stalled	Spec	Cycles MU is not empty but 0 instructions scheduled
Com:24	Cycles LRU schedule stalled	Spec	Cycles LRU is not empty but 0 instructions scheduled
Com:25	Cycles BU schedule stalled	Spec	Cycles BU is not empty but 0 instructions scheduled
Load/Store, Data Cache, and Data Line Fill Buffer (DLFB) Events			
Com:26	Total translated	Spec	Total of load and store micro-ops that reach the second stage of the LSU [1, 2]
Com:27	Loads translated	Spec	Cacheable **l*** or **evl*** micro-ops translated. (includes load micro-ops from load-multiple and load-update instructions) [1,2]
Com:28	Stores translated	Spec	Cacheable **st*** or **evst*** micro-ops translated. (includes micro-ops from store-multiple, and store-update instructions) [1,2]
Com:29	Touches translated	Spec	Cacheable **dcbt** and **dcbtst** instructions translated (L1 only) (Doesn't count touches that are converted to nops i.e. exceptions, noncacheable, HID0[NOPTI] is set.)
Com:30	Cacheops translated	Spec	**dcba**, **dcbf**, **dcbst**, and **dcbz** instructions translated.
Com:31	Cache-inhibited accesses translated	Spec	Cache inhibited accesses translated
Com:32	Guarded loads translated	Spec	Guarded loads translated
Com:33	Write-through stores translated	Spec	Write-through stores translated
Com:34	Misaligned load or store accesses translated	Spec	Misaligned load or store accesses translated.
Com:35	Total allocated to DLFB	Spec	—
Com:36	Loads translated and allocated to DLFB	Spec	Applies to same class of instructions as loads translated.

Table 7-10. Performance Monitor Event Selection (continued)

Number	Event	Spec/ Nonspec	Count Description
Com:37	Stores completed and allocated to DLFB	Nonspec	Applies to same class of instructions as stores translated.
Com:38	Touches translated and allocated to DLFB	Spec	Applies to same class of instructions as touches translated.
Com:39	Stores completed	Nonspec	Cacheable **st*** or **evst*** micro-ops completed. (Applies to the same class of instructions as stores translated.) [1,2]
Com:40	Data L1 cache locks	Nonspec	Cache lines locked in the data L1 cache. (Counts a lock even if an overlock condition occurs.)
Com:41	Data L1 cache reloads	Spec	Counts cache reloads for any reason. Typically used to determine data cache miss rate (along with loads/stores completed).
Com:42	Data L1 cache castouts	Spec	Does not count castouts due to **dcbf**.
Data Side Replay Conditions: Times Detected			
Com:43	Load miss with DLFB full.	Spec	Counts number of stalls; Com:51 counts cycles stalled.
Com:44	Load miss with load queue full.	Spec	Counts number of stalls; Com:52 counts cycles stalled.
Com:45	Load guarded miss when the load is not yet at the bottom of the CQ.	Spec	Counts number of stalls; Com:53 counts cycles stalled.
Com:46	Translate a store when the store queue is full.	Spec	Counts number of stalls; Com:54 counts cycles stalled.
Com:47	Address collision.	Spec	Counts number of stalls; Com:55 counts cycles stalled.
Com:48	Data MMU miss.	Spec	Counts number of stalls; Com:56 counts cycles stalled.
Com:49	Data MMU busy.	Spec	Counts number of stalls; Com:57 counts cycles stalled.
Com:50	Second part of misaligned access when first part missed in cache.	Spec	Counts number of stalls; Com:58 counts cycles stalled.
Data Side Replay Conditions: Cycles Stalled			
Com:51	Load miss with DLFB full.	Spec	Counts cycles stalled; Com:43 counts number of stalls.
Com:52	Load miss with load queue full.	Spec	Counts cycles stalled; Com:44 counts number of stalls.
Com:53	Load guarded miss when the load is not yet at the bottom of the CQ.	Spec	Counts cycles stalled; Com:45 counts number of stalls.
Com:54	Translate a store when the store queue is full.	Spec	Counts cycles stalled; Com:46 counts number of stalls.
Com:55	Address collision.	Spec	Counts cycles stalled; Com:47 counts number of stalls.
Com:56	Data MMU miss.	Spec	Counts cycles stalled; Com:48 counts number of stalls.
Com:57	Data MMU busy.	Spec	Counts cycles stalled; Com:49 counts number of stalls.
Com:58	Second part of misaligned access when first part missed in cache.	Spec	Counts cycles stalled; Com:50 counts number of stalls.
Fetch, Instruction Cache, Instruction Line Fill Buffer (ILFB), and Instruction Prefetch Events			
Com:59	Instruction L1 cache locks	Nonspec	Counts cache lines locked in the instruction L1 cache. (Counts a lock even if an overlock occurs.)

Table 7-10. Performance Monitor Event Selection (continued)

Number	Event	Spec/ Nonspec	Count Description
Com:60	Instruction L1 cache reloads from fetch	Spec	Counts reloads due to demand fetch. Typically used to determine instruction cache miss rate (along with instructions completed)
Com:61	Number of fetches	Spec	Counts fetches that write at least one instruction to the IQ. (With instruction fetched (com:4), can used to compute instructions-per-fetch)
Instruction MMU, Data MMU and L2 MMU Events			
Com:62	Instruction MMU TLB4K reloads	Spec	Counts reloads in the level 1 instruction MMU TLB4K.þA reload in the level 2 MMU TLB4Kis not counted.
Com:63	Instruction MMU VSP reloads	Spec	Counts reloads in the level 1 instruction MMU VSP.þA reload in the level 2 MMU VSP is not counted.
Com:64	Data MMU TLB4K reloads	Spec	Counts reloads in the level 1 data MMU TLB4K.þA reload in the level 2 MMU TLB4K is not counted.
Com:65	Data MMU VSP reloads	Spec	Counts reloads in the level 1 data MMU VSP.þA reload in the level 2 MMU VSP is not counted.
Com:66	L2MMU misses	Nonspec	Counts instruction TLB/data TLB error interrupts
BIU Interface Usage			
Com:67	BIU master requests	Spec	Master transaction starts (assertions of $\overline{ts}$
Com:68	BIU master instruction-side requests	Spec	Master instruction-side assertions of $\overline{ts}$
Com:69	BIU master data-side requests	Spec	Master data-side assertions of $\overline{ts}$
Com:70	BIU master data-side castout requests	Spec	Includes replacement pushes and snoop pushes, but not DCBF castouts. ($\overline{ts}$ assertions caused by master data-side non-program-demand castouts)
Com:71	BIU master retries	Spec	Transactions initiated by this processor that were retried on the BIU interface. (The e500 is master and another device retries the e500 transaction.)
Snoop			
Com:72	Snoop requests	N/A	Externally generated snoop requests. (Counts snoop TSs.)
Com:73	Snoop hits	N/A	Snoop hits on all data-side resources regardless of the cache state (modified or exclusive)
Com:74	Snoop pushes	N/A	Snoop pushes from all data-side resources. (Counts snoop ARTRYs and WOPs.)
Com:75	Snoop retries	N/A	Retried snoop requests. (Counts snoop ARTRYs.) (opposite of com 71—another device drives *artry*).
Threshold Events			
C0:76 C1:76	Data line fill buffer load miss cycles	Spec	Instances when the number of cycles between a load allocation in the data line fill buffer (entry 0) and write-back to the data L1 cache exceeds the threshold.
C0:77 C1:77	ILFB fetch miss cycles	Spec	Instances when the number of cycles between allocation in the ILFB (entry 0) and write-back to the instruction L1 cache exceeds the threshold.

Table 7-10. Performance Monitor Event Selection (continued)

Number	Event	Spec/ Nonspec	Count Description
C0:78 C1:78	External input interrupt latency cycles	N/A	Instances when the number of cycles between request for interrupt (*int*) asserted (but possibly masked/disabled) and redirecting fetch to external interrupt vector exceeds threshold.
C0:79 C1:79	Critical input interrupt latency cycles	N/A	Instances when the number of cycles between request for critical interrupt (*cint*) is asserted (but possibly masked/disabled) and redirecting fetch to the critical interrupt vector exceeds threshold.
C0:80 C1:80	External input interrupt pending latency cycles	N/A	Instances when the number of cycles between external interrupt pending (enabled and pin asserted) and redirecting fetch to the external interrupt vector exceeds the threshold. Note that this and the next event may count multiple times for a single interrupt if the threshold is very small and the interrupt is masked a few cycles after it is asserted and later becomes unmasked.
C0:81 C1:81	Critical input interrupt pending latency cycles	N/A	Instances when the number of cycles between pin request for critical interrupt pending (enabled and pin asserted) and redirecting fetch to the critical interrupt vector exceeds the threshold. See note for previous event.
Chaining Events[3]			
Com:82	PMC0 overflow	N/A	PMC0[32] transitions from 1 to 0.
Com:83	PMC1 overflow	N/A	PMC1[32] transitions from 1 to 0.
Com:84	PMC2 overflow	N/A	PMC2[32] transitions from 1 to 0.
Com:85	PMC3 overflow	N/A	PMC3[32] transitioned from 1 to 0.
Interrupt Events			
Com:86	Interrupts taken	Nonspec	—
Com:87	External input interrupts taken	Nonspec	—
Com:88	Critical input interrupts taken	Nonspec	—
Com:89	System call and trap interrupts	Nonspec	—
Ref:90	(e500v2 only) Transitions of TBL bit selected by PMGC0[TBSEL].	Nonspec	Counts transitions of the TBL bit selected by PMGC0[TBSEL].

1 Basic instructions are counted as one micro-op; load and store with update instructions count as one load or store micro-op and one add micro-op; and load or store multiple instructions are counted as from 1–32 load or store micro-ops, depending on how the instruction is encoded.

2 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.

3 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 8
Debug Support

This chapter discusses the debug features of the e500v1 and e500v2 core complex, with particular attention given to the e500 debug facility as an implementation of the Book E–defined debug architecture. Additional debug capabilities associated with an integrated device that implements the e500 core are documented in the reference manual for that device.

References to e500 apply to both the e500v1 and the e500v2.

8.1 Overview

Internal debug mechanisms allow for software and hardware debug by providing debug functions, such as instruction and data breakpoints and program trace mode. e500 debug facilities consist of a set of software-accessible debug registers and interrupt mechanisms largely defined by the Book E PowerPC architecture.

8.2 Programming Model

This section describes the registers, instructions, and interrupts defined by the Book E architecture to support the debug facility.

8.2.1 Register Set

The Book E architecture defines the special-purpose registers (SPRs) listed in Table 8-1 for use with the debug facilities. SPRs not implemented on the e500 are indicated. This table gives cross-references to full descriptions of these SPRs in Chapter 2, "Register Model."

Table 8-1. Debug SPRs

SPR	Name	Defined SPR Number		Access	Supervisor Only	Section/ Page
		Decimal	Binary			
CSRR0	Critical save/restore register 0	58	00001 11010	R/W	Yes	2.7.1.1/2-18
CSRR1	Critical save/restore register 1	59	00001 11011	R/W	Yes	2.7.1.1/2-18
DAC1[1]	Data address compare 1	316	01001 11100	R/W	Yes	2.13.4/2-48
DAC2[1]	Data address compare 2	317	01001 11101			
DBCR0	Debug control register 0	308	01001 10100	R/W	Yes	2.13.1/2-46
DBCR1	Debug control register 1	309	01001 10101	R/W	Yes	
DBCR2	Debug control register 2	310	01001 10110	R/W	Yes	

Table 8-1. Debug SPRs (continued)

SPR	Name	Defined SPR Number		Access	Supervisor Only	Section/ Page
		Decimal	Binary			
DBSR	Debug status register	304	01001 10000	Read/Clear[2]	Yes	2.13.2/2-47
DEAR	Data exception address register	61	00001 11101	R/W	Yes	2.7.1.3/2-18
DEC	Decrementer	22	00000 10110	R/W	Yes	2.6.4/2-16
DECAR	Decrementer auto-reload	54	00001 10110	Write-only		
ESR	Exception syndrome register	62	00001 11110	R/W	Yes	2.7.1.6/2-20
IAC1[1]	Instruction address compare 1	312	01001 11000	R/W	Yes	2.13.3/2-48
IAC2[1]	Instruction address compare 2	313	01001 11001			
IAC3	Instruction address compare 3 (not implemented)	314	01001 11010			
IAC4	Instruction address compare 4 (not implemented)	315	01001 11011			
IVOR15	Debug interrupt offset	415	01100 11111	R/W	Yes	2.7.1.5/2-19

1 Address comparisons only compare effective, not real, addresses.

2 The DBSR is read using **mfspr**. It cannot be directly written to. Instead, DBSR bits corresponding to 1 bits in the GPR can be cleared using **mtspr**.

In addition, Book E defines the debug enable bit in the machine state register, MSR[DE], which must be set for debug events to cause debug interrupts to be taken. This bit is described in Section 2.5.1, "Machine State Register (MSR)." Note that debug interrupts are not affected by the critical enable bit (MSR[CE]).

8.2.2 Instruction Set

The SPRs listed in Table 8-1 are accessed by the **mtspr** and **mfspr** instructions. The MSR is accessed with **mtmsr** and **mfmsr** instructions. Also, the MSR is updated with the contents of CSRR1 when an **rfmci** instruction is executed, typically at the end of an interrupt handler.

8.2.3 Debug Interrupt Model

Book E defines the debug interrupt as a critical class interrupt. Critical class interrupts use a separate pair of save and restore registers (CSRR0 and CSRR1) whose contents are updated when a critical interrupt is taken. The Return from Critical Interrupt (**rfci**) instruction uses these registers to restore state at the end of the interrupt handler. Debug interrupts do not affect the save/restore registers, SRR0 and SRR1, and CSRR registers are not affected by the Return from Interrupt (**rfi**) instruction.

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] = MSR[DE] = 1). CSRR0, CSRR1, MSR, and DBSR are updated as shown in Table 8-2.

Table 8-2. Debug Interrupt Register Settings

Register	Setting
CSRR0	For debug exceptions that occur while debug interrupts are enabled (DBCR0[IDM] = 1 and MSR[DE] = 1), CSRR0 is set as follows: • For instruction address compare (IAC1 and IAC2 debug events), data address compare (DAC1R, DAC1W, DAC2R, and DAC2W debug events), trap, or branch taken debug exceptions, set to the address of the instruction causing the debug interrupt. • For instruction complete debug exceptions, set to the address of the instruction that would have executed after the one that caused the debug interrupt. • For unconditional debug event (UDE) debug exceptions, set to the address of the instruction that would have executed next if the debug interrupt had not occurred. • For interrupt taken debug exceptions, set to the interrupt vector value of the interrupt that caused the interrupt taken debug event. • For return from interrupt (RET) debug exceptions, set to the address of the instruction that would have executed after the **rfi** or **rfci** that caused the debug interrupt. • For debug exceptions that occur while debug interrupts are disabled (DBCR0[IDM] = 0 or MSR[DE] = 0), a debug interrupt occurs at the next synchronizing event if DBCR0[IDM] and MSR[DE] are modified such that they are both set and if the debug exception status is still set in the DBSR. When this occurs, CSRR0 holds the address of the instruction that would have executed next, not with the address of the instruction that modified DBCR0 or MSR and thus caused the interrupt.
CSRR1	Set to the contents of the MSR at the time of the interrupt.
MSR	ME is unchanged. All other MSR bits are cleared.
DBSR	Set to indicate type of debug event (see Chapter 8, “Debug Support”).

Instruction execution resumes at address IVPR[32–47] || IVOR15[48–59] || 0b0000.

8.2.4 Deviations from the Book E Debug Model

The e500 core complex supports Book E debug mode with the following exceptions:

- Instruction address compare registers 3 and 4 (IAC3, IAC4) and data address compare registers 3 and 4 (DAC3, DAC4) along with their debug exceptions, are not implemented.
- Only effective addresses are compared with instruction address compare (IAC1 or IAC2 debug events), and data address compare (DAC1 or DAC2 debug events).
- Return debug events for the **rfci** instruction are not logged if MSR[DE] is cleared (debug interrupts are disabled).

Table 8-3 describes the differences in DBCR0 and DBSR.

Table 8-3. DBCR0 and DBSR Field Differences

Bits	Name	Description
DBCR0[34–35]	RST	Reset 0*x* Default 1*x* A hard reset occurs if MSR[DE] and DBCR0[IDM] are set. Cleared on subsequent cycle.
DBSR[34–35]	MRR	Most recent reset. Undefined at power-up. 0*x* No hard reset occurred since this bit was last cleared by software. 1*x* The previous reset was a hard reset.

8.2.5 Hardware Facilities

The TAP (test access port) unit is a modified IEEE 1149.1 communication interface that facilitates external test and debugging. However, because the core complex is a building block for further integration, it does not contain IEEE 1149.1 standard boundary cells on its I/O periphery, so it should not be considered IEEE 1149.1 compliant.

Private instructions allow an external debugger to freeze or halt the core complex, read and write internal state, and resume normal execution.

8.3 TAP Controller and Register Model

JTAG (joint test action group) is a serial protocol that specifies data flow though special registers connected between test data in (TDI) and test data out (TDO). Figure 8-1 shows the TAP registers implemented by the core complex. For more information, refer to *IEEE Standard Test Access Port and Boundary Scan Architecture IEEE STD 1149-1a-1993*.

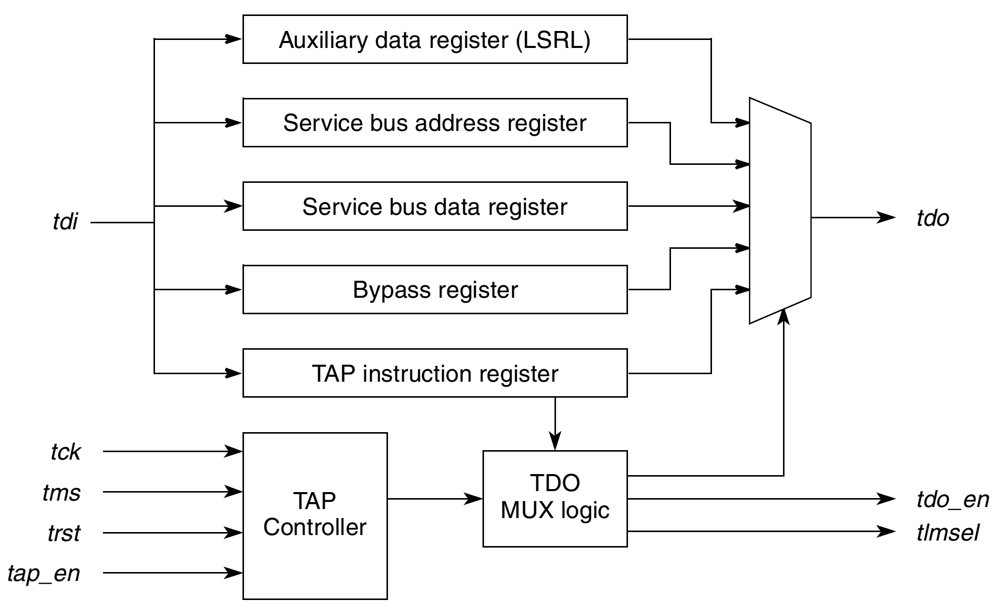


Figure 8-1. TAP Controller with Supported Registers

8.3.1 TAP Interface Signals

The TAP interface signals are summarized in Table 8-4 and discussed briefly in the following sections. The test data input (TDI) and test data output (TDO) scan ports are used to scan instructions and data into the various scan registers for JTAG operations. The scan operation is controlled by the TAP controller, which in turn is controlled by the test mode select (TMS) input sequence. The scan data is latched at the rising edge of test clock (TCK).

The TAP and boundary-scan logic are not used under typical operating conditions. Detailed discussion of all e500 test functions is beyond the scope of this document. However, sufficient information is provided to allow the system designer to disable test functions that would impede normal operation.

Table 8-4. TAP/IEEE/JTAG Interface Signal Summary

Signal Name	Description	Input/Output	IEEE 1149.1a Function
TCK	Test clock	In	Scan clock
TDI	Test data input	In	Serial scan input signal
TDO	Test data output	Out	Serial scan output signal
TMS	Test mode select	In	TAP controller mode signal
$\overline{\text{TRST}}$	Test reset	In	TAP controller reset
TAP_EN	TAP enable	In	N/A
TDO_EN	Test data output enable	Out	N/A
TLMSEL	TLM selected	Out	N/A

Test reset ($\overline{\text{TRST}}$) is an optional JTAG signal used in the e500 to reset the TAP controller asynchronously. This signal is not used during normal operation. It is recommended that $\overline{\text{TRST}}$ be asserted and negated coincident with the assertion of $\overline{\text{HRESET}}$ to ensure that the test logic does not affect normal operation of the core complex.

$\overline{\text{TRST}}$ must be asserted sometime during power-up for JTAG logic initialization. Note that if $\overline{\text{TRST}}$ is connected low, unnecessary power is consumed.

Table 8-5 describes JTAG signals in detail.

Table 8-5. JTAG Signal Details

Signal	I/O	Description
TCK	I	JTAG test clock. Primary clock input for the test logic on the e500. May be asynchronous with respect to all other core complex clocks. **State Meaning** Asserted/Negated—This input should be driven by a free-running clock signal. Input signals to the test access port are sampled on the rising edge of TCK. TAP output signal changes occur on the falling edge of TCK. The test logic allows TCK to be stopped.
TDI	I	JTAG test data input. Primary JTAG data input to both scan chain and test control registers. **State Meaning** Asserted/Negated—The value present on the rising edge of TCK is loaded into the selected JTAG test instruction or data register.
TDO	O	JTAG test data output. Primary JTAG data output. **State Meaning** Asserted/Negated—The contents of the selected internal instruction or data register are shifted out onto this signal. Valid data appears on the falling edge of TCK. Quiescent except when scanning of data is in progress.
TMS	I	JTAG test mode select. Primary JTAG mode control input. **State Meaning** Asserted/Negated—Decoded by the internal JTAG TAP controller to determine the primary operation of the test support circuitry.
$\overline{\text{TRST}}$	I	JTAG test reset. JTAG initialization input. **State Meaning** Asserted—Causes asynchronous initialization of the internal JTAG test access port controller. Must be asserted sometime during the assertion of $\overline{\text{HRESET}}$ to properly initialize the JTAG test access port. Negated—Indicates normal operation.
TAP_EN	I	TAP enable. Used by the TAP linking module (TLM) logic external to the core complex to select the core complex TAP module. When there is no TLM connected to the TAP, the TAP_EN is connected high via an internal pull-up resistor. **State Meaning** Asserted—A valid TMS signal is applied to the TAP controller. Negated—A valid TMS signal is not being applied to the TAP controller.
TDO_EN	O	TDO enable. Provides feedback to the external TAP linking module logic. **State Meaning** Asserted—Valid data is available on TDO. Negated—Value of TDO is meaningless.
TLMSEL	O	TLM selected. Provides feedback to the external TAP linking module logic. **State Meaning** Asserted—The core complex is currently executing a TLM TAP instruction. Negated—The core complex is not currently executing a TLM TAP instruction

8.4 Book E Debug Events

Debug events cause debug exceptions to be recorded in the DBSR (see Section 2.13.2, “Debug Status Register (DBSR)”). Except for an unconditional debug event, the specific event type must be enabled by corresponding bits in the debug control registers (DBCR0–DBCR2) for any debug event to set a DBSR bit and thereby cause a debug exception. Setting a DBSR bit causes a debug interrupt only if debug interrupts are enabled.

If interrupts are disabled, some debug events are not recorded; that is, no DBSR bit is set by the event. However, some debug events can cause exceptions and set DBSR bits regardless of the state of MSR[DE]. Interrupts resulting from such exceptions are delayed until MSR[DE] is set (unless they have been cleared from the DBSR in the meantime).

Any time a DBSR bit can be set while MSR[DE] is cleared, the imprecise debug event bit (DBSR[IDE]) is also set. IDE indicates whether the associated DBSR bit was set while debug interrupts were disabled. Debug interrupt handler software can use this bit to interpret the address in CSRR0. If IDE is zero, CSRR0 holds the address of the instruction causing the debug exception; otherwise, it holds the address of the instruction following the one that enabled the delayed debug interrupt.

Debug exceptions are prioritized with respect to other exceptions (see Section 5.11.1, "e500 Exception Priorities").

Table 8-6 lists the types of debug events, which are discussed in subsequent sections.

Table 8-6. Debug Events

Event Type	Description	Section
Instruction address compare	Each instruction address is compared in a specific way with a specific value. A debug event occurs when they match.	8.4.1
Data address compare	Each data address is compared with a value. A debug event occurs when they match.	8.4.2
Trap	A debug event occurs when a trap is set.	8.4.3
Branch taken	A debug event occurs when any branch is taken.	8.4.4
Instruction complete	A debug event occurs when any instruction completes.	8.4.5
Interrupt taken	A debug event occurs when an interrupt is taken.	8.4.6
Return	A debug event occurs when a return from interrupt occurs.	8.4.7
Unconditional	A debug event occurs whenever this instruction is executed.	8.4.8

8.4.1 Instruction Address Compare Debug Event

One or more instruction address compare debug events (IAC1 and IAC2) occur if they are enabled and execution is attempted of an instruction at an address that meets the criteria specified in DBCR0, DBCR1, and the IAC registers.

8.4.1.1 Instruction Address Compare User and Supervisor Modes

The debug control registers specify user and supervisor modes as follows:

- DBCR1[IAC1US] specifies whether IAC1 debug events can occur in user mode, in supervisor mode, or in both.
- DBCR1[IAC2US] specifies whether IAC2 debug events can occur in user mode, in supervisor mode, or in both.

8.4.1.2 Effective Address Mode

The debug control registers specify effective address modes as follows:

- DBCR1[IAC1ER] specifies whether effective addresses alone, effective addresses and MSR[IS] cleared, or effective addresses and MSR[IS] set are used in determining an address match on IAC1 debug events.
- DBCR1[IAC2ER] specifies whether effective addresses alone, effective addresses and MSR[IS] cleared, or effective addresses and MSR[IS] set are used in determining an address match on IAC2 debug events.

8.4.1.3 Instruction Address Compare Mode

The debug control registers specify instruction address compare modes as follows:

- DBCR1[IAC12M] specifies the following:
  — Whether all or some of the bits of the address of the instruction fetch must match the contents of IAC1 or IAC2
  — Whether the address must be inside or outside of a specific range specified by IAC1 and IAC2 to trigger a corresponding debug event.

The four instruction address compare modes are described in Table 8-7.

Table 8-7. Instruction Address Compare Modes

Mode	Instruction Address Match Condition
Exact address compare	The fetch address equals the value in the enabled IAC register.
Address bit match	For IAC1 and IAC2 debug events, if the fetch address, ANDed with the contents of IAC2, is equal to the contents of IAC1, also ANDed with the contents of IAC2.
Inclusive address range compare mode	For IAC1 and IAC2 debug events, if the fetch address is greater than or equal to the contents of IAC1 and less than the contents of IAC2.
Exclusive address range compare mode	For IAC1 and IAC2 debug events, if the instruction fetch address is less than the contents of IAC1 or greater than or equal to the contents of IAC2.

Section 2.13.1, “Debug Control Registers (DBCR0–DBCR2),” describes DBCR0 and DBCR1 and modes for detecting IAC register debug events. Instruction address compare debug events can occur regardless of the values of MSR[DE] or DBCR0[IDM].

When an instruction address compare debug event occurs, the corresponding DBSR[IAC*n*] bits are set to record the debug exception. If MSR[DE] is cleared, DBSR[IDE] is also set to capture the imprecise debug event.

If MSR[DE] is set at the time of the instruction address compare debug exception, a debug interrupt occurs immediately (if no higher priority exception has caused an interrupt). Execution of the instruction causing the exception is suppressed, and CSRR0 is set to the address of the excepting instruction.

If MSR[DE] is cleared at the time of the instruction address compare debug exception, a debug interrupt does not occur and the instruction completes execution (provided the instruction is not causing another exception that generates an enabled interrupt).

Later, if the debug exception has not been reset by clearing the appropriate DBSR[IAC*n*], bits and MSR[DE] is set, a delayed debug interrupt occurs. In this case, CSRR0 contains the address of the instruction following the one that set DE. Software in the debug interrupt handler can observe DBSR[IDE] to determine how to interpret the CSRR0 value.

8.4.2 Data Address Compare Debug Event

One or more data address compare debug events (DAC1R, DAC1W, DAC2R, or DAC2W) can occur if they are enabled, execution of a data access instruction is attempted, and the type, address, and possibly even the data value of the data access meet the criteria specified in DBCR0, DBCR2, DAC1, and DAC2.

8.4.2.1 Data Address Compare Read/Write Enable

DBCR0[DAC1] specifies whether DAC1R debug events can occur on read-type data accesses and whether DAC1W debug events can occur on write-type data accesses.

DBCR0[DAC2] specifies whether DAC2R debug events can occur on read-type data accesses and whether DAC2W debug events can occur on write-type data accesses.

All load instructions are considered reads with respect to debug events, and all store instructions are considered writes with respect to debug events. In addition, cache management instructions, and certain special cases, are handled as follows.

- **dcbt**, **dcbtst**, **icbt**, and **icbi** are all considered reads with respect to debug events. Note that **dcbt**, **dcbtst**, 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 are allowed to cause debug interrupts, even when no-op would have been asserted due to a data storage or data TLB miss exception.
- **dcbz**, **dcbi**, **dcbf**, and **dcbst** 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 change the data at a given address. However, because execution of these instructions may generate write activity on the processor’s data bus, they are treated as writes with respect to debug events.

8.4.2.2 Data Address Compare User/Supervisor Mode

User/supervisor mode options in data address compare debug events occur as follows:

- DBCR2[DAC1US] specifies whether DAC1R and DAC1W debug events can occur in user mode, supervisor mode, or both.
- DBCR2[DAC2US] specifies whether DAC2R and DAC2W debug events can occur in user mode, supervisor mode, or both.

8.4.2.3 Effective Address Mode

Effective address mode options in debug events occur as follows:

- DBCR2[DAC1ER] specifies whether effective addresses alone, effective addresses and MSR[DS] cleared, or effective addresses and MSR[DS] set, are used to determine an address match on DAC1R and DAC1W debug events.
- DBCR2[DAC2ER] specifies whether effective addresses alone, effective addresses and MSR[DS] cleared, or effective addresses and MSR[DS] set, are used to determine an address match on DAC2R and DAC2W debug events.

8.4.2.4 Data Address Compare (DAC) Mode

DBCR2[DAC12M] specifies the following:

- Whether all or some of the address bits for the data access must match the contents of DAC1 or DAC2
- Whether the address must be inside or outside of a range specified by DAC1 and DAC2 for a DAC1R, DAC1W, DAC2R, or DAC2W debug event to occur.

Table 8-8 describes the four data address compare modes.

Table 8-8. Data Address Compare Modes

Mode Name	Data Address Match Condition
Exact address compare	The data access address is equal to the value in the enabled DAC*n*.
Address bit match	The data access address, ANDed with the contents of DAC2, is equal to the contents of DAC1, also ANDed with the contents of DAC2.
Inclusive address range compare	The data access address is greater than or equal to the contents of DAC1 and less than the contents of DAC2.
Exclusive address range compare	The data access address is less than the contents of DAC1 or greater than or equal to the contents of DAC2.

Section 2.13.1, "Debug Control Registers (DBCR0–DBCR2)," describes DBCR0 and DBCR2 and the modes for detecting DAC debug events, which can occur regardless of the values of MSR[DE] or DBCR0[IDM]. When a DAC debug event occurs, the corresponding DBSR bit (DAC1R, DAC1W, DAC2R, or DAC2W) is set to record the exception.

If MSR[DE] is cleared, DBSR[IDE] is set to capture the imprecise debug event. However, if DE is set, a DAC debug exception causes the following events:

- A debug interrupt is taken immediately (if no higher priority exception has caused an interrupt).
- Execution of the instruction causing the exception is suppressed.
- CSRR0 is loaded with the address of the excepting instruction.

Depending on the type of instruction and the alignment of the access, the instruction causing the exception may have been partially executed (see Section 5.9, “Partially Executed Instructions”).

If debug interrupts are disabled when a DAC debug exception occurs, no interrupt is taken and the instruction completes normally (provided the instruction is not causing some other exception that generates an enabled interrupt). Also, DBSR[IDE] is set to indicate that the exception occurred while debug interrupts were disabled.

Later, if MSR[DE] is set and the debug exception has not been reset by clearing the appropriate DBSR bit (DAC1R, DAC1W, DAC2R, or DAC2W), a delayed debug interrupt occurs. In this case, CSRR0 contains the address of the instruction following the instruction that enabled the debug interrupt. The debug interrupt handler can observe DBSR[IDE] to determine how to interpret the CSRR0 value.

8.4.3 Trap Debug Event

A trap debug event occurs if DBCR0[TRAP] is set (trap debug events are enabled) and a trap instruction (**tw** or **twi**) is executed and the trap conditions specified by the instruction are met. The event can occur regardless of the values of MSR[DE] or DBCR0[IDM].

When a trap debug event occurs, DBSR[TRAP] is set to capture the debug exception. If MSR[DE] is cleared, DBSR[IDE] is also set to record the imprecise debug event.

If MSR[DE] is set at the time of the trap debug exception, a debug interrupt occurs immediately (if no higher priority exception has caused an interrupt), and CSRR0 is set to the address of the excepting instruction.

If debug interrupts are disabled at the time of the exception, no interrupt is taken and a trap exception type program interrupt occurs.

Later, if MSR[DE] is set, and the debug exception has not been reset by clearing DBSR[TRAP], a delayed debug interrupt occurs. In this case, CSRR0 contains the address of the instruction following the one that enabled the debug interrupt (by setting MSR[DE]). The debug interrupt handler can observe DBSR[IDE] to determine how to interpret the CSRR0 value.

8.4.4 Branch Taken Debug Event

A branch taken debug event occurs if both MSR[DE] and DBCR0[BRT] are set (branch taken debug events are enabled) and execution is attempted of a branch instruction whose direction is taken (an unconditional branch or a conditional branch whose branch condition is met).

Because branch instructions occur very frequently, branch taken debug events are not recognized if MSR[DE] is cleared when the branch instruction executes and thus DBSR[IDE] cannot be set by a branch taken debug event. Allowing these common events to be recorded as exceptions in the DBSR while debug interrupts are disabled would cause an inordinate number of imprecise debug interrupts.

The following actions are taken when a branch taken debug event occurs:

- DBSR[BRT] is set (to capture the debug exception).
- A debug interrupt occurs immediately (if no higher priority exception has caused an interrupt).
- Execution of the exception-causing instruction is suppressed.
- CSRR0 is set to the address of the excepting instruction.

8.4.5 Instruction Complete Debug Event

An instruction complete debug event occurs when any instruction completes execution so long as MSR[DE] and DBCR0[ICMP] are both set (instruction complete debug events are enabled). Note that no instruction complete debug event occurs if execution of an instruction is suppressed because it caused some other interrupt-generating exception. The **sc** instruction does not fall into the category of an instruction whose execution is suppressed, because the instruction actually completes execution and then generates a system call interrupt. In this case, the instruction complete debug exception is also set.

Instruction complete debug events are not recognized if MSR[DE] is cleared at the time of the instruction execution. DBSR[IDE] cannot be set by an instruction complete debug event because allowing the common instruction completion event to log an exception in the DBSR while debug interrupts are disabled would cause the debug interrupt handler software to receive an inordinate number of imprecise debug interrupts whenever debug interrupts were reenabled.

The following actions are taken when an instruction complete debug event occurs:

- DBSR[ICMP] is set (to record the debug exception).
- A debug interrupt occurs immediately (if no higher priority exception has caused an interrupt).
- CSRR0 is set to the address of the instruction following the one that caused the instruction complete debug exception.

8.4.6 Interrupt Taken Debug Event

An interrupt taken debug event occurs if DBCR0[IRPT] is set (interrupt taken debug events are enabled) and a noncritical interrupt occurs. Interrupt taken debug events can occur regardless of the value of MSR[DE].

Only noncritical interrupts can cause an interrupt taken debug event because all critical interrupts automatically clear DE and thus would always prevent the associated debug interrupt from occurring precisely. Also, debug interrupts themselves are critical interrupts, so any additional debug interrupt (for a second debug event) would always set the additional DBSR[IRPT] exception when it entered the debug interrupt handler. At this point, the debug interrupt handler could not determine if the second interrupt taken debug event was related to the original event.

When an interrupt taken debug event occurs, IRPT is set to capture the debug exception. If DE is zero, DBSR[IDE] is also set to record the imprecise debug event. If DE is set at the time of the event, the following occurs:

- A debug interrupt occurs immediately if no higher priority exception caused an interrupt.
- CSRR0 is set to the address of the noncritical interrupt vector that caused the event. No instructions at the noncritical interrupt handler are executed.

If debug interrupts are disabled when the event occurs, no interrupt is generated. However, if the debug exception has not been reset by clearing DBSR[IRPT], a delayed debug interrupt occurs when interrupts are reenabled (MSR[DE] is set). In this case, CSRR0 contains the address of the instruction following the one that set DE. The interrupt handler can observe DBSR[IDE] to determine how to interpret CSRR0.

8.4.7 Return Debug Event

A return debug event occurs if DBCR0[RET] is set (enabling return debug events) and an attempt is made to execute an **rfi**. Results from executing an **rfci** while RET is set are implementation dependent; the e500 does the following:

- If MSR[DE] is set, a debug interrupt is generated.
- If DE is cleared, no debug interrupt is generated and no debug event is logged.

When a return debug event occurs, DBSR[RET] is set to capture the debug exception. If MSR[DE] is cleared when **rfi** executes (before the MSR is updated by the **rfi**), DBSR[IDE] is also set to record the imprecise debug event. If DE is set at the time of the return debug exception, the following events occur:

- A debug interrupt is taken immediately (unless the **rfi** or **rfci** causing the event clears MSR[DE] or a higher priority exception has caused an interrupt).
- CSRR0 is loaded with the address of the instruction that would have executed next had the interrupt not occurred.

If DE is zero (either at the time of the execution of the **rfi** or after the MSR is updated by the **rfi**) at the time of the return debug exception, a debug interrupt does not occur.

Provided the debug exception has not been reset by clearing DBSR[RET], a delayed imprecise debug interrupt occurs when MSR[DE] is set. In this case, CSRR0 contains the address of the instruction following the one that set MSR[DE]. The interrupt handler can observe DBSR[IDE] to determine how to interpret the value in CSRR0 unless MSR[DE] was cleared by the **rfi**. In that case, DBSR[IDE] has not been set and the software cannot determine that the interrupt was precise.

8.4.8 Unconditional Debug Event

An unconditional debug event occurs when the debug mechanism asserts the *ude* signal. The exact definition of *ude* and how it is activated are implementation dependent. See the reference manual for the device that implements the e500 core for details. An unconditional debug event can occur regardless of the value of MSR[DE] and is the only debug event that does not have a corresponding debug control register enable bit.

If MSR[DE] is set, an unconditional debug event causes the following:

- A debug interrupt is taken immediately, if no higher priority exception caused an interrupt.
- CSRR0 is loaded with the address of the instruction that would have executed next had the interrupt not occurred.

When an unconditional debug event occurs, DBSR[UDE] is set to record the exception. If the event occurs while debug interrupts are disabled, DBSR[IDE] is set and the interrupt is delayed until MSR[DE] is set, provided the exception has not been cleared from the DBSR in the meantime. IDE indicates whether the associated DBSR exception bit was set while debug interrupts were disabled. Debug interrupt handler software can use this bit to determine whether the address recorded in CSRR0 should be interpreted as the address associated with the instruction causing the debug exception or is simply the address of the instruction after the one that set MSR[DE], thereby enabling the delayed debug interrupt.

Part II
e500 Core Complex

This part describes the features of the e500 core complex that comprise its memory subsystem and auxiliary features. It contains the following chapters:

- Chapter 9, "Timer Facilities," describes the Book E-defined timer facilities implemented in the e500 core. These resources include the time base (TB), decrementer (DEC), fixed-interval timer (FIT), and watchdog timer.
- Chapter 10, "Auxiliary Processing Units (APUs)," describes APUs implemented on the e500, such as the **isel** instruction, performance monitor, signal processing engine, branch target buffer (BTB) locking, cache block lock and unlock, and machine check APUs.
- Chapter 11, "L1 Caches," describes the organization of the on-chip level-one instruction and data caches, cache coherency protocols, cache control instructions, and various cache operations. It describes the interaction that occurs in the memory subsystem, which consists of the memory management unit (MMU), caches, load/store unit (LSU), and core complex bus (CCB). The chapter also describes the replacement algorithms used for each of the L1 caches.
- Chapter 12, "Memory Management Units," describes the implementation details of the e500 core complex MMU relative to the Book E architecture and the Motorola Book E standards.
- Chapter 13, "Core Complex Bus (CCB)," describes those aspects of the CCB that are configurable or that provide status information through the programming interface. It provides a glossary of those signals that are mentioned in other chapters to offer a clearer understanding of how the core is integrated as part of a larger device.

Chapter 9
Timer Facilities

This chapter describes specific implementation details of the e500v1 and e500v2 implementations of the Book E–defined timer facilities. These resources, which include the time base (TB), decrementer (DEC), fixed-interval timer (FIT), and watchdog timer, are described in detail in the *EREF: A Reference for Freescale Book E and the e500 Core*.

Section 9.3.2, "Performance Monitor Time Base Event," describes the time base event implemented by the e500v2 performance monitor.

9.1 Timer Facilities

The TB, DEC, FIT, and watchdog timer provide timing functions for the system. All of these must be initialized during start-up.

- The TB provides a long-period counter driven by a frequency that is implementation dependent.
- The decrementer, a counter that is updated at the same rate as the TB, provides a means of signaling an exception after a specified amount of time has elapsed unless one of the following occurs:
  - DEC is altered by software in the interim.
  - The TB update frequency changes.

  The DEC is typically used as a general-purpose software timer.
- The clock source for the TB and the DEC is specified by two fields in HID0: time base enable (TBEN), and select time base clock (SEL_TBCLK). If the TB is enabled (HID0[TBEN] = 1) the clock source is determined as follows:
  - If [SEL_TBCLK] = 0, the TB is updated every 8 core complex bus (CCB) clocks.
  - If HID0[SEL_TBCLK] = 1, the time base is updated on the rising edge of *tbclk* (or a clock input specified by the implementation). The exact frequency range is specified in the hardware specification for the integrated device, but the maximum value should not exceed 1/ 8th the core frequency.

  See Section 2.10.1, "Hardware Implementation-Dependent Register 0 (HID0)."
- The fixed-interval timer is essentially a selected bit of the TB, which provides a means of signaling an exception whenever the selected bit transitions from 0 to 1, in a repetitive fashion. The fixed-interval timer is typically used to trigger periodic system maintenance

functions. Software may select one of four bits in the TB to serve as the fixed-interval timer. Which bits may be selected depends on the implementation.

- The watchdog timer is also a selected bit of the TB, which provides a means of signalling a critical class exception whenever the selected bit transitions from 0 to 1. In addition, if software does not respond in time to the initial exception (by clearing the associated status bits in the TSR before the next expiration of the watchdog timer interval), then a watchdog timer-generated processor reset may result, if so enabled. The watchdog timer is typically used to provide a system error recovery function.

The relationship of these timer facilities to each other is shown in Figure 9-1.

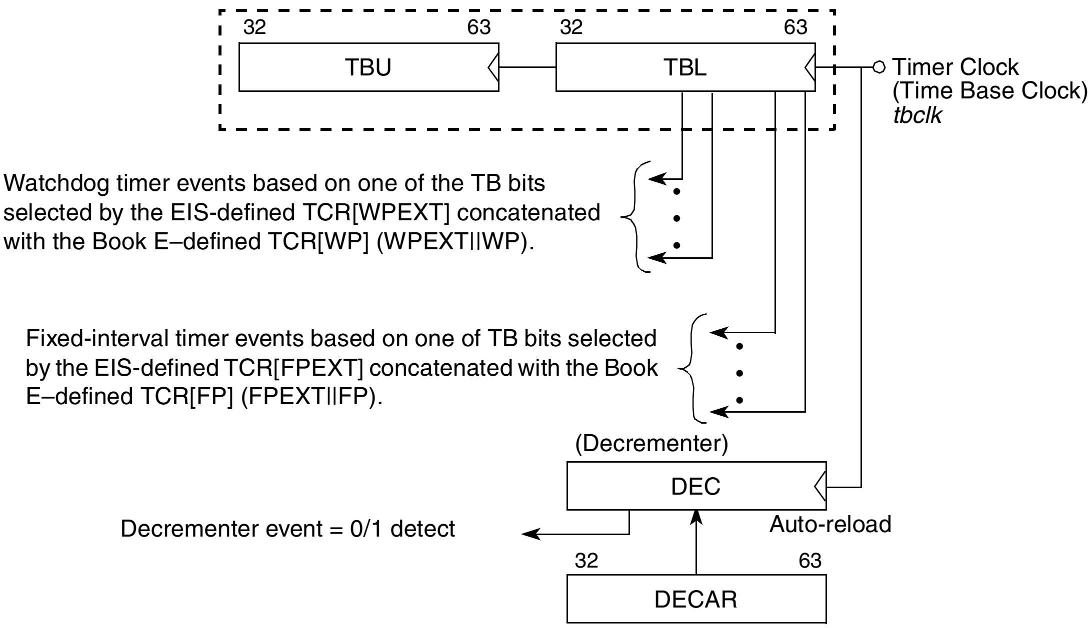


Figure 9-1. Relationship of Timer Facilities to Time Base

9.2 Timer Registers

This section describes registers used by the timer facilities.

- HID0—Clock source select and enable: The clock source for the core timer facilities is specified by two fields in the hardware implementation-dependent register 0 (HID0): time base enable (TBEN), and select time base clock (SEL_TBCLK). HID0[TBEN] enables the time base, and HID0[SEL_TBCLK] selects the time base clock, *tbclk*. (Some implementations may use a signal with a different name.) For more information, see Section 2.10.1, “Hardware Implementation-Dependent Register 0 (HID0).” Section 9.3, “The e500 Timer Implementation,” describes how these bits interact with other registers.
- Timer control register (TCR). Provides control information for the on-chip timer of the core complex. The core complex implements two fields not specified in Book E: TCR[WPEXT]

and TCR[FPEXT]. The TCR controls decrementer, fixed-interval timer, and watchdog timer options.

Section 2.6.1, "Timer Control Register (TCR)," describes the TCR in detail.

- Timer status register (TSR). Contains status on timer events and the most recent watchdog-timer-initiated processor reset. Section 2.6.2, "Timer Status Register (TSR)," describes the TSR in detail.
- Decrementer register (DEC). DEC contents can be read into bits 32–63 of a GPR using **mfspr**, clearing bits 0–31. GPR contents can be written to the decrementer using **mtspr**. See Section 2.6.4, "Decrementer Register (DEC)," for more information.
- Decrementer auto-reload register (DECAR). Supports the auto-reload feature of the decrementer. The DECAR contents cannot be read. See Section 2.6.5, "Decrementer Auto-Reload Register (DECAR)," for more information.

9.3 The e500 Timer Implementation

The clock source for the e500 timer facilities is specified by two fields in HID0: time base enable (TBEN) and select time base clock (SEL_TBCLK). If HID0[TBEN] = 0, the time base is static; there is no counting. If the time base is enabled (HID0[TBEN] is set), the clock source is determined as follows:

- If HID0[SEL_TBCLK] = 0, the timer facilities are updated every 8 CCB clocks.
- If HID0[SEL_TBCLK] = 1, the timer facilities are updated on the rising edge of RTC.

The default source is the CCB clock divided by eight. For more details see Section 2.10.1, "Hardware Implementation-Dependent Register 0 (HID0)."

- If HID0[TBEN] = 0, the time base is static (no counting)
- If HID0[TBEN] = 1 and HID0[SEL_TBCLK] = 0, the time base is updated every 8 bus clocks
- If HID0[TBEN] = 1 and HID0[SEL_TBCLK] = 1, the time base is sampled at the bus rate; that is, it is updated on the rising edge of *tbclk*. (Some implementations may use a signal with a different name.) The maximum supported frequency can be found in the electrical specifications, but this value is approximately 25% of the bus clock frequency.

The decrementer, TBL, and TBU are updated in that order during three successive internal processor clock cycles.

The core output signals $\overline{wrs}$[0:1] reflect the value of TSR[WRS]. The intention is to signal to the system that a watchdog reset event has occurred. The system can then implement a reset strategy. The core can be reset by asserting $\overline{hreset}$. No automatic resetting is done when a watchdog reset occurs.

9.3.1 Alternate Time Base APU

The alternate time base APU defines a time base counter similar to the time base defined in PowerPC architecture. It is intended to be used for measuring time in implementation-defined intervals. It differs from the PowerPC defined time base in that it is not writable, it counts at a different frequency, and it always counts up, wrapping when the 64-bit count overflows.

The alternate time base is a 64-bit counter that counts up at an implementation-dependent rate. While not required, the rate is encouraged to be at the core clock frequency or as small a multiple of the frequency as practical for the implementation. On the e500v2, this frequency is the core frequency.

The ATBU and ATBL registers can be read by executing an **mfspr** instruction, but cannot be written. Reading the ATB (or ATBL) register places the lower 32 bits of the counter into the target register. A second SPR, ATBU, is defined that accesses only the upper 32 bits of the counter. Thus the upper 32 bits of the counter may be read into a register by reading the ATBU register regardless of computation mode.

The ATB registers are described in Section 2.6.6, "Alternate Time Base Registers (ATBL and ATBU)."

The effect of power-savings mode or core frequency changes on counting in the alternate time base is implementation dependent. See the User's Manual for details.

9.3.2 Performance Monitor Time Base Event

The e500v2 has added 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 7, "Performance Monitor."

Chapter 10
Auxiliary Processing Units (APUs)

This chapter describes the e500 APU support. It fully describes those APUs that are specific to the e500 and the double-precision floating-point APU implemented on the e500v2. Full descriptions of the APUs defined by the Freescale Book E implementation standards (EIS) are provided in the *EREF: A Reference for Freescale Book E and the e500 Core* (EREF).

References to e500 apply to both e500v1 and e500v2.

10.1 Overview

The e500 supports the following APUs defined by the EIS:

- Integer select APU
- Performance monitor APU
- Signal processing engine APU (SPE APU)
- Embedded floating-point APUs
  - Embedded vector single-precision floating-point APU
  - Embedded scalar single-precision floating-point APUs
  - Embedded scalar double-precision floating-point APUs. See 10.4, "Double-Precision Floating-Point APU (e500 v2 Only)."

  Note that the e500 diverges from the architected definition provided in the EREF. Details are provided in Section 3.8.1.4, "Embedded Floating-Point APU Instructions," and in Section 2.5.1, "Machine State Register (MSR)."
- Cache block lock and unlock APU
- Machine check APU
- The e500v2 supports the alternate time base APU, described in Section 10.3, "Alternate Time Base APU."

Note that the SPE APU and the two single-precision floating-point APUs were combined in the original implementation of the e500v1, as shown in Figure 10-1.

	Vector and Floating-Point APUs	e500 v1	e500 v2
Original SPE Definition	SPE vector instructions **ev**…	√	√
	Vector single-precision floating-point **evfs**…	√	√
	Scalar single-precision floating-point **efs**…	√	√
	Scalar double-precision floating-point **efd**…		√

Figure 10-1. Vector and Floating-Point APUs

The e500 also implements the branch target buffer (BTB) locking APU, which is not defined by the EIS. See Section 10.2, “Branch Target Buffer (BTB) Locking APU.”

10.2 Branch Target Buffer (BTB) Locking APU

The core complex provides a 512-entry BTB for efficient processing of branch instructions. The BTB is a branch target address cache, organized as 128 rows with four-way set associativity, that holds the address and target instruction of the 512 most-recently taken branches, each with a 2-bit, dynamically updated branch history table that indicates four levels of likelihood that the branch will be taken (strongly taken, taken, not taken, strongly not taken). The BTB provides quick access to branch targets and history bits that allow efficient branch prediction.

The core complex also provides support for locking and unlocking BTB entries for deterministic branch behavior. In particular, the BTB locking APU gives the user the ability to lock, unlock, and invalidate BTB entries.

10.2.1 BTB Locking APU Programming Model

The BTB locking APU defines additional instructions and register resources, which are described in the following sections. It does not define additional interrupts.

10.2.1.1 BTB Locking APU Instructions

Table 10-1 lists the BTB locking instructions, which are described in detail in Section 3.9.1, “Branch Target Buffer (BTB) Locking Instructions.”

Table 10-1. BTB Locking APU Instructions

Name	Mnemonic	Syntax
Branch Buffer Load Entry and Lock Set	**bblels**	—
Branch Buffer Entry Lock Reset	**bbelr**	—

10.2.1.2 BTB Locking APU Registers

The BTB APU register model includes the following register resources for enabling the locking and unlocking of BTB entries:

- Branch unit control and status register (BUCSR)—SPR 1013. This register has bits that are used to enable or disable BTB locking and to control unlocking, invalidation, and overlocking of BTB entries. See Section 2.9.3, “Branch Unit Control and Status Register (BUCSR).”
- Branch buffer entry address register (BBEAR)—SPR 512. This register holds the address of a BTB entry. See Section 2.9.1, “Branch Buffer Entry Address Register (BBEAR).”
- Branch buffer target address register (BBTAR)—SPR 513. This register includes branch target address bits and a field that allows the programmer to specify whether a branch should be predicted as taken or not taken. See Section 2.9.2, “Branch Buffer Target Address Register (BBTAR).”
- MSR[UBLE], the user branch locking enable bit, determines whether user mode programs can lock or unlock BTB entries. See Section 2.5.1, “Machine State Register (MSR).”

10.3 Alternate Time Base APU

The alternate time base APU defines a time base counter similar to the time base defined in PowerPC architecture. It is intended to be used for measuring time in implementation-defined intervals. It differs from the PowerPC defined time base in that it is not writable, it counts at a different frequency, and it always counts up, wrapping when the 64-bit count overflows.

10.3.1 Programming Model

The alternate time base is a 64-bit counter that counts up at an implementation-dependent rate. While not required, the rate is encouraged to be at the core clock frequency or as small a multiple of the frequency as practical for the implementation. On the e500v2, this frequency is the core frequency.

The ATBU and ATBL registers can be read by executing a **mfspr** instruction, but cannot be written. Reading the ATB (or ATBL) register places the lower 32 bits of the counter into the target register. A second SPR, ATBU, is defined that accesses only the upper 32 bits of the counter. Thus the upper 32 bits of the counter may be read into a register by reading the ATBU register regardless of computation mode.

ATB registers are described in Section 2.6.6, “Alternate Time Base Registers (ATBL and ATBU).”

The effect of power-savings mode or core frequency changes on counting in the alternate time base is implementation-dependent. See the User’s Manual for details.

10.4 Double-Precision Floating-Point APU (e500 v2 Only)

This section describes the double-precision floating-point APU. The vector and scalar floating-point APUs are described in the EREF.

Except where otherwise noted, the double-precision floating-point APU adheres to the embedded floating-point APUs programming model and notation conventions as described in the EREF.

10.4.1 Programming Model

Floating-point double-precision instructions operate on the entire 64 bits of the GPRs where a floating-point data item consists of 64 bits. The double-precision floating-point APU uses the thirty-two 64-bit GPRs, which is also used by the vector single-precision floating-point APU and the signal-processing engine (SPE) APU.

There are no record forms of embedded floating-point instructions. Floating-point compare instructions treat NaNs, Infinity and Denorm as normalized numbers for the comparison calculation when default results are provided.

- SPE floating-point status and control register (SPEFSCR)—Double-precision floating-point operations use the SPEFSCR as it is described in the EREF. Double-precision floating-point instructions affect only the low element floating-point status flags and leave the high element floating-point status flags undefined.
- Embedded floating-point exception bit in ESR. The double-precision floating-point APU is affected by the embedded floating-point exception bit, ESR[SPE], as it is described in the EREF. This bit is set whenever the processor takes an interrupt related to the execution of the embedded floating-point instructions.

The double-precision floating-point APU can generate the following embedded floating-point APU interrupts as described in the EREF:

- SPE/embedded floating-point unavailable interrupt—IVOR32 (SPR 528)
- Embedded floating-point data interrupt—IVOR33 (SPR 529)
- Embedded floating-point round interrupt—IVOR34 (SPR 530)

10.4.2 Double-Precision Floating-Point APU Operations

This section describes operational modes and formats. Note that IEEE 754–compliance and sticky bit handling for exception conditions is as described in the EREF.

10.4.2.1 Operational Modes

Double-precision floating-point operations are governed by the setting of the mode bit in SPESCR. The mode bit defines how floating-point results are computed and how floating-point exceptions are handled. Mode 0 defines a real-time, default results–oriented mode that saturates results. No other modes are currently defined.

10.4.2.2 Floating-Point Data Formats

As shown in Figure 10-2, double-precision floating-point data elements are 64 bits wide with 1 sign bit (*s*), 11 bits of biased exponent (*e*) and 52 bits of fraction (*f*).

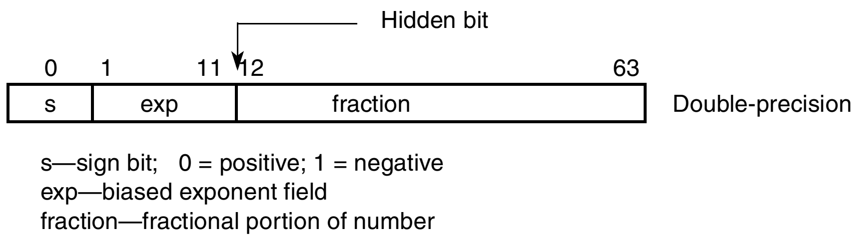


Figure 10-2. Floating-Point Data Format

For double-precision normalized numbers, the biased exponent value ‘*e*’ lies in the range of 1 to 2046 corresponding to an actual exponent value E in the range -1022 to +1023. With the hidden bit implied to be ‘1’ (for normalized numbers), the value of the number is interpreted as follows:

$$(-1)^{s} \times 2^{E} \times (1.\mathrm{fraction})$$

where E is the unbiased exponent and 1.fraction is the mantissa (or significand) consisting of a leading ‘1’ (the hidden bit) and a fractional part (fraction field). The maximum positive normalized number (*pmax*) is represented by the encoding 0x7FEF_FFFF_FFFF_FFFF which is approximately 1.8E+307 (2^{1024}), and the minimum positive normalized value (pmin) is represented by the encoding 0x0010_0000_0000_0000, approximately 2.2E-308 (2^{-1022})

Biased exponent values 0 and 2047 are reserved for encoding special values of +0, -0, +infinity, -infinity, and NaNs.

Zeros of both positive and negative sign are represented by a biased exponent value *e* of zero and a fraction *f* which is zero.

Infinities of both positive and negative sign are represented by a maximum exponent field value (2047) and a fraction which is zero.

Denormalized numbers of both positive and negative sign are represented by a biased exponent value *e* of 0 and a fraction *f*, which is non-zero. For these numbers, the hidden bit is defined by the IEEE 754 standard to be ‘0’. This number type is not directly supported in hardware. Instead, either a software interrupt handler is invoked, or a default value is defined.

Double-precision not-a-Numbers (NaNs) are represented by a maximum exponent field value (2047) and a fraction *f* which is non-zero.

10.4.2.3 Overflow and Underflow

Defining *pmax* to be the most positive normalized value (farthest from zero), *pmin* the smallest positive normalized value (closest to zero), *nmax* the most negative normalized value (farthest from zero) and *nmin* the smallest normalized negative value (closest to zero), an overflow is said to have occurred if the numerically correct result of an instruction is such that r>*pmax* or r<*nmax*. Additionally, an implementation may also signal overflow by comparing the exponents of the operands. In this case, the hardware examines both exponents ignoring the fractional values. If it is determined that the operation to be performed may overflow (ignoring the fractional values), an overflow may be said to occur. For addition and subtraction this can occur if the larger exponent of both operands is 2046 for double-precision. For multiplication this can occur if the sum of the exponents of the operands less the bias is 2046 for double-precision. Thus:

```
double-precision addition:
    if Aexp >= 2046 | Bexp >= 2046 then overflow
double-precision multiplication:
    if Aexp + Bexp - 1023 >= 2046 then overflow
```

An underflow is said to have occurred if the numerically correct result of an instruction is such that 0<r<*pmin* or *nmin*<r<0. In this case, r may be denormalized, or may be smaller than the smallest denormalized number. As with overflow detection, an implementation may also signal underflow by comparing the exponents of the operands. In this case, the hardware examines both exponents regardless of the fractional values. If it is determined that the operation to be performed may underflow (ignoring the fractional values), an underflow may be said to occur. For division this can occur if the difference of the exponent of the A operand less the exponent of the B operand less the bias is 1. Thus:

```
double-precision multiplication:
    if Aexp - Bexp - 1023 <= 1 then underflow
```

10.4.3 Instruction Descriptions

This section describes double-precision floating-point computational and logical instructions. The following load and store instructions defined by the SPE APU are used to load and store operands:

- **evldd**—Vector Load Double Word into Double Word
- **evlddx**—Vector Load Double Word into Double Word Indexed
- **evstdd**—Vector Store Double Word of Double Word
- **evstddx**—Vector Store Double Word of Double Word
- **evmergehi**—Vector Merge High
- **evmergelo**—Vector Merge Low

These instruction descriptions follow the conventions used in the EREF.

efdabs efdabs

Floating-Point Double-Precision Absolute Value

efdabs **r**D,**r**A

0–5	6–10	11–15	16–20	21–31
0 0 0 1 0 0	rD	rA	0 0 0 0 0	0 1 0 1 1 1 0 0 1 0 0

$rD_{0:63} \leftarrow 0b0 \; || \; rA_{1:63}$

The sign bit of **r**A is cleared and the result is placed into **r**D.

Exception detection for **efdabs** is implementation dependent. On the e500v2, the exception is handled as follows: If **r**A is Infinity, Denorm, or NaN, SPEFSCR[FINV] is set, and FG and FX are cleared. If SPEFSCR[FINVE] = 0, the results are the same as for a normalized number. If SPEFSCR[FINVE] = 1, an interrupt is taken and **r**D is not updated.

efdadd efdadd

Floating-Point Double-Precision Add

efdadd **r**D,**r**A,**r**B

0–5	6–10	11–15	16–20	21–31
0 0 0 1 0 0	rD	rA	rB	0 1 0 1 1 1 0 0 0 0 0

$rD_{0:63} \leftarrow rA_{0:63} +_{dp} rB_{0:63}$

rA is added to **r**B and the result is stored in **r**D. If **r**A is NaN or infinity, the result is either *pmax* (a_{sign}==0), or *nmax* (a_{sign}==1). Otherwise, If **r**B is NaN or infinity, the result is either *pmax* (b_{sign}==0), or *nmax* (b_{sign}==1). Otherwise, if overflow occurs, *pmax* or *nmax* (as appropriate) is stored in **r**D. If underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in **r**D.

Exceptions:

If the contents of **r**A or **r**B are Infinity, Denorm, or NaN, SPEFSCR[FINV] is set. If SPEFSCR[FINVE] is set, an interrupt is taken and **r**D is not updated. Otherwise, if overflow or underflow occurs, SPEFSCR[FOVF] or SPEFSCR[FUNF] is set, and, if the underflow or overflow exception is enabled, an interrupt is taken. If any of these interrupts is taken, **r**D is not updated.

If the result is inexact or if an overflow occurs but overflow exceptions are disabled, and no other interrupt is taken, SPEFSCR[FINXS] is set. If the floating-point inexact exception is enabled, a floating-point round interrupt is taken, **r**D is updated with the truncated result, and FG and FX are updated to allow rounding to be performed in the interrupt handler.

FG and FX are cleared if an overflow, underflow, or invalid operation/input error is signaled, regardless of enabled exceptions.

efdcfs efdcfs

Floating-Point Double-Precision Convert from Single-Precision

efdcfs **r**D,**r**B

0–5	6–10	11–15	16–20	21–31
0 0 0 1 0 0	rD	0 0 0 0 0	rB	0 1 0 1 1 1 0 1 1 1 1

```
FP32format f;
FP64format result;

f ← rB32:63

if (fexp = 0) & (ffrac = 0)) then
    result ← fsign || 630   // signed zero value
else if Isa32NaNorInfinity(f) | Isa32Denorm(f) then
    SPEFSCRFINV ← 1
    result ← fsign || 0b11111111110 || 521   // max value
else if Isa32Denorm(f) then
    SPEFSCRFINV ← 1
    result ← fsign || 630
else
    resultsign ← fsign
    resultexp ← fexp - 127 + 1023
    resultfrac ← ffrac || 290

rD0:63 = result
```

The single-precision floating-point value in the low element of **r**B is converted to a double-precision floating-point value and the result is placed into **r**D. The rounding mode is not used since this conversion is always exact.

Exceptions:

If the low element of **r**B is Infinity, Denorm, or NaN, SPEFSCR[FINV] is set. If SPEFSCR[FINVE] is set, an interrupt is taken, and **r**D is not updated.

FG and FX are always cleared.

efdcfsf efdcfsf

Convert Floating-Point Double-Precision from Signed Fraction

efdcfsf **r**D,**r**B

0–5	6–10	11–15	16–20	21–31
0 0 0 1 0 0	rD	0 0 0 0 0	rB	0 1 0 1 1 1 1 0 0 1 1

$rD_{0:63} \leftarrow$ `CnvtI32ToFP64(rB`$_{32:63}$`, SIGN, F)`

The signed fractional low element in **r**B is converted to a double-precision floating-point value using the current rounding mode and the result is placed into **r**D.

Exceptions: None

efdcfsi efdcfsi

Convert Floating-Point Double-Precision from Signed Integer

efdcfsi **r**D,**r**B

0–5	6–10	11–15	16–20	21–31
0 0 0 1 0 0	rD	0 0 0 0 0	rB	0 1 0 1 1 1 1 0 0 0 1

$rD_{0:63} \leftarrow$ `CnvtSI32ToFP64(rB`$_{32:63}$`, SIGN, I)`

The signed integer low element in **r**B is converted to a double-precision floating-point value using the current rounding mode and the result is placed into **r**D.

Exceptions: None

efdcfuf efdcfuf

Convert Floating-Point Double-Precision from Unsigned Fraction

efdcfuf **r**D,**r**B

0–5	6–10	11–15	16–20	21–31
0 0 0 1 0 0	rD	0 0 0 0 0	rB	0 1 0 1 1 1 1 0 0 1 0

$rD_{0:63} \leftarrow$ `CnvtI32ToFP64(rB`$_{32:63}$`, UNSIGN, F)`

The unsigned fractional low element in **r**B is converted to a double-precision floating-point value using the current rounding mode and the result is placed into **r**D.

Exceptions: None

efdcfui efdcfui

Convert Floating-Point Double-Precision from Unsigned Integer

efdcfui **r**D,**r**B

0–5	6–10	11–15	16–20	21–31
0 0 0 1 0 0	**r**D	0 0 0 0 0	**r**B	0 1 0 1 1 1 1 0 0 0 0

```
rD0:63 ← CnvtSI32ToFP64(rB32:63, UNSIGN, I)
```

The unsigned integer low element in **r**B is converted to a double-precision floating-point value using the current rounding mode and the result is placed into **r**D.

Exceptions: None

efdcmpeq efdcmpeq

Floating-Point Double-Precision Compare Equal

efdcmpeq **crf**D,**r**A,**r**B

0–5	6–8	9–10	11–15	16–20	21–31
0 0 0 1 0 0	**crf**D	0 0	**r**A	**r**B	0 1 0 1 1 1 0 1 1 1 0

```
al ← rA0:63
bl ← rB0:63
if (al = bl) then cl ← 1
else cl ← 0
CR4*crD:4*crD+3 ← undefined || cl || undefined || undefined
```

rA is compared against **r**B. If **r**A is equal to **r**B, the bit in the **crf**D is set, otherwise it is cleared. Comparison ignores the sign of 0 (+0 = -0).

Exceptions:

If the contents of **r**A or **r**B are Infinity, Denorm, or NaN, SPEFSCR[FINV] is set, and the FGH FXH, FG and FX bits are cleared. If floating-point invalid input exceptions are enabled, an interrupt is taken and the condition register is not updated. Otherwise, the comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of ‘*e*’ and ‘*f*’ directly.

efdcmpgt

Floating-Point Double-Precision Compare Greater Than

efdcmpgt **crf**D,**r**A,**r**B

0–5	6–8	9–10	11–15	16–20	21–31
0 0 0 1 0 0	**crf**D	0 0	**r**A	**r**B	0 1 0 1 1 1 0 1 1 0 0

```
al ← rA0:63
bl ← rB0:63
if (al > bl) then cl ← 1
else cl ← 0
CR4*crD:4*crD+3 ← undefined || cl || undefined || undefined
```

rA is compared against **r**B. If **r**A is greater than **r**B, the bit in the **crf**D is set, otherwise it is cleared. Comparison ignores the sign of 0 (+0 = -0).

Exceptions:

If the contents of **r**A or **r**B are Infinity, Denorm, or NaN, SPEFSCR[FINV] is set, and the FGH FXH, FG and FX bits are cleared. If floating-point invalid input exceptions are enabled, an interrupt is taken and the condition register is not updated. Otherwise, the comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of ‘*e*’ and ‘*f*’ directly.

efdcmplt

Floating-Point Double-Precision Compare Less Than

efdcmplt **crf**D,**r**A,**r**B

0–5	6–8	9–10	11–15	16–20	21–31
0 0 0 1 0 0	**crf**D	0 0	**r**A	**r**B	0 1 0 1 1 1 0 1 1 0 1

```
al ← rA0:63
bl ← rB0:63
if (al < bl) then cl ← 1
else cl ← 0
CR4*crD:4*crD+3 ← undefined || cl || undefined || undefined
```

rA is compared against **r**B. If **r**A is less than **r**B, the bit in the **crf**D is set, otherwise it is cleared. Comparison ignores the sign of 0 (+0 = -0).

Exceptions:

If the contents of **r**A or **r**B are Infinity, Denorm, or NaN, SPEFSCR[FINV] is set. and FGH FXH, FG and FX are cleared. If floating-point invalid input exceptions are enabled, an interrupt is taken and the condition register is not updated. Otherwise, the comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of ‘*e*’ and ‘*f*’ directly.

efdctsf efdctsf

Convert Floating-Point Double-Precision to Signed Fraction

efdctsf **r**D,**r**B

0–5	6–10	11–15	16–20	21–31
0 0 0 1 0 0	**r**D	0 0 0 0 0	**r**B	0 1 0 1 1 1 1 0 1 1 1

```
rD32:63 ← CnvtFP64ToI32Sat(rB0:63, SIGN, ROUND, F)
```

The double-precision floating-point value in **r**B is converted to a signed fraction using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit fraction. NaNs are converted as though they were zero.

Exceptions:

If the **r**B contents are Infinity, Denorm, or NaN, or if an overflow occurs, SPEFSCR[FINV] is set and FG and FX are cleared. If SPEFSCR[FINVE] is set, an interrupt is taken and **r**D is not updated.

If conversion is inexact, inexact status is signalled and SPEFSCR[FINXS] is set. If the floating-point inexact exception is enabled, a floating-point round interrupt is taken, **r**D is updated with the truncated result, and FG and FX are updated so the handler can perform rounding.

efdctsi efdctsi

Convert Floating-Point Double-Precision to Signed Integer

efdctsi **r**D,**r**B

0–5	6–10	11–15	16–20	21–31
0 0 0 1 0 0	**r**D	0 0 0 0 0	**r**B	0 1 0 1 1 1 1 0 1 0 1

```
rD32:63 ← CnvtFP64ToI32Sat(rB0:63, SIGN, ROUND, I)
```

The double-precision floating-point value in **r**B is converted to a signed integer using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Exceptions:

If **r**B contents are Infinity, Denorm, or NaN or if an overflow occurs, SPEFSCR[FINV] is set and FG and FX are cleared. If SPEFSCR[FINVE] is set, an interrupt is taken, **r**D is not updated, and no other status bits are set.

If conversion is inexact, inexact status is signalled and SPEFSCR[FINXS] is set. If the floating-point inexact exception is enabled, a floating-point round interrupt is taken, **r**D is updated with the truncated result, and FG and FX are updated so the handler can perform rounding.

efdctsiz efdctsiz

Convert Floating-Point Double-Precision to Signed Integer with Round toward Zero

efdctsiz **r**D,**r**B

0–5	6–10	11–15	16–20	21–31
0 0 0 1 0 0	rD	0 0 0 0 0	rB	0 1 0 1 1 1 1 1 0 1 0

```
rD32:63 ← CnvtFP64ToI32Sat(rB0:63, SIGN, TRUNC, I
```

The double-precision floating-point value in **r**B is converted to a signed integer using the rounding mode Round toward Zero and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Exceptions:

If the contents of **r**B are Infinity, Denorm, or NaN, or if an overflow occurs, SPEFSCR[FINV] is set, and the FG, and FX bits are cleared. If SPEFSCR[FINVE] is set, an interrupt is taken, **r**D is not updated, and no other status bits are set.

If conversion is inexact, inexact status is signalled and SPEFSCR[FINXS] is set. If the floating-point inexact exception is enabled, a floating-point round interrupt is taken, **r**D is updated with the truncated result, and FG and FX are updated so the handler can perform rounding.

efdctuf efdctuf

Convert Floating-Point Double-Precision to Unsigned Fraction

efdctuf **r**D,**r**B

0–5	6–10	11–15	16–20	21–31
0 0 0 1 0 0	rD	0 0 0 0 0	rB	0 1 0 1 1 1 1 0 1 1 0

```
rD32:63 ← CnvtFP64ToI32Sat(rB0:63, UNSIGN, ROUND, F)
```

The double-precision floating-point value in **r**B is converted to an unsigned fraction using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit unsigned fraction. NaNs are converted as though they were zero.

Exceptions:

If the contents of **r**B are Infinity, Denorm, or NaN, or if an overflow occurs, SPEFSCR[FINV] is set, and the FG, and FX bits are cleared. If SPEFSCR[FINVE] is set, an interrupt is taken, and **r**D is not updated.

If conversion is inexact, inexact status is signalled and SPEFSCR[FINXS] is set. If the floating-point inexact exception is enabled, a floating-point round interrupt is taken, **r**D is updated with the truncated result, and FG and FX are updated so the handler can perform rounding.

efdctui efdctui

Convert Floating-Point Double-Precision to Unsigned Integer

efdctui **r**D,**r**B

0–5	6–10	11–15	16–20	21–31
0 0 0 1 0 0	**r**D	0 0 0 0 0	**r**B	0 1 0 1 1 1 1 0 1 0 0

```
rD32:63 ← CnvtFP64ToI32Sat(rB0:63, UNSIGN, ROUND, I
```

The double-precision floating-point value in **r**B is converted to an unsigned integer using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Exceptions:

If **r**B contents are Infinity, Denorm, or NaN or if an overflow occurs, SPEFSCR[FINV] is set, and FG and FX are cleared. If SPEFSCR[FINVE] is set, an interrupt is taken and **r**D is not updated.

If conversion is inexact, inexact status is signalled and SPEFSCR[FINXS] is set. If the floating-point inexact exception is enabled, a floating-point round interrupt is taken, **r**D is updated with the truncated result, and FG and FX are updated so the handler can perform rounding.

efdctuiz efdctuiz

Convert Floating-Point Double-Precision to Unsigned Integer with Round toward Zero

efdctuiz **r**D,**r**B

0–5	6–10	11–15	16–20	21–31
0 0 0 1 0 0	**r**D	0 0 0 0 0	**r**B	0 1 0 1 1 1 1 1 0 0 0

```
rD32:63 ← CnvtFP64ToI32Sat(rB0:63, UNSIGN, TRUNC, I)
```

The double-precision floating-point value in **r**B is converted to an unsigned integer using the rounding mode Round toward Zero and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Exceptions:

If **r**B contents are Infinity, Denorm, or NaN or if an overflow occurs, SPEFSCR[FINV] is set and FG and FX are cleared. If SPEFSCR[FINVE] is set, an interrupt is taken, and **r**D is not updated.

If conversion is inexact, inexact status is signalled and SPEFSCR[FINXS] is set. If the floating-point inexact exception is enabled, a floating-point round interrupt is taken, **r**D is updated with the truncated result, and FG and FX are updated to allow the handler to perform rounding .

efddiv efddiv

Floating-Point Double-Precision Divide

efddiv **r**D,**r**A,**r**B

0–5	6–10	11–15	16–20	21–31
0 0 0 1 0 0	**r**D	**r**A	**r**B	0 1 0 1 1 1 0 1 0 0 0

$rD_{0:63} \leftarrow rA_{0:63} \div_{dp} rB_{0:63}$

rA is divided by **r**B and the result is stored in **r**D. If **r**B is a NaN or infinity, the result is a properly signed zero. Otherwise, if **r**B is a zero (or a denormalized number optionally transformed to zero by the implementation), or if **r**A is either NaN or infinity, the result is either *pmax* ($a_{sign}==b_{sign}$), or *nmax* ($a_{sign}!=b_{sign}$). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in **r**D. If an underflow occurs, +0 or -0 (as appropriate) is stored in **r**D.

Exceptions:

If the contents of **r**A or **r**B are Infinity, Denorm, or NaN, or if both **r**A and **r**B are +/-0, SPEFSCR[FINV] is set. If SPEFSCR[FINVE] is set, an interrupt is taken, and **r**D is not updated. Otherwise, if the content of **r**B is +/-0 and the content of **r**A is a finite normalized non-zero number, SPEFSCR[FDBZ] is set. If floating-point divide by zero Exceptions are enabled, an interrupt is then taken. Otherwise, if an overflow occurs, SPEFSCR[FOVF] is set, or if an underflow occurs, SPEFSCR[FUNF] is set. If either underflow or overflow exceptions are enabled and the corresponding bit is set, an interrupt is taken. If any of these interrupts are taken, **r**D is not updated.

If the result of this instruction is inexact or if an overflow occurs but overflow exceptions are disabled, and no other interrupt is taken, SPEFSCR[FINXS] is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, **r**D is updated with the truncated result, FG and FX are updated to allow rounding to be performed in the interrupt handler.

FG and FX are cleared if an overflow, underflow, divide by zero, or invalid operation/input error is signaled, regardless of enabled exceptions.

efdmul efdmul

Floating-Point Double-Precision Multiply

efdmul **r**D,**r**A,**r**B

0–5	6–10	11–15	16–20	21–31
0 0 0 1 0 0	rD	rA	rB	0 1 0 1 1 1 0 1 0 0 0

$rD_{0:63} \leftarrow rA_{0:63} \times_{dp} rB_{0:63}$

rA is multiplied by **r**B and the result is stored in **r**D. If **r**A or **r**B are zero (or a denormalized number optionally transformed to zero by the implementation), the result is a properly signed zero. Otherwise, if **r**A or **r**B are either NaN or infinity, the result is either *pmax* ($a_{sign}==b_{sign}$), or *nmax* ($a_{sign}!=b_{sign}$). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in **r**D. If an underflow occurs, +0 or -0 (as appropriate) is stored in **r**D.

Exceptions:

If the contents of **r**A or **r**B are Infinity, Denorm, or NaN, SPEFSCR[FINV] is set. If SPEFSCR[FINVE] is set, an interrupt is taken, and **r**D is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF] is set, or if an underflow occurs, SPEFSCR[FUNF] is set. If either underflow or overflow exceptions are enabled and the corresponding bit is set, an interrupt is taken. If any of these interrupts are taken, **r**D is not updated.

If the result of this instruction is inexact or if an overflow occurs but overflow exceptions are disabled, and no other interrupt is taken, SPEFSCR[FINXS] is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, **r**D is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX are cleared if an overflow, underflow, or invalid operation/input error is signaled, regardless of enabled exceptions.

efdnabs efdnabs

Floating-Point Double-Precision Negative Absolute Value

efdnabs **r**D,**r**A

0–5	6–10	11–15	16–20	21–31
0 0 0 1 0 0	**r**D	**r**A	0 0 0 0 0	0 1 0 1 1 1 0 0 1 0 1

$rD_{0:63} \leftarrow 0b1 \ || \ rA_{1:63}$

The sign bit of **r**A is set to 1 and the result is placed into **r**D.

Exception detection for **efdnabs** is implementation dependent. On the e500v2, the exception is handled as follows: If **r**A is Infinity, Denorm, or NaN, SPEFSCR[FINV] is set, and FG and FX are cleared. If SPEFSCR[FINVE] = 0, the results are the same as for a normalized number. If SPEFSCR[FINVE] = 1, an interrupt is taken and **r**D is not updated.

efdneg efdneg

Floating-Point Double-Precision Negate

efdneg **r**D,**r**A

0–5	6–10	11–15	16–20	21–31
0 0 0 1 0 0	**r**D	**r**A	0 0 0 0 0	0 1 0 1 1 1 0 0 1 1 0

$rD_{0:63} \leftarrow \neg rA_0 \ || \ rA_{1:63}$

The sign bit of **r**A is complemented and the result is placed into **r**D.

Exception detection for **efdneg** is implementation dependent. On the e500v2, the exception is handled as follows: If **r**A is Infinity, Denorm, or NaN, SPEFSCR[FINV] is set, and FG and FX are cleared. If SPEFSCR[FINVE] = 0, the results are the same as for a normalized number. If SPEFSCR[FINVE] = 1, an interrupt is taken and **r**D is not updated.

efdsub efdsub

Floating-Point Double-Precision Subtract

efdsub **r**D,**r**A,**r**B

0–5	6–10	11–15	16–20	21–31
0 0 0 1 0 0	**r**D	**r**A	**r**B	0 1 0 1 1 1 0 0 0 0 1

$rD_{0:63} \leftarrow rA_{0:63} -_{dp} rB_{0:63}$

rB is subtracted from rA and the result is stored in **r**D. If **r**A is NaN or infinity, the result is either *pmax* (a_{sign}==0), or *nmax* (a_{sign}==1). Otherwise, If **r**B is NaN or infinity, the result is either *nmax* (b_{sign}==0), or *pmax* (b_{sign}==1). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in **r**D. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in **r**D.

Exceptions:

If the contents of **r**A or **r**B are Infinity, Denorm, or NaN, SPEFSCR[FINV] is set. If SPEFSCR[FINVE] is set, an interrupt is taken, and **r**D is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF] is set, or if an underflow occurs, SPEFSCR[FUNF] is set. If either underflow or overflow exceptions are enabled and the corresponding bit is set, an interrupt is taken. If any of these interrupts are taken, **r**D is not updated.

If the result is inexact or if overflow occurs but overflow exceptions are disabled, and no other interrupt is taken, SPEFSCR[FINXS] is set. If the floating-point inexact exception is enabled, a floating-point round interrupt is taken, **r**D is updated with the truncated result, and FG and FX are updated to allow the interrupt handler to perform rounding.

FG and FX are cleared if an overflow, underflow, or invalid operation/input error is signaled, regardless of enabled exceptions.

efdtsteq efdtsteq

Floating-Point Double-Precision Test Equal

efdtsteq **crf**D,**r**A,**r**B

0–5	6–8	9–10	11–15	16–20	21–31
0 0 0 1 0 0	**crf**D	0 0	**r**A	**r**B	0 1 0 1 1 1 1 1 1 1 0

```
al ← rA0:63
bl ← rB0:63
if (al = bl) then cl ← 1
else cl ← 0
CR4*crD:4*crD+3 ← undefined || cl || undefined || undefined
```

rA is compared against **r**B. If **r**A is equal to **r**B, the bit in the **crf**D is set, otherwise it is cleared. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of '*e*' and '*f*' directly.

No exceptions are generated during the execution of **efdtsteq** If strict IEEE 754 compliance is required, the program should use **efdcmpeq**.

efdtstgt efdtstgt

Floating-Point Double-Precision Test Greater Than

efdtstgt **crf**D,**r**A,**r**B

0–5	6–8	9–10	11–15	16–20	21–31
0 0 0 1 0 0	**crf**D	0 0	**r**A	**r**B	0 1 0 1 1 1 1 1 1 0 0

```
al ← rA0:63
bl ← rB0:63
if (al > bl) then cl ← 1
else cl ← 0
CR4*crD:4*crD+3 ← undefined || cl || undefined || undefined
```

rA is compared against **r**B. If **r**A is greater than **r**B, the bit in the **crf**D is set, otherwise it is cleared. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of '*e*' and '*f*' directly.

No exceptions are generated during the execution of **efdtstgt**. If strict IEEE 754 compliance is required, the program should use **efdcmpgt**.

efdtstlt efdtstlt

Floating-Point Double-Precision Test Less Than

efdtstlt **crf**D,**r**A,**r**B

0–5	6–8	9–10	11–15	16–20	21–31
0 0 0 1 0 0	**crf**D	0 0	**r**A	**r**B	0 1 0 1 1 1 1 1 1 0 1

```
al ← rA0:63
bl ← rB0:63
if (al < bl) then cl ← 1
else cl ← 0
CR4*crD:4*crD+3 ← undefined || cl || undefined || undefined
```

rA is compared against **r**B. If **r**A is less than **r**B, the bit in the **crf**D is set, otherwise it is cleared. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of ‘*e*’ and ‘*f*’ directly.

No exceptions are generated during the execution of **efdtstlt**. If strict IEEE 754 compliance is required, the program should use **efdcmplt**.

efscfd efscfd

Floating-Point Single-Precision Convert from Double-Precision

efscfd **rD,rB**

0–5	6–10	11–15	16–20	21–31
0 0 0 1 0 0	rD	0 0 0 0 0	rB	0 1 0 1 1 0 0 1 1 1 1

```
FP64format f;
FP32format result;
f ← rB_{0:63}
if (f_{exp} = 0) & (f_{frac} = 0)) then
    result ← f_{sign} || ^{31}0   // signed zero value
else if Isa64NaNorInfinity(f) then
    SPEFSCR_{FINV} ← 1
    result ← f_{sign} || 0b11111110 || ^{23}1   // max value
else if Isa64Denorm(f) then
    SPEFSCR_{FINV} ← 1
    result ← f_{sign} || ^{31}0
else
    unbias ← f_{exp} - 1023
    if unbias > 127 then
        result ← f_{sign} || 0b11111110 || ^{23}1    // max value
        SPEFSCR_{FOVF} ← 1
    else if unbias < -126 then
        result ← f_{sign} || 0b00000001 || ^{23}0   // min value
        SPEFSCR_{FUNF} ← 1
    else
        result_{sign} ← f_{sign}
        result_{exp} ← unbias + 127
        result_{frac} ← f_{frac[0:22]}
        guard ← f_{frac[23]}
        sticky ← (f_{frac[24:51]} ≠ 0)
        result ← Round32(result, LOWER, guard, sticky)
        SPEFSCR_{FG} ← guard
        SPEFSCR_{FX} ← sticky
        if guard | sticky then
            SPEFSCR_{FINXS} ← 1
rD_{32:63} ← result
```

The double-precision floating-point value in **r**B is converted to a single-precision floating-point value using the current rounding mode and the result is placed into the low element of **r**D.

Exceptions:

If the **r**B value is Infinity, Denorm, or NaN, SPEFSCR[FINV] is set. If SPEFSCR[FINVE] is set, an interrupt is taken and **r**D is not updated. Otherwise, if overflow occurs, SPEFSCR[FOVF] is set; if underflow occurs, SPEFSCR[FUNF] is set. If underflow or overflow exceptions are enabled and the corresponding bit is set, an interrupt is taken. If an interrupts is taken, **r**D is not updated.

If the result of this instruction is inexact or if an overflow occurs but overflow exceptions are disabled, and no other interrupt is taken, SPEFSCR[FINXS] is set. If the floating-point inexact exception is enabled, a floating-point round interrupt is taken, **r**D is updated with the truncated result, FG and FX are updated so the interrupt handler can perform rounding.

FG and FX are cleared if an overflow, underflow, or invalid operation/input error is signaled, regardless of enabled exceptions.

10.4.4 Embedded Floating-Point Results Summary

Tables in the “Embedded Floating-Point Results” appendix in the EREF summarize the results of various types of floating-point operations on various combinations of input operands. Flag settings are performed on appropriate element flags. Double-precision values appropriate to those tables are as follows:

- *pmax* denotes the maximum normalized positive number. The encoding for double-precision is 0x7FEF_FFFF_FFFF_FFFF.
- *nmax* denotes the maximum normalized negative number. The encoding for double-precision is 0xFFEF_FFFF_FFFF_FFFF.
- *pmin* denotes the minimum normalized positive number. The encoding for double-precision is 0x0010_0000_0000_0000.
- *nmin* denotes the minimum normalized negative number. The encoding for double-precision is0x8010_0000_0000_0000.

10.4.5 Floating-Point Conversion Models

The floating-point to and from non–floating-point conversion model pseudo RTL is provided here as a group of functions that is called from the individual instruction pseudo RTL descriptions.

10.4.5.1 Common Functions

```
// Determine if fp value is a NaN or Infinity
Isa32NaNorInfinity(fp)
return (fp_exp = 255)
Isa32NaN(fp)
return ((fp_exp = 255) & (fp_frac ≠ 0))
Isa32Infinity(fp)
return ((fp_exp = 255) & (fp_frac = 0))
// Determine if fp value is denormalized
Isa32Denorm(fp)
return ((fp_exp = 0) & (fp_frac ≠ 0))
// Determine if fp value is a NaN or Infinity
Isa64NaNorInfinity(fp)
return (fp_exp = 2047)
Isa64NaN(fp)
return ((fp_exp = 2047) & (fp_frac ≠ 0))
Isa64Infinity(fp)
return ((fp_exp = 2047) & (fp_frac = 0))
// Determine if fp value is denormalized
Isa64Denorm(fp)
```

```
return ((fp_exp = 0) & (fp_frac ≠ 0))
```

```
// Signal a Floating Point Error in the SPEFSCR
```

SignalFPError(upper_lower, bits)

```
if (upper_lower = UPPER) then
    bits ← bits << 15
SPEFSCR ← SPEFSCR | bits
bits ← (FG | FX)
if (upper_lower = UPPER) then
    bits ← bits << 15
SPEFSCR ← SPEFSCR & ¬bits
```

```
// Round a result
```

Round32(fp, guard, sticky)

```
FP32format fp;
```

```
if (SPEFSCR_FINXE = 0) then
    if (SPEFSCR_FRMC = 0b00) then    // nearest
        if (guard) then
            if (sticky | fp_frac[22]) then
                v_0:23 ← fp_frac + 1
                if v_0 then
                    if (fp_exp >= 254) then
                        // overflow
                        fp ← fp_sign || 0b11111110 || ^23 1
                    else
                        fp_exp ← fp_exp + 1
                        fp_frac ← v_1:23
                else
                    fp_frac ← v_1:23
    else if ((SPEFSCR_FRMC & 0b10) = 0b10) then    // infinity modes
        // implementation dependent
return fp
```

```
// Round a result
```

Round64(fp, guard, sticky)

```
FP32format fp;
```

```
if (SPEFSCR_FINXE = 0) then
    if (SPEFSCR_FRMC = 0b00) then    // nearest
        if (guard) then
            if (sticky | fp_frac[51]) then
                v_0:52 ← fp_frac + 1
                if v_0 then
                    if (fp_exp >= 2046) then
                        // overflow
                        fp ← fp_sign || 0b11111111110 || ^52 1
                    else
                        fp_exp ← fp_exp + 1
                        fp_frac ← v_1:52
                else
                    fp_frac ← v_1:52
    else if ((SPEFSCR_FRMC & 0b10) = 0b10) then    // infinity modes
        // implementation dependent
return fp
```

10.4.5.2 Convert from Double-Precision Floating-Point to Integer Word with Saturation

```
// Convert 64 bit floating point to integer/fractional
//   signed = SIGN or UNSIGN
//   round = ROUND or TRUNC
//   fractional = F (fractional) or I (integer)
```

CnvtFP64ToI32Sat(fp, signed, round, fractional)

```
FP64format fp;
if (Isa64NaNorInfinity(fp)) then   // SNaN, QNaN, +-INF
    SignalFPError(LOWER, FINV)
    if (Isa64NaN(fp)) then
        return 0x00000000    // all NaNs
    if (signed = SIGN) then
        if (fp_sign = 1) then
            return 0x80000000
        else
            return 0x7fffffff
    else
        if (fp_sign = 1) then
            return 0x00000000
        else
            return 0xffffffff


if (Isa64Denorm(fp)) then
    SignalFPError(LOWER, FINV)
    return 0x00000000    // regardless of sign

if ((signed = UNSIGN) & (fp_sign = 1)) then
    SignalFPError(LOWER, FOVF)  // overflow
    return 0x00000000

if ((fp_exp = 0) & (fp_frac = 0)) then
    return 0x00000000    // all zero values

if (fractional = I) then  // convert to integer
    max_exp ← 1054
    shift ← 1054 - fp_exp
    if (signed ← SIGN) then
        if ((fp_exp ≠ 1054) | (fp_frac ≠ 0) | (fp_sign ≠ 1)) then
            max_exp ← max_exp - 1
else                       // fractional conversion
    max_exp ← 1022
    shift ← 1022 - fp_exp
    if (signed = SIGN) then
        shift ← shift + 1

if (fp_exp > max_exp) then
    SignalFPError(LOWER, FOVF)  // overflow
    if (signed = SIGN) then
        if (fp_sign = 1) then
            return 0x80000000
        else
            return 0x7fffffff
    else
        return 0xffffffff

result ← 0b1 || fp_frac[0:30]  // add U to frac
guard ← fp_frac[31]
sticky ← (fp_frac[32:63] ≠ 0)
for (n ← 0; n < shift; n ← n + 1) do
    sticky ← sticky | guard
    guard ← result & 0x00000001
    result ← result > 1

// Report sticky and guard bits
SPEFSCR_FG ← guard
SPEFSCR_FX ← sticky

if (guard | sticky) then
    SPEFSCR_FINXS ← 1

// Round the result

if ((round = ROUND) & (SPEFSCR_FINXE = 0)) then
    if (SPEFSCR_FRMC = 0b00) then    // nearest
```

```
        if (guard) then
            if (sticky | (result & 0x00000001)) then
                result ← result + 1
    else if ((SPEFSCR_FRMC & 0b10) = 0b10) then    // infinity modes
        // implementation dependent

if (signed = SIGN) then
    if (fp_sign = 1) then
        result ← ¬result + 1

return result
```

10.4.5.3 Convert to Double-Precision Floating-Point from Integer Word with Saturation

```
// Convert from integer/fractional to 64 bit floating point
//   signed = SIGN or UNSIGN
//   fractional = F (fractional) or I (integer)

CnvtI32ToFP64Sat(v, signed, fractional)

FP64format result;

result_sign ← 0
if (v = 0) then
    result ← 0
    SPEFSCR_FG ← 0
    SPEFSCR_FX ← 0
else
    if (signed = SIGN) then
        if (v_0 = 1) then
            v ← ¬v + 1
            result_sign ← 1
    if (fractional = F) then    // fractional bit pos alignment
        maxexp ← 1023
        if (signed = UNSIGN) then
            maxexp ← maxexp - 1
    else
        maxexp ← 1054    // integer bit pos alignment
    sc ← 0
    while (v_0 = 0)
        v ← v << 1
        sc ← sc + 1
    v_0 ← 0    // clear U bit
    result_exp ← maxexp - sc

// Report sticky and guard bits
    SPEFSCR_FG ← 0
    SPEFSCR_FX ← 0

    result_frac ← v_1:31 || ^21 0

return result
```

Chapter 11
L1 Caches

The e500 core complex contains separate 32-Kbyte, eight-way set associative level 1 (L1) instruction and data caches to provide the execution units and registers rapid access to instructions and data.

This chapter describes the organization of the on-chip L1 instruction and data caches, cache coherency protocols, cache control instructions, and various cache operations. It describes the interaction that occurs in the memory subsystem, which consists of the memory management unit (MMU), the caches, the load/store unit (LSU), and the core complex bus (CCB). This chapter also describes the replacement algorithms used for L1 caches.

Note that in this chapter, the term ‘multiprocessor’ is used in the context of maintaining cache coherency. These multiprocessor devices could be actual processors or other devices that can access system memory, maintain their own caches, and function as bus masters requiring cache coherency.

11.1 Overview

The core complex L1 cache implementation has the following characteristics:

- Separate 32-Kbyte 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.
- 2-cycle access time provides 3-cycle read latency for instruction and data caches accesses; pipelined accesses provide single-cycle throughput from caches.
- Instruction and data caches have 32-byte cache blocks. A cache block is the block of memory that a coherency state describes, also referred to as a cache line.
- Four-state modified/exclusive/shared/invalid (MESI) protocol supported for the data cache. See Section 11.3.1, “Data Cache Coherency Model.”
- Both L1 caches support parity generation and checking (enabled through L1CSR0 and L1CSR1 bits), as follows:
  - Instruction cache: 1 parity bit per byte of instruction
  - Data cache: 1 parity bit per byte of data

  See Section 11.2.3, “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 11.2.4, “Cache Parity Error Injection.”
- Each cache can be independently invalidated through cache flash invalidate (CFI) control bits located in L1CSR1 and L1CSR0. See Section 11.4.3, “L1 Instruction and Data Cache Flash Invalidation.”
- Pseudo–least-recently-used (PLRU) replacement algorithm. See Section 11.6.2.1, “PLRU Replacement.”
- Support for individual line locking. See Section 11.4.4, “L1 Instruction and Data Cache Line Locking/Unlocking.”

Bus snooping ensures the coherency of global memory with respect to the data cache.

Both instruction and data cache lines are filled in a single-cycle 32-byte write from line fill buffers as described in Section 11.1.1.1, “Load/Store Unit (LSU),” and Section 11.1.1.2, “Instruction Unit.” Cache line fills write all 32 bytes at once, and therefore do not occur until all four 8-byte data beats have been loaded into the line fill buffer from the CCB.

Both instruction and data accesses are performed critical double word first on the CCB. For data accesses, the LSU receives the critical double word as soon as it is available; it does not wait for all 32 bytes. That data is then forwarded to the requesting unit before being written to the cache, thus minimizing stalls due to cache fill latency. For instruction accesses, instruction fetching cannot resume until the entire cache line is loaded in the instruction line fill buffer (ILFB). Then, the critical double word is written to the cache and instruction fetching can resume.

11.1.1 Block Diagram

The instruction and data caches are integrated with the LSU, the instruction unit, and the core interface unit in the memory subsystem of the core complex as shown in Figure 11-1.

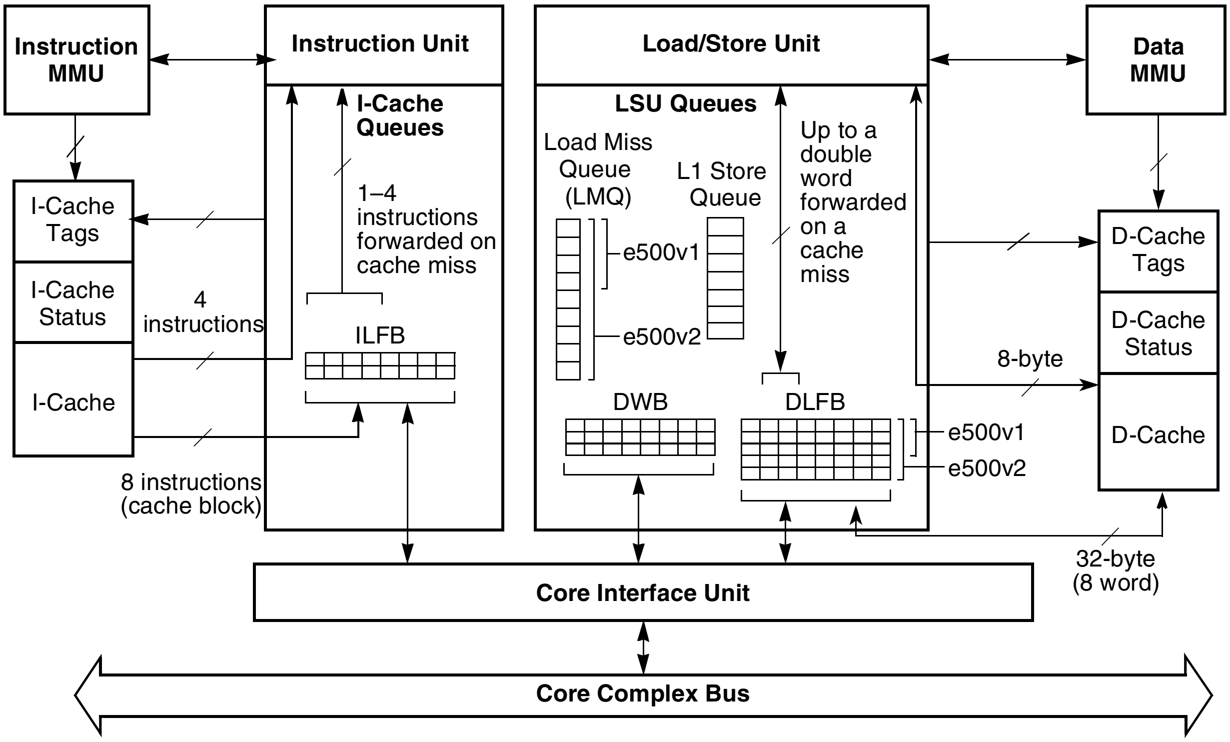


Figure 11-1. Cache/Core Interface Unit Integration

The following sections briefly describe the LSU, the instruction unit, the core interface unit, and the CCB.

11.1.1.1 Load/Store Unit (LSU)

The data cache supplies data to the general-purpose registers (GPRs) by means of the LSU. The core complex LSU is directly coupled to the data cache with a 32-byte interface (the width of a cache block) to allow efficient movement of data to and from the GPRs. The LSU provides all of the logic required to calculate effective addresses, handles data alignment to and from the data cache, provides sequencing for load/store multiple operations, and interfaces with the core interface unit. Write operations to the data cache can be performed on a byte, half-word, word, or double-word basis.

This section describes the LSU queues that support the L1 data cache. See Section 11.3.5, "Load/Store Operations," for more information on architectural coherency implications of load/store operations and the LSU on the core complex. Also, see Section 4.4.4, "Load/Store Execution," for more information on other aspects of the LSU and instruction scheduling considerations.

11.1.1.1.1 Caching-Allowed Loads and the LSU

When free of data dependencies, caching-allowed loads execute in the LSU in a speculative manner with a maximum throughput of one instruction per cycle and a total 3-cycle latency for integer loads. Data returned from the cache on a load is held in a rename buffer until the completion logic commits the value to the processor state.

11.1.1.1.2 Store Queue

Stores cannot be executed speculatively and are held in the seven-entry store queue, shown in Figure 11-1, until the completion logic indicates that the store instruction is to be committed. The store queue arbitrates for access to the L1 data cache. When arbitration is successful, the data is written to the data cache and the store is removed from the store queue. If a store is caching-inhibited, the operation moves through the store queue on to the rest of the memory subsystem.

11.1.1.1.3 L1 Load Miss Queue (LMQ)

As loads reach the LSU, the LSU tries to access the cache. If there is a hit, the cache returns the data. If there is a miss, the LSU allocates an entry in the four-entry load miss queue (LMQ) (nine-entry in the e500v2) and the three-entry data line fill buffer (DLFB) (five-entry in the e500v2); see Section 4.4.2.1, "Load/Store Unit Queueing Structures." The LSU then queues a bus transaction to read the line. If a subsequent load hits, the cache returns the results. If a subsequent load misses, the LSU allocates a second LMQ entry and, if the load is to a different cache line than the outstanding miss, allocates a second DLFB entry and queues a second read transaction on the bus. If the load miss is to the same cache line as the already outstanding miss, the LSU does not allocate a second DLFB entry.

The LSU continues processing load hits and load misses until one of the following conditions occurs:

- A load miss occurs and the LMQ is full.
- The LSU tries to perform a load miss, all DLFB entries are full, and the load is not to any of the cache lines represented in the DLFB.

11.1.1.1.4 Data Line Fill Buffer (DLFB)

The data line fill buffer (DLFB) is located in the LSU; there are three entries in the e500v1 DLFB and five in the e500v2 DLFB. DLFB entries are used for loads and caching-allowed stores. Stores are allocated in the DLFB so that loads can access data from the store immediately (loads cannot access data from the L1 store queue). Also, using DLFB entries for stores, frees up entries in the L1 store queue. Multiple caching-allowed store misses are merged in the DLFB. See Section 11.6.1.4, "Store Miss Merging," for more information.

The e500v2 implements an extra status bit in each LFB entry, indicating whether data in the entry is bad (due to address errors, data bus errors or faults, or data bus parity). Any load that hits in an entry marked bad does not finish. Therefore, completion eventually stalls on the unfinished load until an interrupt occurs. (Under normal operation, this generates an interrupt from the system logic; however, if HID0[RFXE] = 1 (and MSR[ME] = 1), a machine check interrupt is generated.)

11.1.1.1.5 Data Write Buffer (DWB)

When a full line of data is available in the DLFB, the data cache is updated. If a data cache update requires that a line currently in the cache be evicted, that line is cast out and placed in the data write buffer (DWB) until the data has been transferred through the core interface unit to the CCB. If global memory's coherency needs to be maintained, as a result of bus snooping, the L1 cache can also evict a line to the DWB. This write is called a snoop push operation. Note that all cast-out and snoop push writes from the L1 cache are cache-line aligned (critical word is not written first). This is independent of which word in a modified cache line is accessed.

There are three DWB entries: one for snoop pushes, one for castouts, and one that can be used for either.

11.1.1.2 Instruction Unit

The instruction unit interfaces with the L1 instruction cache and the core interface unit. When instructions miss in the instruction cache they are accumulated in the two-entry instruction line fill buffer (ILFB) as they are fetched. After an entire line is available, it is written into the instruction cache and the ILFB is emptied.

The e500v2 implements an extra status bit in each LFB entry, indicating whether data in the entry is bad (due to address errors, data bus errors or faults, or data bus parity). Any load that hits in an entry marked bad does not finish. Therefore, completion eventually stalls on the unfinished load until an interrupt occurs. (Under normal operation, this generates an interrupt from the system logic; however, if HID0[RFXE] = 1 (and MSR[ME] = 1), a machine check interrupt is generated.)

11.1.1.3 Core Interface Unit

The core interface unit handles all bus transactions initiated by the ILFB, DLFB, and DWB. The core interface unit handles all ordering and bus protocol and is the interface between the core complex and the external memory and caches.

The core interface unit performs transactions through the CCB by transferring either the critical double word first (8 bytes) or the critical quad word first (16 bytes). It then forwards the transaction to the instruction or data line fill buffer critical double word first. The CCB also captures snoop addresses for the L1 data cache and the memory reservation (**lwarx** and **stwcx.**) operations.

11.2 L1 Cache Organization

The L1 instruction and data caches of the core complex are both organized as 128 sets of eight blocks with 32 bytes in each cache line. The following subsections describe the differences in the organization of the instruction and data caches.

11.2.1 L1 Data Cache Organization

The L1 data cache is organized as shown in Figure 11-2.

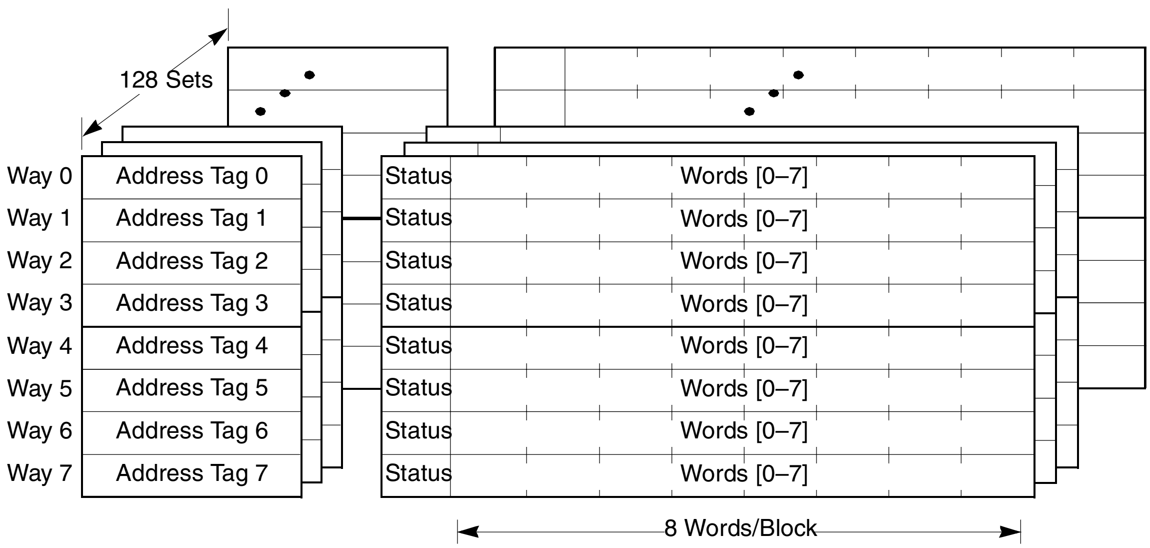


Figure 11-2. L1 Data Cache Organization

Each block consists of 32 bytes of data, 3 status bits, 1 lock bit, and an address tag. For the L1 data cache, a cache block is the 32-byte cache line. Also, although it is not shown in Figure 11-2, the data cache has 1 parity bit/byte (4 parity bits/word).

Each cache block contains 8 contiguous words from memory that are loaded from an 8-word boundary (that is, physical addresses bits 27–31 are zero). Cache blocks are also aligned on page boundaries. Physical address bits PA[20:26] provide the index to select a cache set. The tags consist of physical address bits PA[0:19]. Address translation occurs in parallel with set selection (from PA[20:26]). Lower address bits PA[27:31] locate a byte within the selected block.

The data cache can be accessed internally while a fill for a miss is pending (allowing hits under misses) and the data from a hit can be used as soon as it is available. The LSU forwards the critical word to any pending load misses and allows them to finish. Later, when all the data for the miss has arrived, the entire cache line is reloaded. In addition, subsequent misses can also be sent to the memory subsystem before the original miss is serviced (allowing misses under misses). Up to four misses can be pending in the load miss queue. See Section 4.4.2.1, “Load/Store Unit Queueing Structures,” for more information.

There are status bits associated with each cache block, used to implement the modified/exclusive/shared/invalid (MESI) cache coherency protocol. The coherency protocols are described in Section 11.3, “Cache Coherency Support.”

11.2.2 L1 Instruction Cache Organization

The L1 instruction cache is organized as shown in Figure 11-3.

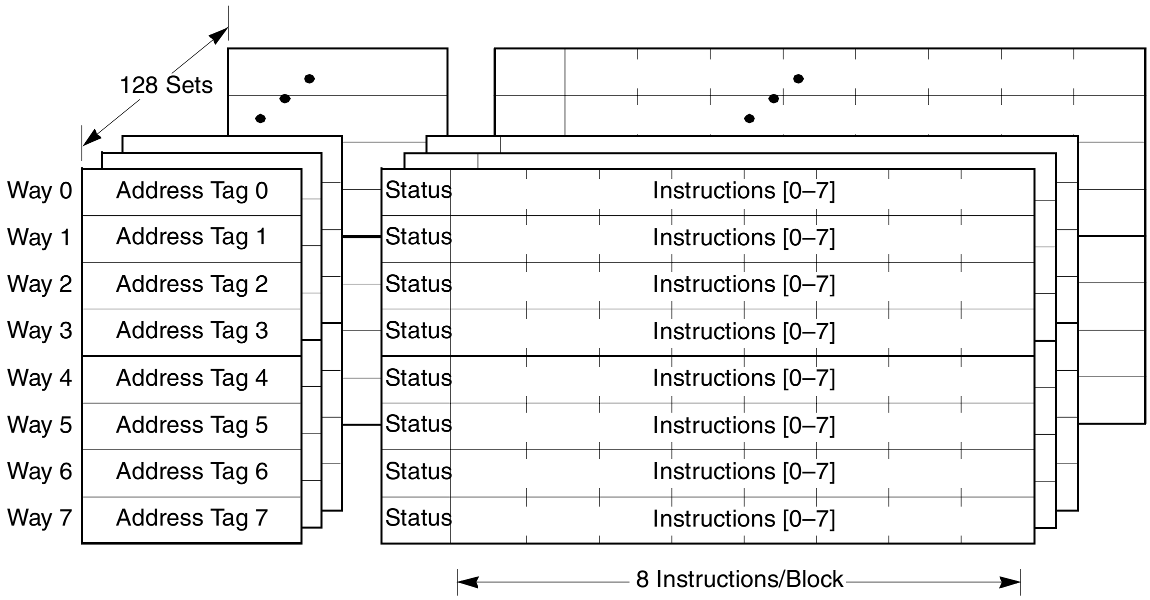


Figure 11-3. L1 Instruction Cache Organization

Each block consists of eight instructions, 1 status bit, 1 lock bit, and an address tag. Also, although it is not shown in Figure 11-3, the instruction cache has 1 parity bit/byte, yielding 32 parity bits for each line.

As with the data cache, each instruction cache block is loaded from an 8-word boundary (that is, bits 27–31 of the physical addresses are zero). Instruction cache blocks are also aligned on page boundaries. Also, PA[20:26] provides the index to select a set, and PA[27:28] selects an instruction within a block. The tags consist of physical address bits PA[0:19]. Address translation occurs in parallel with set selection (from PA[20:26]).

The instruction cache can be accessed internally while a fill for a miss is pending (allowing hits under misses). Although the data cannot be used, the hit information stops a subsequent miss from requesting a fill. In addition, subsequent misses can also be sent to the memory subsystem before the original miss is serviced (allowing misses under misses). When a miss is actually updating the cache, subsequent accesses are blocked for 1 cycle. (But up to four instructions being loaded into the instruction cache can be forwarded to the instruction unit simultaneously.)

The instruction cache differs from the data cache in that it does not implement a multiple-state cache coherency protocol. A single status bit indicates whether a cache block is valid or invalid and there is a single bit for locking.

NOTE

On the e500v1, it is possible for multiple entries in the L1 instruction cache to contain data for the same physical memory location. This error can occur when two different effective addresses (EA) map to the same physical address and accesses to these two EAs occur within the same context and relatively close together in time.

This is avoided by not fetching instructions from one physical address through two or more different EAs within any given context.

11.2.3 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** or **dcba**) modifies the data cache
- A line fill 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
- A line is cast out of the L1 data cache

L1 cache parity checking is disabled by default, and can be enabled by setting L1CSR0[CPE] and L1CSR1[ICPE].

The CCB is also protected by parity. Parity is checked whenever data is read on either of the two CCB read buses; a machine check is generated if errors occur. Additionally, parity is generated whenever data is written on the CCB write bus, giving the SoC platform an opportunity to identify and report errors when data is cast out of the cache or written with a cache-inhibited or write-through store. Parity checking on the CCB read buses is disabled by default and can be enabled by setting HID1[R1DPE] and HID1[R2DPE].

If a cache parity error is detected, a machine check interrupt occurs (as described in Section 5.7.2, “Machine Check Interrupt”).

11.2.4 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[ICPI] is set, any instruction cache line fill has all of its parity bits inverted in the instruction cache.
- If L1CSR0[CPI] is set, any data line fill has all of its 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**, and **dcba**.

NOTE

L1 cache parity checking for the instruction cache must be enabled (L1CSR1[ICPE] = 1) when L1CSR1[ICPI] is set. Similarly for the data cache, L1CSR0[CPE] must be set if L1CSR0[CPI] is set. If the programmer attempts to set L1CSR0[CPI] (using **mtspr**) without setting L1CSR0[CPE], then L1CSR0[CPI] will not be set. If the programmer attempts to set L1CSR1[ICPI] without setting L1CSR1[ICPE], then L1CSR1[ICPI] will not be set.

As described above, if a cache parity error is detected, a machine check interrupt occurs. Sources for cache parity errors are described in Section 5.7.2, “Machine Check Interrupt.”

11.3 Cache Coherency Support

This section describes the L1 cache coherency models and coherency support.

11.3.1 Data Cache Coherency Model

The core complex data cache supports four-state cache coherency protocol for cache lines in the data cache.The four-state protocol (also referred to as MESI protocol) includes the additional shared state. This protocol supports efficient and frequent sharing of data between bus masters.

Each 32-byte data cache block contains status bits that define the MESI state of the cache line. The core complex uses these bits to support coherency protocols and to direct reload operations. Table 11-1 describes data cache states.

Table 11-1. Cache Line State Definitions

Status Bits	Name	Description
101	Modified (M)	The line is in the cache and has been modified with respect to main memory. It does not reside in any other coherent caches.
100	Exclusive (E)	The line is present in the cache, and this cache has exclusive ownership of the line. It is not present in any other coherent cache and it is the same as main memory. This processor may subsequently modify this line without notifying other bus masters.
110	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.
0xx	Invalid (I)	The cache location does not contain valid data.

Every data cache block state is defined by its status bits. Note that in a multiprocessor system, a cache line can exist in the exclusive state in at most one L1 data cache at a time.

Table 11-2 describes how execution of some instructions affects L1 data cache coherency states and WIM bit settings. For more information, see Section 11.3.4, "WIMGE Settings and Effect on L1 Caches."

Table 11-2. L1 Data Cache Coherency State Transitions

Event	WIM	Initial State	Final State
dcba	00x[1]	Any	M
dcbf	xxx	Any	I
dcbi	xxx	Any	I
dcblc (CT = 0)	xxx	Any	same
dcblc (CT = 1)	xxx	Any	same
dcbst	xxx	Any	I
dcbt (CT = 0)	x0x	M, E, or S	same
dcbt (CT = 0)	x0x	I	S or E
dcbt (CT = 1)	x0x	Any	I
dcbtls (CT = 0)	x0x	M, E, or S	same
dcbtls (CT = 0)	x0x	I	S or E
dcbtls (CT = 1)	x0x	Any	I
dcbtst (CT = 0)	00x	M, E, or S	same
dcbtst (CT = 0)	00x	I	E
dcbtst (CT = 1)	00x	Any	I
dcbtstls (CT = 0)	00x	M or E	same
dcbtstls (CT = 0)	00x	S or I	E

Table 11-2. L1 Data Cache Coherency State Transitions (continued)

Event	WIM	Initial State	Final State
dcbtstls (CT = 1)	00x	Any	I
dcbz	00x	Any	M
icblc (CT = 1)	xxx	Any	same
icbt (CT = 1)	x0x	Any	I
icbtls (CT = 1)	x0x	Any	I
Load	xxx	M, E, or S	same
Load	x0x	I	S or E
Load	x1x	Any	same
lwarx	00x	M, E, or S	same
lwarx	00x	I	S or E
lwarx	01x	Any	same
Store	00x	Any	M
Store	10x	M or E	same
Store	10x	S or I	I
Store	01x	Any	same
stwcx	00x	Any	M

[1] The x indicates that the value is either 0 or 1

The core complex provides full hardware support for PowerPC cache coherency and ordering instructions and full hardware implementation of the TLB management instructions.

The core complex broadcasts cache management instructions (**dcbst**, **dcblc** (CT = 1), **icblc** (CT = 1), **dcbf**, **dcbi**, **mbar**, **msync**, **tlbsync**, **icbi**) only if the address broadcast enable bit (HID1[ABE]) is set. On some implementations, ABE must be set to allow management of external L2 caches.

11.3.2 Instruction Cache Coherency Model

The instruction cache supports only invalid and valid state. Table 11-3 describes how execution of instruction cache control instructions affect L1 instruction cache coherency states.

Table 11-3. L1 Instruction Cache Coherency State Transitions

Event	WIM	Initial State	Final State
icbi	xxx	V or I	I
icblc (CT = 0)	xxx	V or I	same
icbtls (CT = 0)	x01	V or I	V

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**) 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 11.5, "L1 Data Cache Flushing." Also, the flushing of self-modifying code from the data cache is described in Section 3.3.1.2.1, "Self-Modifying Code."

11.3.3 Snoop Signaling

Cache coherency is maintained automatically by hardware through snooping the CCB. A bus transaction is enabled for snooping by setting the coherency-required bit (M) in the TLBs (WIMGE = 0bxx1xx). The M bit state is sent with the address on the internal global signal ($\overline{gbl}$). If $\overline{gbl}$ is asserted, the CCB transaction should be snooped by other bus masters.

To determine the action to take due to a snoop, the cache coherency protocol uses transfer type (*ttx*) encodings, which are transmitted on the CCB with the address. See Section 13.2, "Signal Summary." These encodings indicate whether a transaction is a read or write and whether a reading bus master has an intent to modify the cache line. The core complex uses these encodings as a CCB master to signal its intent to other snooping caches.

Clean, flush, and kill are three basic snoops that affect the L1 data cache. Table 11-4 describes the state changes caused by these snoops.

The instruction cache is not snooped, except in the case of the ikill, so coherency must be maintained by software. However, the core complex does support a fast instruction cache invalidation capability as described in Section 11.4.3, "L1 Instruction and Data Cache Flash Invalidation." Also, Section 3.3.1.2.1, "Self-Modifying Code," describes flushing of self-modifying code.

Table 11-4. Data Cache Snoop Coherency State Transitions

Event	Initial State	Final State
clean	M, E, or S	S
clean	I	I
flush	Any	I
kill	Any	I

Table 11-5 describes state changes caused by the ikill snoop.

Table 11-5. Instruction Cache Snoop Coherency State Transitions

Event	Initial State	Final State
ikill	V or I	I

11.3.4 WIMGE Settings and Effect on L1 Caches

All instruction and data accesses are performed under control of the WIMGE bits. This section describes how WIMGE bit settings affect the behavior of the L1 caches. For more information see the EREF.

11.3.4.1 Write-Back Stores

A write-back store that hits a line that is already in exclusive state is immediately stored to the line; the state is changed to modified. If a write-back store hits a line that is already in the modified state, it is immediately stored to the line, and the line stays as modified.

11.3.4.2 Write-Through Stores

A write-through store operation (WIMGE = 0b10xxx) may hit an exclusive cache line. In this case, the store data is written into the data cache and the write-through store goes to the CCB as a single-beat write. The cache line stays exclusive.

A write-through store may also hit in a cache line that is already in the modified state. This situation normally occurs as a result of page table aliasing in which two effective addresses are mapped to the same physical page, but with one mapped as write-through and the other mapped as write-back (that is, non-write-through). In this case, the cache line remains in its current state, the store data is written into the data cache, and the store goes to the CCB as a single-beat write.

11.3.4.3 Caching-Inhibited Loads and Stores

A caching-inhibited load or store (WIMGE = 0bx1xxx) that hits in the cache presents a cache coherency paradox and is normally considered a programming error. If a caching-inhibited load hits in the cache, the cache data is ignored and the load is provided from the CCB 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 the CCB anyway as a single-beat write.

11.3.4.4 Misaligned Accesses and the Endian (E) Bit

Misaligned accesses that cross page boundaries could cause data corruption if the two pages are not set to have the same endianness (that is, one page is big endian while the other is little endian) and the access is allowed. When this situation occurs, the core complex takes a DSI exception and sets the BO (byte ordering) bit in the exception syndrome register (ESR) instead of performing the accesses.

11.3.4.5 Speculative Accesses to Guarded Memory

There is no restriction on how the core complex performs instruction fetching from guarded memory, if the memory area is marked as execute-permitted (UX/SX = 1) in the TLBs. Note that software should mark guarded space as no-execute (UX = 0 and SX = 0) to prevent inadvertent instruction fetching from guarded areas of memory. Then, if the effective address of the current instruction is in guarded, no-execute memory, an execute access control exception occurs, generating an instruction storage interrupt.

The core complex does not perform speculative stores to guarded memory. However, loads from guarded memory may be accessed speculatively if one of the following applies:

- The target location is valid in the data cache.
- The load is guaranteed to be executed. In this case, the entire cache block containing the referenced data may be loaded into the cache.

For more information, see the EREF.

NOTE

On the e500 v1, memory areas must never be set up to be both cacheable and guarded. This is because if the processor detects an error (such as an uncorrectable L2 ECC error) to an area that is both cacheable and guarded, the processor may hang (requiring a hard reset to recover). This is because on the e500v1, if a guarded load encounters a bus error, the transaction never completes and external interrupts cannot be recognized. On the e500v2, external interrupts can be recognized when a guarded load is in progress so the above precautions do not apply.

11.3.5 Load/Store Operations

Load and store operations are assumed to be weakly ordered on the core complex. The LSU can perform load operations that occur later in the program ahead of store operations, even when the data cache is disabled (see Section 11.3.5.2, “Sequential Consistency of Memory Accesses”).

11.3.5.1 Performed Loads and Stores

The architecture defines a performed load operation as one that has the addressed memory location bound to the target register of the load instruction. The architecture defines a performed store operation as one where the stored value is the value that any other processor will receive when executing a load operation (that is, of course, until it is changed again). With respect to the core complex, caching-allowed (WIMGE = 0bx0xxx) loads and caching-allowed, write-back (WIMGE = 0b00xxx) stores are performed when they have arbitrated to address the cache block in the L1 data cache or the CCB and therefore gained coherency ownership of the cache line (that is, they have gained M or E, or S rights to the line). The e500 considers caching-inhibited (WIMGE = 0bx1xxx) loads and stores, and write-through (WIMGE = 0b10xxx) stores performed when they have been successfully presented onto the CCB. Note that loads are considered performed at the L1 data cache only if the respective cache contains a valid copy of that address. Write-back stores are considered performed at the L1 data cache only if the respective cache contains a valid, nonshared copy of that address.

11.3.5.2 Sequential Consistency of Memory Accesses

The architecture requires that all memory operations executed by a single processor be sequentially consistent with respect to that processor as described in the EREF. This means that memory accesses appear to occur in program order with respect to exceptions and data dependencies.

The core complex achieves sequential consistency by operating a single data pipeline to the cache/MMU. Therefore, all memory accesses are presented to the MMU in program order and exceptions are determined in order. Loads are allowed to bypass stores after exception checking has been performed for the store, but data dependency checking is handled in the load/store unit so that a load does not bypass a store with an address match. Newer non-guarded, caching-allowed loads can bypass older non-guarded, caching-allowed loads. Newer non-guarded. caching-allowed write-back stores can bypass older non-guarded, caching-allowed write-back stores if they do not store to overlapping bytes of data.

Note that although memory accesses that miss in the L1 cache are forwarded onto the core interface unit for future arbitration onto the CCB, all potential synchronous exceptions are resolved before the cache access. In addition, although subsequent memory accesses can address the cache, full coherency checking between the cache and the core interface unit is provided to avoid dependency conflicts.

11.3.5.3 Enforcing Store Ordering with Respect to Loads

The e500 core complex guarantees that any load followed by any store is performed in order (with respect to each other). The reverse, however, is not guaranteed. An **mbar** instruction must be inserted between a store followed by a load to ensure sequential ordering between that store and that load.

11.3.5.4 Atomic Memory References

The core complex implements **lwarx** and **stwcx.** as described in Book E and in Section 3.3.1.7, “Atomic Update Primitives Using **lwarx and stwcx.**.” If the EA is not a multiple of 4 for either instruction, an alignment interrupt is invoked. Executing **lwarx** or **stwcx.** to areas marked write-through causes a DSI exception.

As specified in Book E, the core complex requires that, for **stwcx.** to succeed, its EA must be to the same reservation granule as the EA of a preceding **lwarx**. The core complex makes reservations on behalf of aligned 32-byte blocks of the memory address space.

If the reservation has been canceled for any reason, then **stwcx.** fails and clears CR0[EQ]. The architectural intent is to follow the **lwarx**/**stwcx.** instruction pair with a conditional branch that checks whether **stwcx.** failed.

The state of the reservation coherency bit is always signaled. This can be used to determine when an internal condition caused the coherency bit to be reset.

The reservation is invalidated when any asynchronous interrupt is signaled. External interrupts and watchdog timer interrupts are examples of asynchronous interrupts.

11.4 L1 Cache Control

The core complex L1 caches are controlled by programming specific L1CSR*n* bits and by issuing dedicated cache control instructions. Section 11.4.1, "Cache Control Instructions," describes the cache control instructions and gives implementation-specific information. The remainder of this section describes how the cache control instructions and the L1CSR*n* bits are used to control the L1 cache.

11.4.1 Cache Control Instructions

The following instructions can be used for management of the e500 L1 caches—**dcba**, **dcbf**, **dcbi**, **dcblc**, **dcbst**, **dcbt**, **dcbtls**, **dcbtst**, **dcbtstls**, **dcbz**, **icbi**, **icblc**, **icbt**, and **icbtls**.

Table 11-6 shows how cache-control instructions apply to the e500 core, Book E architecture, and the AIM definition of the PowerPC architecture.

Table 11-6. Cache Instruction Comparison

Mnemonic	Instruction	e500 Core	Book E	AIM Architecture
dcba	Data Cache Block Allocate	x	x	x
dcbf	Data Cache Block Flush	x	x	x
dcbi	Data Cache Block Invalidate	x	x	x
dcblc	Data Cache Block Lock Clear	x		
dcbst	Data Cache Block Store	mapped to **dcbf**	x	x
dcbt	Data Cache Block Touch	x	x	x
dcbtls	Data Cache Block Touch and Lock Set	x		
dcbtst	Data Cache Block Touch for Store	x	x	x
dcbtstls	Data Cache Block Touch for Store and Lock Set	x		
dcbz	Data Cache Block Zero	x	x	x
icbi	Instruction Cache Block Invalidate	x	x	x
icblc	Instruction Cache Block Lock Clear	x		

Table 11-6. Cache Instruction Comparison (continued)

Mnemonic	Instruction	e500 Core	Book E	AIM Architecture
icbt	Instruction Cache Touch	no-op	x	
icbtls	Instruction Cache Block Touch and Lock Set	x		

If a cache instruction causes multiple no-op or exception conditions, the results are determined by the order of precedence described in Table 11-7. The priority of the conditions decreases from left to right and the dashes indicate that the operation executes normally. Note that a dash in this table indicates that a failure does not occur under the conditions described.

Table 11-7. Failed Cache Events

Operation	MMU Miss	MSR[PR] = 1 MSR[UCLE] = 0	Protection Violation	CT = CE = 0	CT ≠ 0 or 1	CI	WT
dcbt **dcbtst**	no-op no-op	—[1] —	no-op no-op	— —	no-op no-op	no-op no-op	— no-op
dcbtls **dcbtstls** **dcblc**	DTLB DTLB DTLB	DLK DLK DLK	DSI DSI DSI	CUL CUL no-op	CUL CUL no-op	CUL CUL —	— CUL —
icbtls **icblc**	DTLB DTLB	ILK ILK	DSI DSI	CUL no-op	CUL no-op	CUL —	— —
dcbz [2] **dcba** [2]	DTLB no-op	— —	DSI no-op	——	— —	ALI no-op	ALI no-op
dcbf [2] **dcbi** [2] **icbi** [2]	DTLB DTLB DTLB	— — —	DSI DSI DSI	— — —	— — —	— — —	— — —
lwarx [2] **stwcx.** [2]	DTLB DTLB	— —	DSI DSI	— —	— —	— —	DSI DSI
Load [2] Store [2]	DTLB DTLB	— —	DSI DSI	— —	— —	— —	— —

[1] These instructions are not affected by the value of UCLE

[2] These instructions do not use a CT operand.

Note that CE corresponds to the cache enable bit in L1CSR1 (for the instruction cache) or L1CSR0 (for the data cache). DLK and ILK indicate that the condition causes a data storage interrupt and sets the ESR[DLK] or ESR[ILK]. CUL indicates the unable-to-lock condition that results in a no-op and sets L1CSR1[ICUL] or L1CSR0[CUL].

Acronyms are used to signify the following interrupts:

- DTLB (data TLB interrupt)
- ALI (alignment interrupt)
- DSI (data storage interrupt)

All cache control instructions except **dcba**, **dcbt**, and **dcbtst** generate TLB miss exceptions if the effective address cannot be translated. The **dcba, dcbt,** and **dcbtst** instructions are treated as no-ops if the address cannot be translated.

If a **dcbt** or **dcbtst** instruction accesses a page marked caching-inhibited, it is treated as a no-op. The **icbt** instruction is treated as a no-op when the CT operand is equal to zero. The **dcbst** instruction maps to **dcbf**.

The core complex broadcasts the cache control instructions according to the value of HID1[ABE]. If ABE is cleared, most cache control instructions are not broadcast. If it is set, cache control instructions are broadcast.

11.4.2 L1 Instruction and Data Cache Enabling/Disabling

The instruction and data caches are enabled and disabled with the cache enable (CE) bits in L1CSR1 and L1CSR0, 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. The default power-up state of L1CSR0[CE] and L1CSR1[ICE] is zero (caches disabled).

When the data cache is disabled, snooping of lines in the cache is not performed. Before the data cache is disabled it must be invalidated to prevent coherency problems when it is enabled again.

All cache operations are affected by disabling the cache. Touch instructions (**dcbt**, **dcbtst**, **dcblc**, **dcbtls**, **dcbtstls**, **icblc**, and **icbtls**) performed on the CCB by the e500 do not affect the cache when it is disabled. A **dcba** or **dcbz** instruction to a disabled data cache zeros the cache line in memory, but does not affect the cache when it is disabled.

If CE = 0, the **dcbi** and **dcbf** instructions do not affect the L1 data cache.

The setting of L1CSR0[CE] must be preceded by an **msync** and **isync** instruction, to prevent a cache from being disabled or enabled in the middle of a data or instruction access. See Table 2-42 for more information on synchronization requirements.

11.4.3 L1 Instruction and Data Cache Flash Invalidation

The data cache can be invalidated by executing a series of **dcbi** instructions or by setting L1CSR0[CFI].

If software can guarantee that data is not modified, the cache can be invalidated without updating system memory; if a modified line is invalidated, the data is lost. To prevent the loss of data, modified cache lines must be flushed, as described in Section 11.5, “L1 Data Cache Flushing.”

Because the instruction cache never contains modified data, there is no need to flush the instruction cache before it is invalidated.

The instruction cache can be invalidated by setting L1CSR1[ICFI]. The L1 caches can be flash invalidated independently. The setting of L1CSR0[CFI] and L1CSR1[ICFI] must be preceded by an **msync** and **isync**, respectively.

Both caches are invalidated automatically at power-up. Because a subsequent reset does not invalidate caches automatically, software must set the CFI bits if invalidation is desired after a warm reset. This causes a flash invalidation performed in a single CPU cycle, after which the CFI bits are cleared automatically (CFI bits are not sticky). Note that flash invalidate operations are not broadcast on the CCB.

Note that when an L2 tag parity error occurs on an attempt to write a new line, the L2 cache must be flash invalidated. Performing a **dcbi** does not invalidate the line because it, like the write, is treated as a cache miss, so the status of that line is not changed. L2 functionality is not guaranteed if flash invalidation is not performed after a tag parity error.

Individual instruction or data cache blocks can be invalidated using **icbi** and **dcbi**, respectively. Note that invalidating the caches resets all cache status bits, including lock bits. Also note that with **dcbi**, the e500 core invalidates the cache block without pushing it out to memory. See Section 3.3.1.8.1, "User-Level Cache Instructions."

Exceptions and other events that can access the L1 cache should be disabled during this time so that the PLRU algorithm can function undisturbed.

11.4.4 L1 Instruction and Data Cache Line Locking/Unlocking

User-mode instructions perform cache line locking/unlocking based on the complete address of the cache line. **dcblc**, **dcbtls**, and **dcbtstls** are for data cache locking and unlocking and **icblc** and **icbtls** are for instruction cache locking. For descriptions, see Section 3.8.4, "Cache Locking APU."

The CT operand is used to indicate the cache target of the cache line locking instruction.

Lock instructions are treated as loads when translated by the data TLB, and they cause exceptions when data TLB errors or data storage interrupts occur.

The user-mode cache lock enable bit, MSR[UCLE], is used to restrict user-mode cache line locking by the operating system. If MSR[UCLE] = 0, any cache lock instruction executed in user mode (MSR[PR] = 1) causes a cache-locking DSI exception and sets either ESR[DLK] or ESR[ILK]. This allows the OS to manage and track the locking/unlocking of cache lines by user-mode tasks. If MSR[UCLE] is set, the cache-locking instructions can be executed in user mode and do not cause a DSI for cache locking. However, they may still cause a DSI for access violations.

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.

The following cases cause an attempted lock to fail:

- The target address is marked caching-inhibited.
- The corresponding cache is disabled and the CT operand of the cache locking instruction = 0.
- The cache target operand (CT[6–10]) is greater than 1.
- **dcbtstls** is used for a target address of a write-through page.

In these cases, the lock set instruction is treated as a no-op and the data cache unable-to-lock bit (L1CSR0[CUL]) or the instruction cache unable-to-lock bit (L1CSR1[ICUL]) is set. This condition does not cause an exception.

It is acceptable to lock all ways of a cache set. A non-locking line fill for a new address in a completely locked cache set will not be put into the cache. It is, however, loaded into a DWB and creates the appropriate normal burst write transfer.

The cache-locking DSI handler must decide whether to lock a given cache line based on available cache resources.

If the locking instruction is a set lock instruction, to lock the line, the handler should do the following:

1. Add the line address to its list of locked lines.
2. Execute the appropriate set lock instruction to lock the cache line.
3. Modify save/restore register 0 (SRR0) to point to the instruction immediately after the locking instruction that caused the DSI.
4. Execute an **rfi**.

If the locking instruction is a clear lock instruction, to unlock the line, the handler should do the following:

1. Remove the line address from its list of locked lines.
2. Execute the appropriate clear lock instruction to unlock the cache line.
3. Modify SRR0 to point to the instruction immediately after the locking instruction that caused the DSI.
4. Execute an **rfi**.

Failure to update SRR0 to point to the instruction after the locking/unlocking instruction causes the exception handler to be repeatedly invoked for the same instruction.

11.4.4.1 Effects of Other Cache Instructions on Locked Lines

The following cache instructions do not affect the state of a cache line's lock bit:

- **dcbt** (CT = 0)
- **dcbtst** (CT = 0)

If **dcbt** is performed to a line that is locked in the cache in the modified or exclusive state, **dcbt** takes no action. However, if the line is invalid, and therefore not locked, **dcbt** causes a state change.

If a **dcbtst** (CT=0) is performed to a line that is locked in the cache in the modified or exclusive state, **dcbtst** takes no action. If the line is invalid, and therefore not locked, **dcbtst** causes a state change.

The following cache instructions are treated as stores and may cause the invalidation and unlocking of a cache line in another processor in a multiprocessor system:

- **dcba**
- **dcbz**

In implementations with an L2 cache, the following instructions, when directed to the L2 cache (CT = 1), flush/invalidate and unlock a line in the L1 data cache of the current processor:

- **dcbt**
- **dcbtst**
- **dcbtls**
- **dcbtstls**
- **icbt**
- **icbtls**

The following cache instructions flush/invalidate and unlock a line in the cache of the current processor, and may also flush/invalidate and unlock a cache line in other processors in a multiprocessor system:

- **dcbf**
- **dcbst**
- **icbi**
- **dcbi**

11.4.4.2 Flash Clearing of Lock Bits

The core complex 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 invalidation performed in a single CPU cycle, after which the CLFC bits are automatically cleared (CLFC bits are not sticky).

11.5 L1 Data Cache Flushing

Any modified entries in the data cache can be copied back to memory (flushed) by using a **dcbf** instruction or by executing a series of 12 uniquely addressed load or **dcbz** instructions to each of the 128 sets. The address space should not be shared with any other process to prevent snoop hit invalidations during the flushing routine. Exceptions should be disabled during this time so that the PLRU algorithm is not disturbed.

The following methods can be used to flush a region in the L1 cache:

- Perform reads to any 48-Kbyte region, then execute **dcbf** instructions to that region. Note that a 48-Kbyte region must be used to ensure that the PLRU algorithm flushes all of the cache entries (12 x 128 sets x 32 bits = 48 Kbytes).
- Perform reads from any 48-Kbyte region that is guaranteed to not be modified in the L1 cache (for example, a ROM region).
- Execute **dcbz** instructions to any 48-Kbyte scratch section, then invalidate the cache. Note that it is necessary to use a scratch region because some zeroed lines will be cast out.

For each of these methods, the following is necessary:

- Interrupts must be disabled.
- The 48-Kbyte region chosen is not being used by the system—that is, that snoops do not occur to this region.

On the e500v2 the HID0 register contains a field, DCFA (data cache flush assist), that, when set, forces the data cache to ignore invalid sets on miss replacement selection and follow the replacement sequence defined by the PLRU bits. This reduces the series of uniquely addressed load or **dcbz** instructions to eight per set. The bit should be set just before beginning a cache flush routine and should be cleared when the series of instructions is complete.

11.6 L1 Cache Operation

This section describes operations performed by the L1 instruction and data caches.

11.6.1 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.

11.6.1.1 Data Cache Fills

The core complex data cache blocks are filled (sometimes referred to as a cache reload) from an L2 cache or the memory subsystem when cache misses occur for caching-allowed accesses, as described in Section 11.1.1.1, "Load/Store Unit (LSU)," and Section 11.1.1.2, "Instruction Unit."

When the data cache is disabled (L1CSR0[CE] = 0), data accesses bypass the data cache, are forwarded to the memory subsystem as caching-allowed, and proceed to the CCB. Returned data is forwarded to the requesting execution unit, but is not loaded into any of the caches.

Each of the eight ways of each set in the data cache can be locked (by locking all of the cache lines in the way with the **dcbtls** or **dcbtstls** instruction). When at least one way is unlocked, misses are treated normally and are allocated to one of the unlocked ways on a reload. If all eight ways are locked, store/load misses proceed to the memory subsystem as normal caching-allowed accesses. In this case, the data is forwarded to the requesting execution unit when it returns, but it is not loaded into the data cache. If the data is modified, it is loaded into a DWB and creates the appropriate normal burst write transfer.

Each of the eight ways of each set in the instruction cache can be locked (by locking all of the cache lines in the way with the **icbtls** instruction). When at least one way is unlocked, misses are treated normally and they are allocated to one of the unlocked ways on a reload. If all of the ways are locked, instruction misses proceed to the memory subsystem as normal caching-allowed accesses. In this case, the instruction is forwarded to the instruction unit when it returns, but it is not loaded into the instruction cache.

Note that caching-inhibited stores should not access any of the caches (see Section 11.3.4.3, "Caching-Inhibited Loads and Stores," for more information). See Section 11.6.1.4, "Store Miss Merging," for more information on the handling of caching-allowed store misses.

11.6.1.2 Instruction Cache Fills

The instruction cache provides a 128-bit interface to the instruction unit, so as many as four instructions can be made available to the instruction unit in a single clock cycle on an L1 instruction cache hit. On a miss, the core complex instruction cache blocks are loaded in one 32-byte beat from the CCB; the instruction cache is nonblocking, 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. These accesses are forwarded to the memory subsystem as caching-allowed and proceed to the CCB. When the instructions are returned, they are forwarded to the instruction unit but are not loaded into the instruction cache.

The instruction unit fetches a total of four instructions at a time directly from the memory subsystem for caching-inhibited instruction fetches. Similar to the data cache, when the instructions are returned, they are forwarded to the instruction unit but are not loaded into any of the caches in this case.

11.6.1.3 Cache Allocation on Misses

Instruction cache misses cause a new line to be allocated into the instruction cache on a PLRU basis, provided the cache is not completely locked or disabled.

If there is a data cache miss for a caching-allowed load or store (including touch instructions) and the line is not already going to be allocated into the data cache as a result of a previous load/store miss, the miss causes a new line to be allocated into the data cache on a PLRU basis, provided the cache is not completely locked or disabled. A store that is write-through or caching-inhibited that misses in the data cache does not cause a fill. Also, cache operations such as **dcbi** and **dcbf** that miss in the cache do not cause a fill.

11.6.1.4 Store Miss Merging

When a caching-allowed store misses in the data cache, an entry is allocated in the DLFB. The store data is written into the DLFB. The remainder of the bytes not written by the store data are filled in when the cache block is eventually fetched from memory through the CCB. When all 32 bytes are valid, the cache block is reloaded into the data cache.

If a subsequent store miss hits on a DLFB entry for a previous store miss, the subsequent store miss also writes its data into the DLFB for that entry. Any number of stores that hit the DLFB entry created by the original store miss can be written in to the DLFB before it reloads the data into the data cache. This behavior is known as store miss merging

11.6.1.5 Store Hit to a Data Cache Block Marked Shared

When a write-back store hits in the L1 data cache and the block is in the shared state, the target block is invalidated in the data cache. The store is then treated as a miss.

11.6.1.6 Data Cache Block Push Operation

When an L1 cache block in the core complex is snooped (by another bus master) and the data hits and is modified, the cache block must be written to memory and made available to the snooping device. The push operation propagates to the DWB and then to the CCB.

11.6.2 L1 Cache Block Replacement

When a new block needs to be placed in the instruction or data cache, the pseudo–least-recently-used (PLRU) replacement algorithm is used. Note that data cache replacement selection is performed at reload time and not when the miss occurs. Instruction cache replacement selection occurs when an instruction cache miss is first recognized.

When a cache line is accessed, it is tagged as the most-recently-used line of the set. When a miss occurs, if all lines in the set are valid (occupied), the least-recently-used line is replaced with the new data. The PLRU bits in the cache are updated each time a cache hit occurs based on the most-recently-used cache line.

Modified data to be replaced is written into a DWB and eventually is written back to main memory.

Data load or write-back store accesses that miss in the L1 data cache function similarly to L1 instruction cache misses. They cause a new line to be allocated on a PLRU basis, provided the cache is not completely locked or disabled.

Note that modified data in the replacement line of any cache can cause a castout to occur to the CCB. In all such cases, the castout is not initiated until new data is ready to be loaded.

11.6.2.1 PLRU Replacement

Block replacement is performed using a binary decision tree, PLRU algorithm. There is an identifying bit for each cache way, L[0–7]. There are seven PLRU bits, B[0–6] for each set in the cache to determine the line to be cast out (replacement victim). The PLRU bits are updated when a new line is allocated or replaced and when there is a hit in the set.

This algorithm prioritizes the replacement of invalid entries over valid ones (starting with way 0). Otherwise, if all ways are valid, one is selected for replacement according to the PLRU bit encodings shown in Table 11-8.

Table 11-8. L1 PLRU Replacement Way Selection

PLRU Bits						Way Selected for Replacement
B0	0	**B1**	0	**B3**	0	L0
	0		0		1	L1
	0		1	**B4**	0	L2
	0		1		1	L3
	1	**B2**	0	**B5**	0	L4
	1		0		1	L5
	1		1	**B6**	0	L6
	1		1		1	L7

Figure 11-4 shows the decision tree used to generate the victim line in the PLRU algorithm.

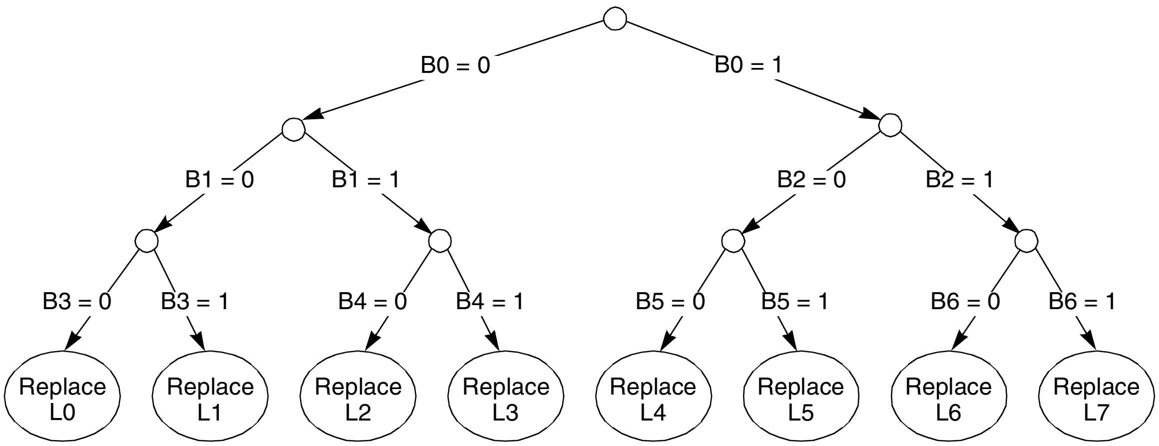


Figure 11-4. PLRU Replacement Algorithm

During power-up or hard reset, the valid bits of the L1 caches are automatically cleared to point to way L0 of each set.

11.6.2.2 PLRU Bit Updates

Except for snoop accesses, each time a cache block is accessed, it is tagged as the most-recently-used way of the set. For every hit in the cache or when a new block is reloaded, the PLRU bits for the set are updated using the rules specified in Table 11-9.

Table 11-9. 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.

11.6.2.3 Cache Locking and PLRU

The core complex does not replace locked lines. Each L1 cache line has a lock bit, which can be set through the cache locking instructions and cleared through the cache unlocking instructions or line invalidation. Lock bits are used at reload time to steer the PLRU algorithm away from selecting locked cache lines.

11.7 L2 Cache Support

This section describes interactions between the e500 core and an L2 cache implementation.

11.7.1 Invalidating the L2 Cache after a Cache Tag Parity Error

If an L2 cache tag parity error occurs on an attempt to write a new line, the L2 cache must be flash invalidated. Performing a **dcbi** does not invalidate the line because it, like the write, is treated as a cache miss, so the status of that line is not changed. L2 functionality is not guaranteed if flash invalidation is not performed after a tag parity error.

See Section 11.4.3, “L1 Instruction and Data Cache Flash Invalidation.”

11.7.2 L2 Locking

The core complex implements specific instructions to selectively lock and unlock lines in its L1 caches or in an L2 cache. To facilitate locking and unlocking of an L2 cache (usually located directly on the CCB), the core complex provides an address lock attribute (CL) on the bus, which can be used in conjunction with the transfer type, *ttx*, encodings to identify which addresses to lock or unlock.

When the core complex executes an instruction to lock a line in an L2 cache (**dcbtls**, **dcbtstls**, or **icbtls**, with CT = 1), it normally performs the associated bus operation as a burst read transaction with a reading-type *ttx* code (READ, RWITM, or RCLAIM) and with the lock attribute asserted. An L2 cache may recognize this transaction as a direction to establish the cache line (if not already valid) and to mark it as locked. Note that this is a complete address/data transaction by the core complex to memory that requires read data to be returned to the core complex. The read data, however, is not used or cached internally by the core complex. The purpose for the bus transaction is to establish a locked line in the L2 cache and to make data available from system memory for the L2 cache to capture.

If a cache locking instruction targeted at an L2 cache also hits to a line modified in the L1 data cache, the core complex pushes the line from the L1 data cache as a non-global burst write operation (similar to a regular L1 castout) with the lock attribute set and the write-through attribute negated, rather than performing a read bus operation as described above. An L2 cache may also recognize this transaction as a direction to establish and capture the cache line and mark it as locked.

11.7.2.1 L2 Unlocking

When the core complex executes an instruction (**dcblc**, **icblc**) to unlock an L2 cache line, it performs the associated bus operation as an address-only transaction with a *ttx* encoding of CLEAN and with the lock attribute asserted. An L2 cache may recognize this transaction as a direction to unlock the specified address from its cache. This transaction always is performed as non-global because it is specifically targeted at an L2 cache.

An L2 cache may also use other bus transactions to cause locks to be cleared, such as bus transactions as a result of **dcbf** (identified on the bus as an address-only FLUSH, or as an L1 push due to **dcbf**).

11.7.2.2 L1 Overlock

A program may attempt to establish a ninth locked entry at a cache index that already has all eight of its ways locked. In this overlock case, the core complex performs a reading transaction on the bus to initially bring in the ninth (newest) line and then immediately push that line out to bus as a nonglobal burst write with the lock attribute asserted, rather than attempt to allocate that line in the L1 data cache. This write operation looks identical on the bus to the hit-to-modified case described in Section 11.7.2, “L2 Locking.”

Chapter 12
Memory Management Units

This chapter describes the implementation details of the e500v1 core complex MMU relative to the Book E architecture and the Freescale Book E standards. In addition, it describes the e500v2 core with its extended page sizes and extended physical addressing. All text denoted as e500 applies to both the e500v1 and the e500v2, unless specifically noted as applying to only one core or the other. For background on the MMU definition in Book E and the Freescale Book E standards, see the *EREF: A Reference for Freescale Book E and the e500 Core* (EREF).

12.1 e500 MMU Overview

The e500 core complex employs a two-level memory management unit (MMU) architecture. There are 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. The programming model for implementing translation lookaside buffers (TLBs) provided in Book E and the Freescale Book E standard applies to the L2 MMU of the core complex.

12.1.1 MMU Features

The e500 core has the following features:

- 32-bit effective address translated to 32-bit real address (using a 41-bit interim virtual address) for the e500v1 core and 36-bit real address for the e500v2 core
- Two-level MMU containing a total of six TLBs for maximizing TLB hit rates
- Three 8-bit PID registers (PID0–PID2) for supporting up to 255 translation IDs at any time in the TLB, with three concurrent translation IDs as potential matches for each access
- TLB entries for variable-sized (4-Kbyte–256-Mbyte pages for the e500v1 and 4-Kbyte–4-Gbyte pages for the e500v2) and fixed-size (4-Kbyte) pages
- No page table format is defined; software is free to use its own page table format.
- TLBs maintained by system software through the TLB instructions and six (e500v1) or seven (e500v2) MAS registers

The level 1 MMUs have the following features:

- Two 4-entry, fully-associative TLB arrays (one for instruction accesses and one for data accesses) supporting the nine (e500v1) or eleven (e500v2) page sizes shown in Table 12-2
- Two 64-entry, 4-way set-associative TLB arrays (one for instruction accesses and one for data accesses) that support only 4-Kbyte pages
- L1 MMU access occurs in parallel with L1 cache access time (address translation/L1 cache access can be fully pipelined so one load/store can be completed on every clock).
- Performs an L1 TLB lookup for an instruction access in parallel with an L1 TLB lookup for a data access
- All L1 TLB entries are a proper subset of TLB entries resident in L2 MMU (completely maintained by the hardware).
- Automatically performs invalidations to maintain consistency with L2 TLBs

The level 2 MMU has the following features:

- A 16-entry, fully-associative unified (for instruction and data accesses) L2 TLB array (TLB1) supports the nine (e500v1) or eleven (e500v2) page sizes shown in Table 12-2.
- A 256-entry, 2-way (e500v1) or 512-entry, 4-way (e500v2) set-associative unified (for instruction and data accesses) L2 TLB array (TLB0) supports only 4-Kbyte pages.
- Hardware assistance for TLB miss exceptions
- TLB1 and TLB0 managed by **tlbre**, **tlbwe**, **tlbsx**, **tlbsync**, **tlbivax**, and **mtspr** instructions
- Performs invalidations in TLB1 and TLB0 caused by **tlbivax** instructions executed by this core. Also supports snooping of TLB1 and TLB0 for invalidation caused by **tlbivax** instructions executed by other masters.
- IPROT bit implemented in TLB1 prevents invalidations, protecting critical entries (so designated by having the IPROT bit set) from being invalidated.

12.1.2 TLB Entry Maintenance Features

The TLB entries of the e500 core complex must be loaded and maintained by the system software; this includes performing any required table search operations in memory. The e500 provides support for maintaining TLB entries in software with the resources shown in Table 12-1. Note that many of these features are defined at the Freescale Book E level.

Table 12-1. TLB Maintenance Programming Model

Features		Description	More Information Section/Page
TLB Instructions	**tlbre**	TLB Read Entry instruction	12.4.1/12-18
	tlbwe	TLB Write Entry instruction	12.4.2/12-19
	tlbsx r**A**, r**B** (preferred form is **tlbsx 0, r**B)	TLB Search for entry instruction	12.4.3/12-19
	tlbivax r**A**, r**B**	TLB Invalidate entries instruction	12.4.4/12-20
	tlbsync	TLB Synchronize invalidations with other masters' instruction	12.4.5/12-22
Registers	PID0–PID2	Process ID registers	See Table 12-7 for more comprehensive cross references
	MMUCSR0	MMU control and status register	
	MMUCFG	MMU configuration register	
	TLB0CFG–TLB1CFG	TLB configuration registers	
	MAS0–MAS4, MAS6; e500v2 also implements MAS7	MMU assist registers. Note that MAS5 is not implemented on the e500.	
	DEAR	Data exception address register	
Interrupts	Instruction TLB miss exception	Causes instruction TLB error interrupt	12.5.1/12-23
	Data TLB miss exception	Causes data TLB error interrupt	
	Instruction permissions violation exception	Causes ISI interrupt	12.5.2.1/12-24
	Data permissions violation exception	Causes DSI interrupt	

Other hardware assistance features for maintenance of the TLBs on the e500 are described in Section 12.5, "TLB Entry Maintenance—Details."

12.2 Effective-to-Real Address Translation

The core complex fetch and load/store units generate 32-bit effective addresses. The MMU translates each of these addresses to 32-bit real addresses, (36 bits for the e500v2) which are then used for memory bus accesses. Figure 12-1 illustrates the high-level translation flow with 32-bit real addressing for the e500v1core, showing that because the smallest page size supported by the e500 core complex is 4 Kbytes, the least-significant 12 bits always index within the page and are untranslated. The appropriate L1 MMU (instruction or data) is checked for a matching address translation first. If it misses, the request for translation is forwarded to the unified (instruction and data) L2 MMU.

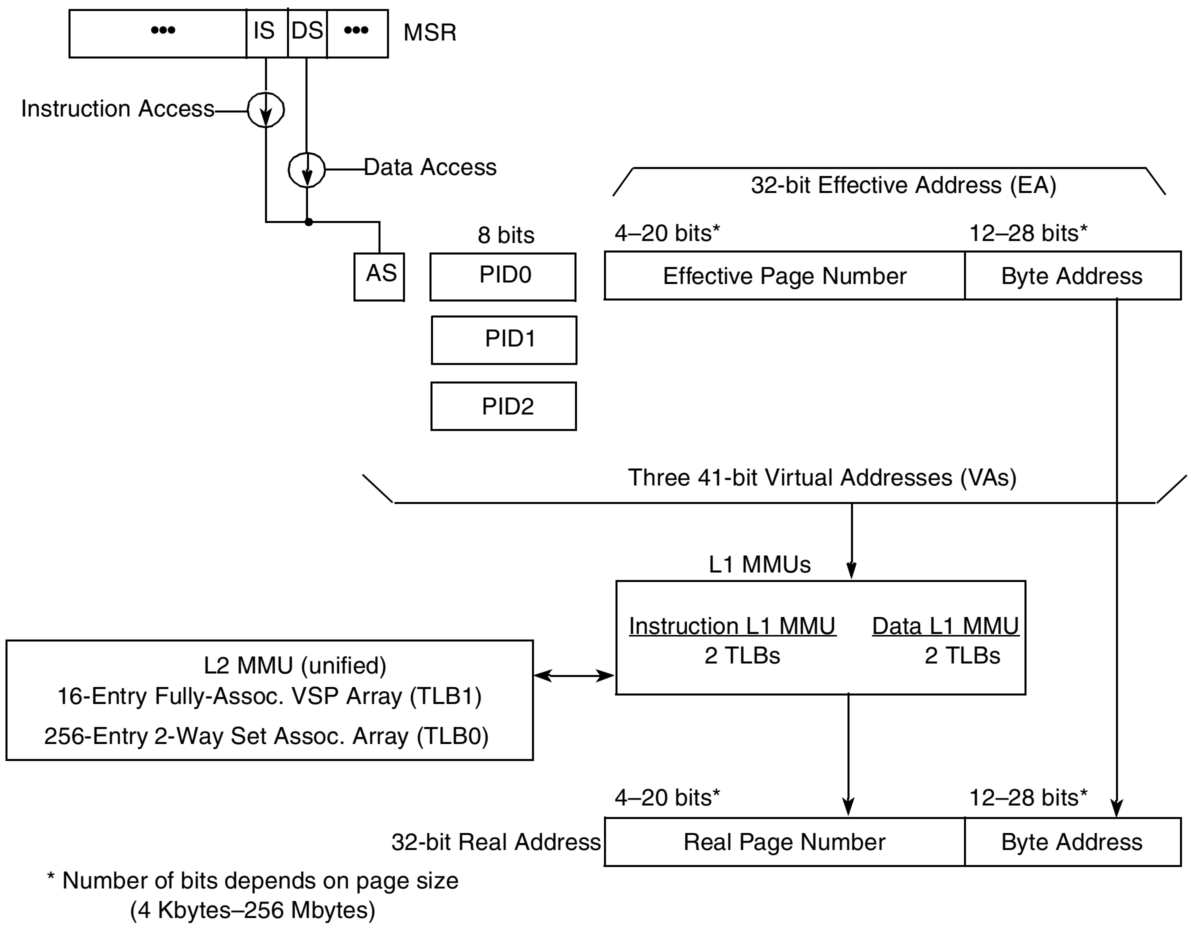


Figure 12-1. Effective-to-Real Address Translation Flow (e500v1)

Figure 12-2 shows the same translation flow for the e500v2 core.

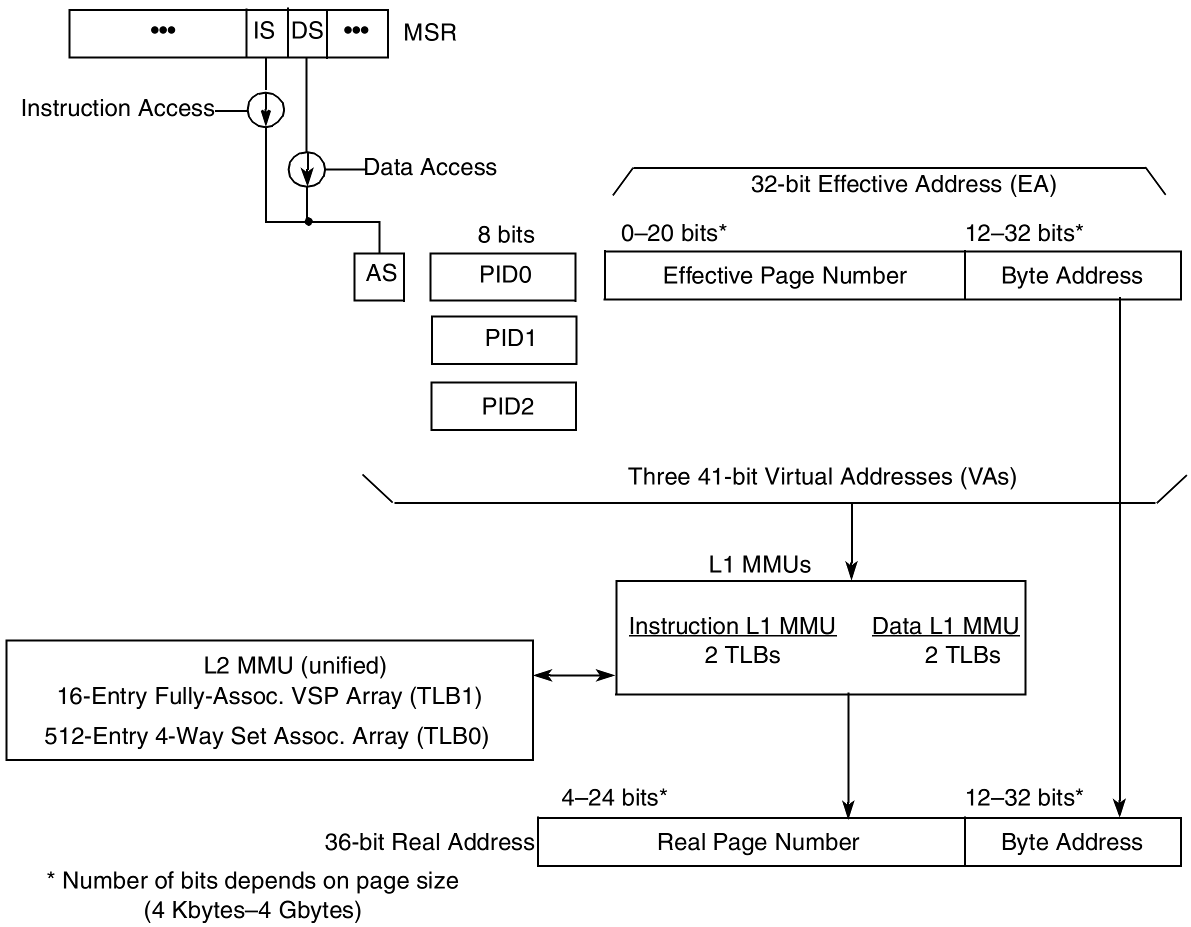


Figure 12-2. Effective-to-Real Address Translation Flow (e500v2)

12.2.1 Virtual Addresses with Three PID Registers

As shown in Figure 12-1 and Figure 12-2, the address translation process starts with an effective address that is prepended with an address space (AS) value and a process ID to construct a virtual address (VA). A virtual address is then translated into a real address based on the translation information found in the on-chip TLB of the appropriate L1 MMU. The AS bit for the access is selected from the value of MSR[IS] or MSR[DS] for instruction or data accesses, respectively.

The e500 constructs three virtual addresses for each access. The core complex implements three process ID (PID) registers, PID0–PID2, as SPRs shown in Section 2.12.1, "Process ID Registers (PID0–PID2)." All of the current values in the PID registers are used in the TLB look-up process and compared with the TID field in all the TLBs. If any of the PID values in PID0–PID2 matches with a TLB entry in which all the other match criteria are met, that entry is used for translation.

Note that when a TID value in a TLB entry is all zeros, it causes a match in the PID compare (effectively ignoring the values of the PID registers). Thus, the operating system can set the values of all the TIDs to zero, effectively eliminating the PID values from all translation comparisons.

The simplest method of using multiple PID registers is to use one PID register for each protected process address space, and a second PID register if the operating system wishes to share TLB entries that map shared memory among different address spaces.

12.2.2 Variable-Sized Pages

There are two kinds of TLBs on the e500 core complex as follows:

- TLBs that translate addresses for 4-Kbyte pages only. These TLBs are set-associative based on the page number (page address).
- TLBs that translate addresses for variable-sized pages. These TLBs are fully-associative.

Table 12-2 shows the nine (e500v1) or eleven (e500v2) page sizes supported by the fully-associative TLBs that support variable-sized pages (VSPs) on the e500 core complex.

Table 12-2. Page Sizes for L1VSPs and TLB1 (L2 MMU) on the e500 Core

Core	Page Sizes
e500 (both e500v1 and e500v2)	4 Kbyte
	16 Kbyte
	64 Kbyte
	256 Kbyte
	1 Mbyte
	4 Mbyte
	16 Mbyte
	64 Mbyte
	256 Mbyte
e500v2	1 Gbyte
	4 Gbyte

For more information on the bit ranges of effective page numbers and offsets that are translated for these pages sizes, see the EREF.

12.2.3 Checking for TLB Entry Hit

Figure 12-3 shows the compare function used by the e500 to check the MMU structures for a hit for the three virtual addresses that correspond to the instruction or data access (one virtual address for each current PID register value). Note that this figure is functionally similar to the figure in the EREF that shows the Book E algorithm, except that this figure shows that three PID values are compared for each access.

A hit to multiple matching TLB entries is considered a programming error. If this occurs, the TLB generates an invalid address and TLB entries may be corrupted (an exception is not reported).

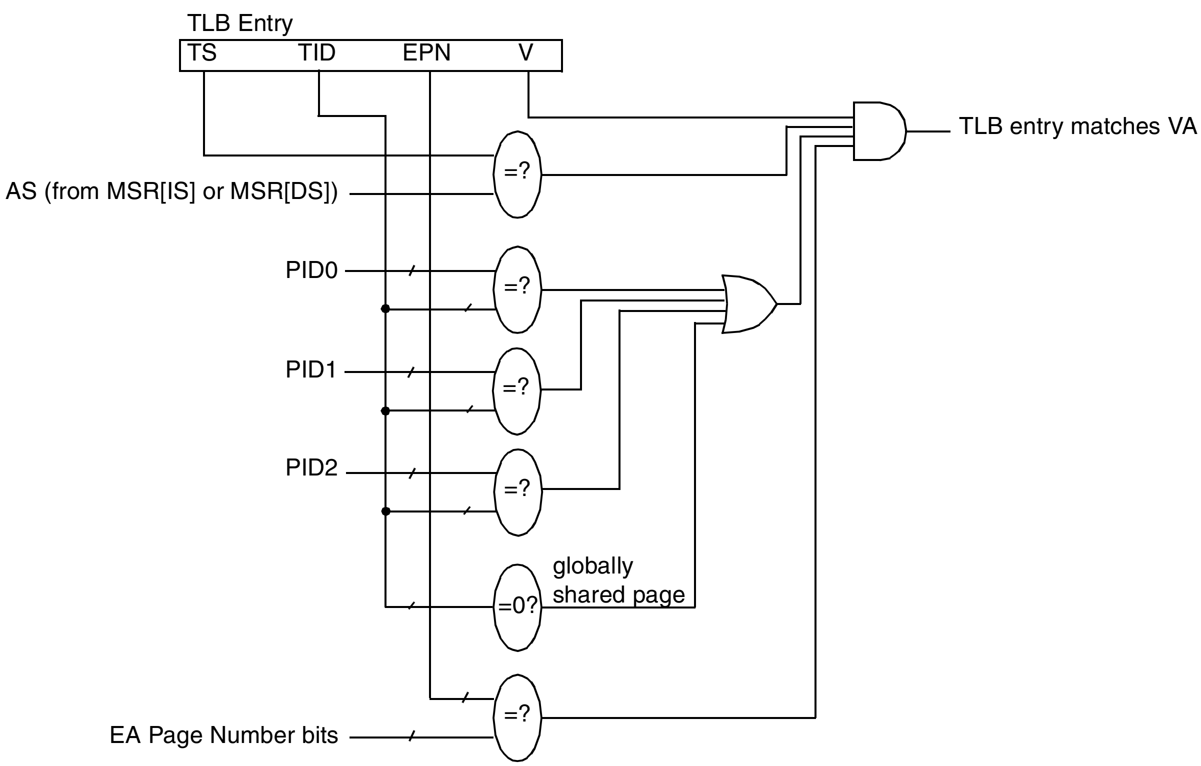


Figure 12-3. Virtual Address and TLB-Entry Compare Process

12.2.4 Checking for Access Permissions

When a TLB entry matches with one of the three virtual addresses of an access, the permission bits of the TLB entry are compared with attribute information of the access (read/write, instruction/data, user/supervisor) to see if the access is allowed to that page. The checking of permissions on the e500 functions as described in the EREF.

12.3 Translation Lookaside Buffers (TLBs)

The e500 core complex implements six TLB arrays to maximize address translation performance and to provide ample flexibility for the operating system. Figure 12-4 contains a more detailed description of the 2-level MMU structure. Note that for an instruction access, both the I-L1VSP and the I-L1TLB4K are checked in parallel for a TLB hit. Similarly, for a data access, both the D-L1VSP and the D-L1TLB4K are checked in parallel for a TLB hit. The instruction L1 MMU and data L1 MMU operate independently and can be accessed in parallel, so that hits for instruction accesses and data accesses can occur in the same clock. This figure shows both the 32-bit real addresses used in the e500v1 and the 36-bit real addresses used in the e500v2. It also shows both the 2-way set associative TLB0 in the e500v1 and the 4-way set associative TLB0 in the e500v2.

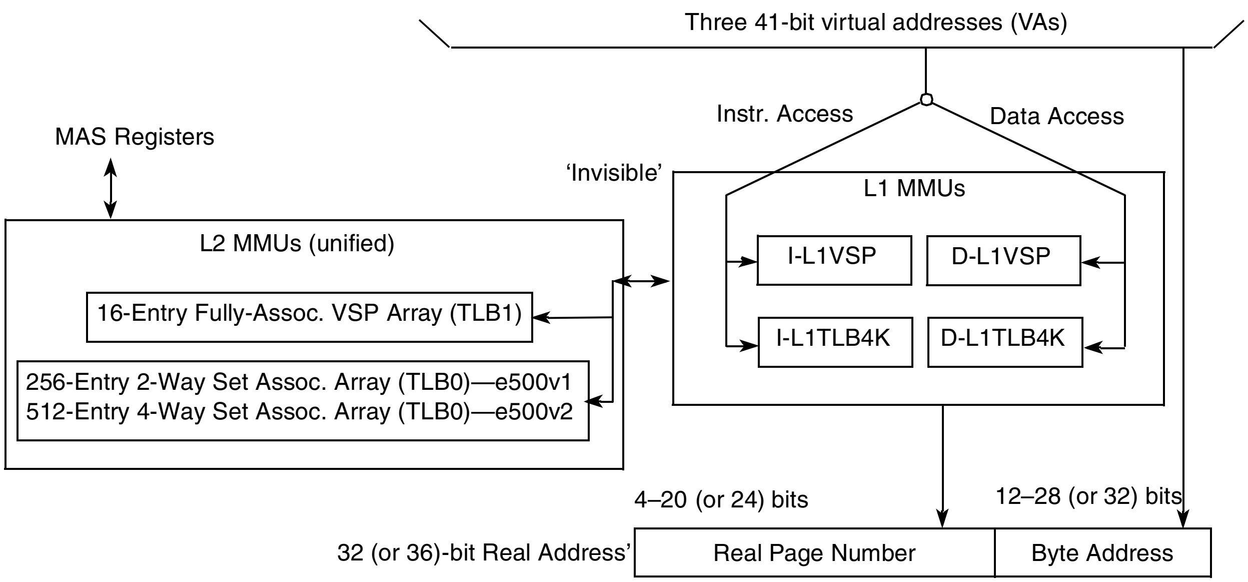


Figure 12-4. Two-Level MMU Structure

Additionally, Figure 12-4 shows that when the L2 MMU is checked for a TLB entry, both TLB1 and TLB0 are checked in parallel. It also identifies the L1 MMUs as invisible to the programming model (not accessible to the operating system); they are managed completely by the hardware as inclusive caches of the corresponding L2 MMU TLB entries. Conversely, the L2 MMU is accessed by the TLB instructions by way of the MAS registers.

A hit to multiple TLB entries in the L1 MMU (even if they are in separate arrays) is considered to be a programming error. This is also the case if an access results in a hit to multiple TLB entries in the L2 MMU. If this occurs, the TLB generates an invalid address and TLB entries may be corrupted (an exception is not reported).

Table 12-3 lists the various TLBs and describes their characteristics. Note that the e500v2 supports eleven page sizes (as shown in parentheses).

Table 12-3. Index of TLBs

Location	Name	Page Sizes Supported	Associativity	Size of TLB (# of entries)	Instruction/Data Translations	Filled by
Instruction L1 MMU	I-L1VSP	9 (or 11) page sizes [1]	Fully associative	4	Instruction	TLB1 hit
	I-L1TLB4K	4 Kbyte	4-way	64	Instruction	TLB0 hit
Data L1 MMU	D-L1VSP	9 (or 11) page sizes [1]	Fully associative	4	Data	TLB1 hit
	D-L1TLB4K	4 Kbyte	4-way	64	Data	TLB0 hit
L2 MMU	TLB1	9 (or 11) page sizes [1]	Fully associative	16	Unified (I and D)	**tlbwe** instruction
	TLB0	4 Kbyte	2-way (e500v1) 4-way (e500v2)	256 (e500v1) 512 (e500v2)	Unified (I and D)	**tlbwe** instruction

[1] See Table 12-2 for supported page sizes.

12.3.1 L1 TLB Arrays

As shown in Figure 12-1, there are two level 1 (L1) MMUs in the core complex. As shown in Figure 12-4 and Table 12-3, the instruction and data L1 MMUs each implement a 4-entry, fully associative L1VSP array and a 64-entry, 4-way set associative L1TLB4K array, comprising the following L1 MMU arrays:

- Instruction L1VSP—4-entry, fully-associative
- Instruction L1TLB4K—64-entry, 4-way set-associative
- Data L1VSP—4-entry, fully associative
- Data L1TLB4K—64-entry, 4-way set-associative

As their names imply, the L1TLB4K arrays only support a 4-Kbyte page size while the L1VSP arrays support nine (e500v1) or eleven (e500v2) page sizes. To perform a lookup for instruction accesses, both the L1TLB4K and the L1VSP TLBs in the instruction MMU are searched in parallel for the matching TLB entry. Similarly, for data accesses, both the L1TLB4K and the L1VSP TLBs in the data MMU are searched in parallel for the matching TLB entry. The contents of a matching TLB entry are then concatenated with the page offset of the original effective address; the bit range that is translated is determined by the page size. The result constitutes the real (physical) address for the access.

Figure 12-5 shows the organization of the L1 TLBs in both the instruction and data L1 MMUs.

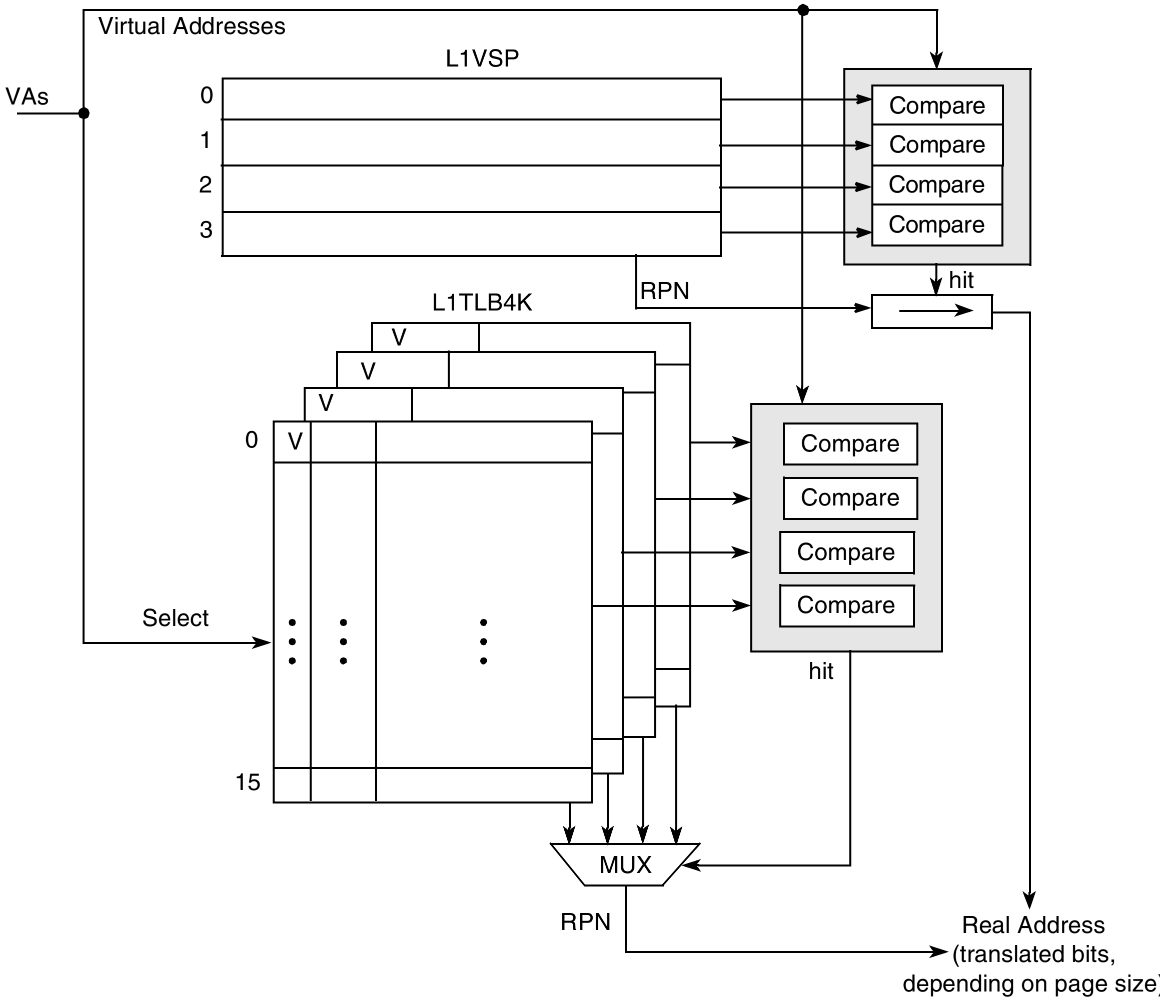


Figure 12-5. L1 MMU TLB Organization

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 (externally) exactly which entry is the next to be replaced.

12.3.2 L2 TLB Arrays

The level 1 MMUs are backed up by a unified level 2 MMU that translates both instruction and data addresses. Like each L1 MMU, the L2 MMU consists of two TLB arrays:

- TLB1: a 16-entry, fully associative array that supports nine (e500v1) or eleven (e500v2) page sizes.
- TLB0: a 256-entry, 2-way (e500v1) or 512-entry, 4-way (e500 v2) set associative array that supports only 4-Kbyte page sizes.

The two L2 TLBs on the e500v1, which are the only TLBs accessible to the software, are shown in Figure 12-6.

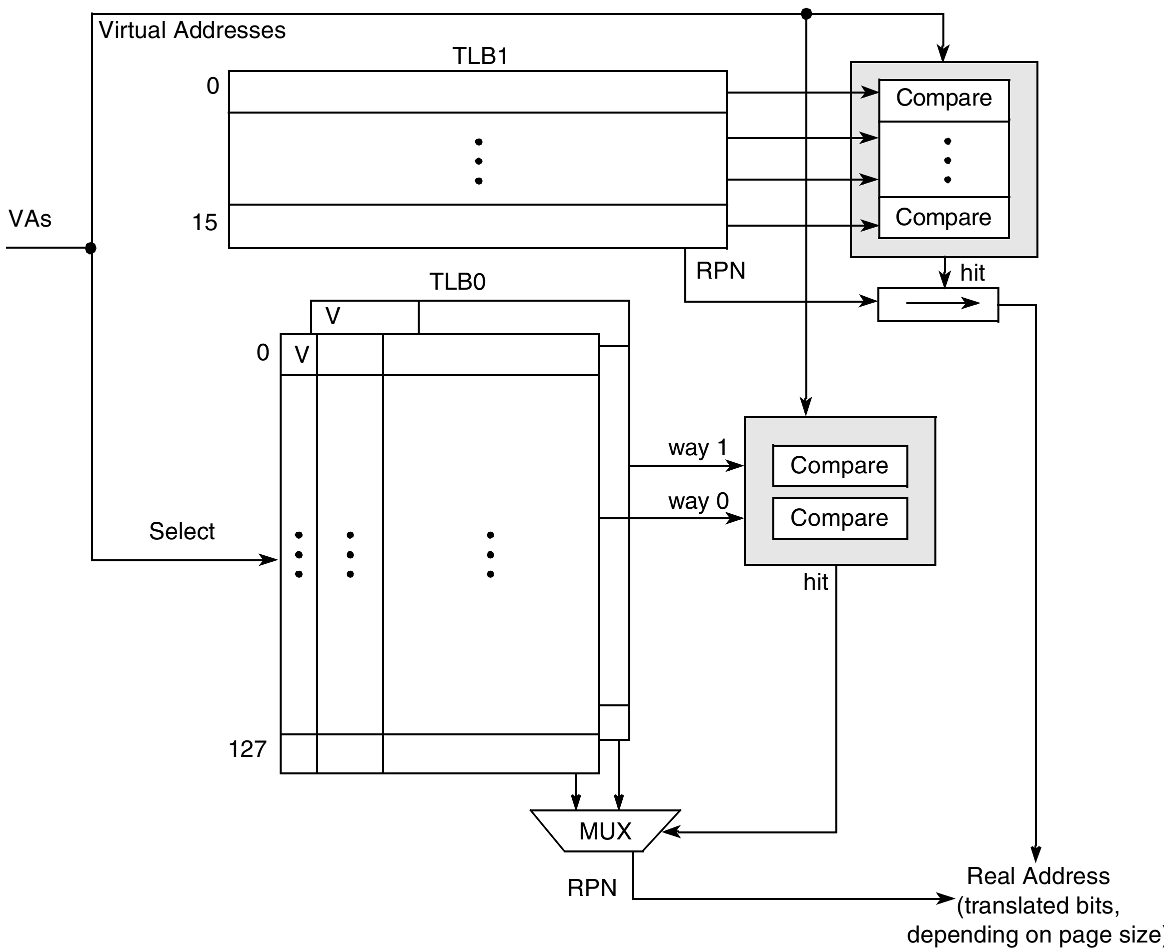


Figure 12-6. L2 MMU TLB Organization—e500v1

The equivalent figure for the e500v2 is shown in Figure 12-7.

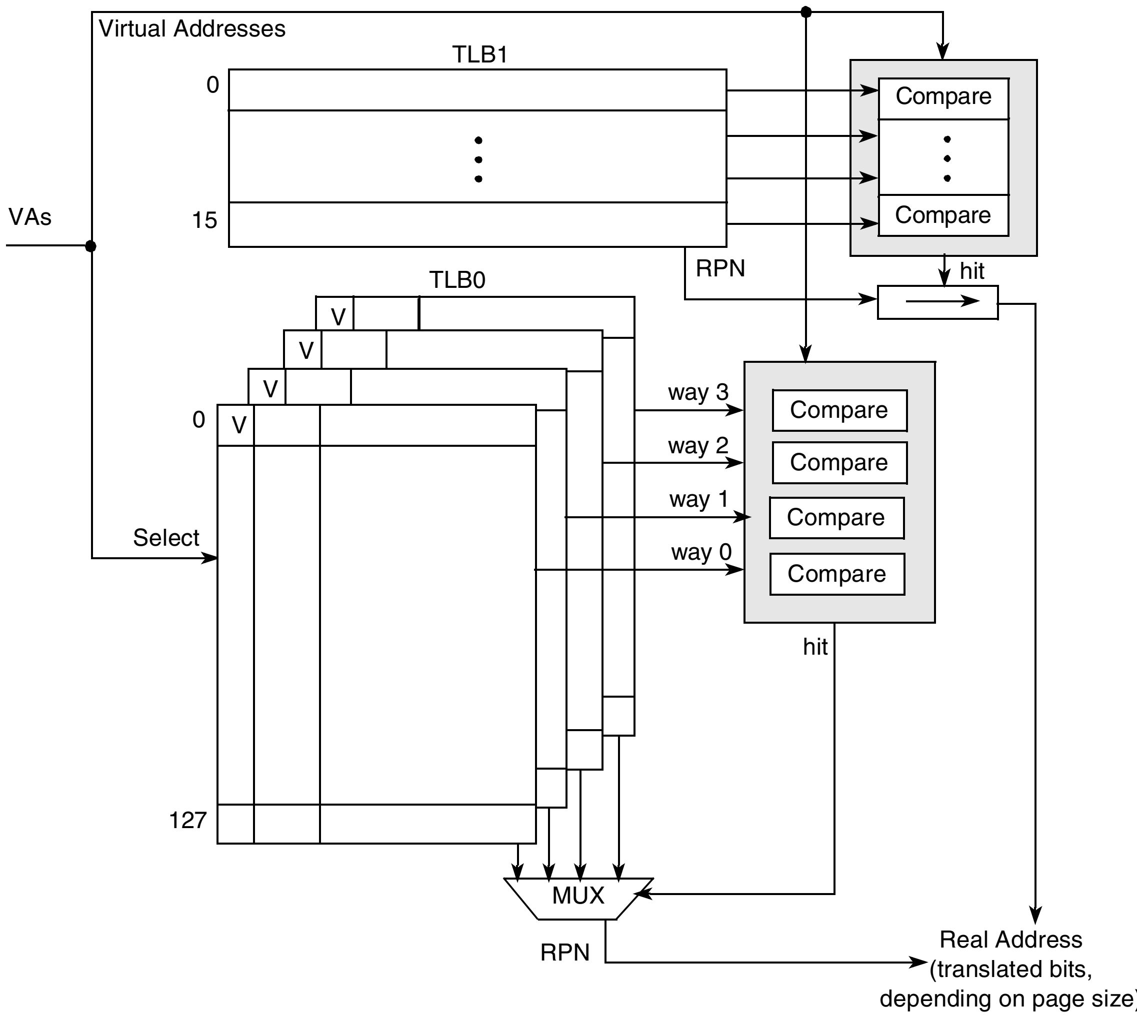


Figure 12-7. L2 MMU TLB Organization—e500v2

12.3.2.1 IPROT Invalidation Protection in TLB1

The IPROT bit in TLB1 is used to protect TLB entries from invalidation. TLB1 entries with IPROT set can never be invalidated by a **tlbivax** instruction executed by this processor (even when the INV_ALL command is indicated) (internal case), by an external **tlbivax** instruction, or by a flash invalidate initiated by writing to the MMUCSR0. The IPROT bit 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 only be invalidated by writing a 0 to the valid bit of the entry (by using the MAS registers and executing the **tlbwe** instruction).

Only TLB entries in TLB1 can be protected from invalidation; entries in TLB0 and in the level 1 MMUs cannot be protected from invalidation (they don't implement the IPROT bit). See the EREF for more background information on the IPROT attribute.

Invalidation operations are guaranteed to invalidate the entry that translates the address specified in the operand of the **tlbivax** instruction. Other entries may also be invalidated by this operation if they are not protected with IPROT. A precise invalidation can be performed by writing a 0 to the valid bit of a TLB entry. Note that successful invalidation operations in the L2 MMU also invalidate matching entries in the L1 MMU.

If HID1[ABE] = 1, enabling broadcast operations on the core complex bus (CCB), execution of **tlbivax** is broadcast onto the CCB, regardless of whether or not the invalidation was successful. Flash invalidations (initiated by writing to the appropriate bits in MMUCSR0) are never broadcast.

12.3.2.2 Replacement Algorithms for L2 MMU

The replacement algorithm for TLB1 (the fully associative TLB in the L2 MMU) 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 the MAS0[ESEL] field before executing a **tlbwe** instruction.

TLB0 entry replacement is also implemented by software. To assist the software with TLB0 replacement, the e500 core complex provides a hint that can be used for implementing a round-robin replacement algorithm. 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 bits:

- TLB0[NV]—the next victim field within TLB0
- MAS0[NV]—the next victim field of MAS0
- MAS0[ESEL]—selects the way to be replaced on **tlbwe**

See Table 12-15 for a complete description of MAS register updates on various exception conditions.

Note that the system software can load any value into MAS0[ESEL] and MAS0[NV] prior to execution of **tlbwe**, effectively overwriting this round robin replacement algorithm. In this case, the value written by software into MAS0[NV] is used as the next TLB0[NV] value on a TLB miss.

Also, note that the value of MAS0[NV] is indeterminate after any TLB entry invalidate operation (including a flash invalidate). If the software must know its value after an invalidate operation, MAS0[NV] must be explicitly read.

12.3.2.2.1 Round-Robin Replacement for TLB0—e500v1

Figure 12-8 shows the round-robin replacement algorithm for the e500v1 core. Note that for the e500v1, TLB[NV] is implemented as a single bit that corresponds to the least significant bit of the MAS0[NV] field.

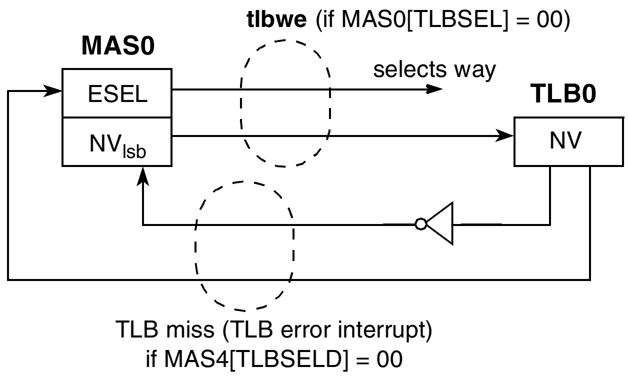


Figure 12-8. Round Robin Replacement for TLB0—e500v1

On execution of a **tlbwe** instruction, MAS0[ESEL] selects the way of TLB0 to be loaded (way 0 or way 1). Also, when MAS0[TLBSEL] = 00 (selecting TLB0), TLB0[NV] is loaded with the MAS0[NV_{lsb}] value on execution of a **tlbwe** instruction. In addition, when a TLB miss exception occurs (causing a TLB error interrupt), if MAS4[TLBSELD] = 00, the hardware automatically loads the current value of TLB0[NV1] into MAS0[ESEL] and the complement of TLB0[NV] into MAS0[NV_{lsb}]. This sets up MAS0 such that if those values are not overwritten, the alternate way will be selected on the next execution of a **tlbwe** instruction, effectively alternating between way 0 and way 1 for writing TLB0 entries.

12.3.2.2.2 Round-Robin Replacement for TLB0—e500v2

The e500v2 core has a 4-way set associative TLB0, and so fully implements the round-robin scheme with a simple 2-bit counter that increments the 2-bit value of NV from TLB0 on each TLB miss and loads the incremented value into MAS0[NV] for use by the next **tlbwe** instruction.

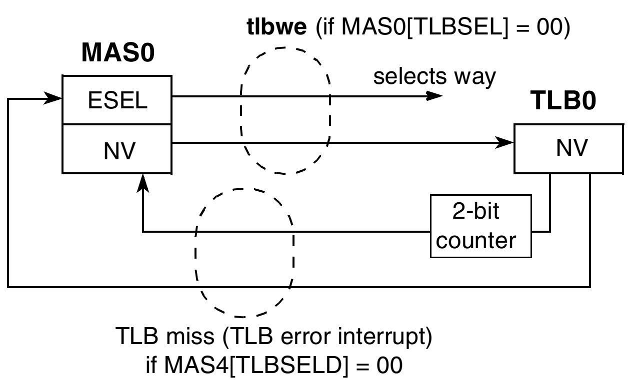


Figure 12-9. Round Robin Replacement for TLB0—e500v2

On execution of a **tlbwe** instruction, MAS0[ESEL] selects the way of TLB0 to be loaded (way 0, 1, 2, or 3). Also, when MAS0[TLBSEL] = 00 (selecting TLB0), the two-bit TLB0[NV] field is loaded with the MAS0[NV] value on execution of a **tlbwe** instruction. When a TLB miss exception occurs (causing a TLB error interrupt), if MAS4[TLBSELD] = 00, the hardware automatically loads the current value of TLB0[NV] into MAS0[ESEL] and the incremented value of TLB0[NV] into MAS0[NV]. This sets up MAS0 such that if those values are not overwritten, the next way will be selected on the next execution of a **tlbwe** instruction.

12.3.3 Consistency Between L1 and L2 TLBs

The contents of the L1 TLBs are always a proper subset of the TLB entries currently resident in the L2 MMU. They serve to improve performance because they have a faster access time than the larger L2 TLBs. The relationships between the six TLBs are shown in Figure 12-10.

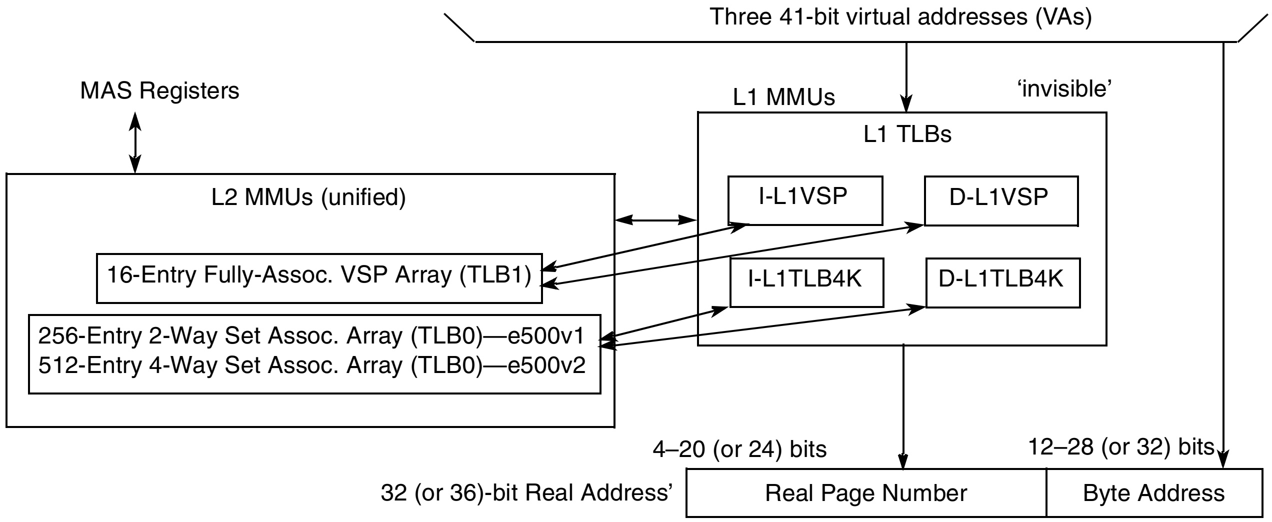


Figure 12-10. L1 MMU TLB Relationships with L2 TLBs

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 one of the three virtual addresses 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.

A hit for a single access to multiple TLB entries in the L2 MMU (even if they are in separate arrays) is considered to be a programming error. If this occurs, the TLB generates an invalid address and TLB entries may be corrupted (an exception is not reported).

A write to any field of a valid L2 TLB entry causes any corresponding L1 TLB entry to be invalidated. Also, changing the value of any PID register causes all L1 TLB entries to be invalidated, except for L1 TLB entries created for TID = 0. Therefore, it is recommended that TID = 0 be used as much as possible to maximize L1 TLB hit rates.

Note that when an L2 TLB entry is invalidated by executing a **tlbwe** instruction that clears a valid bit, any corresponding entry in the L1 TLB arrays is also automatically invalidated. In addition, when L2 TLB entries are invalidated by the execution of **tlbivax**, by the detection of a TLB invalidate command broadcast by another processor, or by a flash invalidate operation, corresponding L1 TLB entries are also invalidated as described in Section 12.4.4, "TLB Invalidate (**tlbivax**) Instruction."

12.3.4 L1 and L2 TLB Access Times

The L1 TLB arrays are checked for a translation hit in parallel with the on-chip L1 cache lookups and incur no penalty on an L1 TLB hit. If the L1 TLB arrays miss, the access proceeds to the L2 TLB arrays. For L1 instruction address translation misses, the L2 TLB latency is at least 5 clocks; for L1 data address translation misses, the L2 TLB latency is at least 6 clocks. These access times may be longer depending on some arbitration performed by the L2 arrays for simultaneous instruction L1 TLB misses, data L1 TLB misses, the execution of TLB instructions, and TLB snoop operations (snooping of TLBINV operations on the CCB).

Note that when a TLBINV operation is detected on the CCB, the L2 MMU arrays become inaccessible due to the snooping activity caused by the TLBINV.

12.3.5 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 e500, if the page for this type of access is marked with G = 0 (unguarded), this type of access may be issued to the CCB regardless of the completion status of other instructions. If G = 1 (guarded), the access stalls (if it misses in the cache) until the exception status of any instructions in progress is known.

When G = 1 for the page, data accesses that miss in the cache are not issued to the CCB until the instruction is known to be required by the program execution model; that is, all previous instructions will have completed without exception and no asynchronous interrupts occur between the time that the access is issued to the CCB and the time that the CCB 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 CCB.

Note that after an access with G = 1 is begun to the CCB, it is guaranteed to be completed. That is, after the address tenure is acknowledged on the CCB, the core completes the access, even if an asynchronous interrupt is pending.

The G bit is ignored for instruction fetches, and instructions are speculatively fetched from guarded pages. To prevent speculative fetches from pages that do not contain instructions and are guarded, the page should be also designated as no-execute (with the UX/SX page permission bits cleared).

12.3.6 TLB Entry Field Definitions

Table 12-4 summarizes the fields of e500 TLB entries. Note that all of these fields are defined at the Freescale Book E level. See the EREF for the definition of TLB fields at the Freescale Book E level.

Table 12-4. TLB Entry Bit Definitions for e500

Field	Comments
V	Valid bit for entry
TS	Translation address space (compared with AS bit of the current access)
TID[0–7]	Translation ID (compared with PID0, PID1, PID2 or TIDZ (all zeros))
EPN[0–19]	Effective page number (compared with EA[32–51] for 4-Kbyte pages)
RPN[0–19] (e500v1); RPN[0–23] (e500v2)	Real page number Translated address RA[32–51] for 4-Kbyte pages for e500v1 Translated address RA[28–51] for 4-Kbyte pages for e500v1
SIZE[0–3]	Encoded page size 0000 Reserved 0001 4 Kbyte 0010 16 Kbyte 0011 64 Kbyte 0100 256 Kbyte 0101 1 Mbyte 0110 4 Mbyte 0111 16 Mbyte 1000 64 Mbyte 1001 256 Mbyte 1010 1 Gbyte (for e500v2 only) 1011 4 Gbyte (for e500v2 only) all others—reserved
PERMIS[0–5]	Supervisor execute, write, and read permission bits, and user execute, write, and read permission bits.
WIMGE	Memory/cache attributes (write-through, cache-inhibit, memory coherence required, guarded, endian)
X0, X1	Extra system attribute bits (for definition by system software)
U0–U3	User attribute bits—used only by software. These bits exist in the L2 MMU TLBs only (TLB1 and TLB0)
IPROT	Invalidation protection (exists in TLB1 only)

12.4 TLB Instructions—Implementation

As described in the Cache and MMU Background chapter of the EREF, the TLBs are accessed indirectly through MMU assist (MAS) registers. Software can write and read the MMU assist registers with **mtspr** and **mfspr** instructions. These registers contain information related to reading and writing a given entry within the TLBs. For example, data is read from the TLBs into the MAS registers with a TLB Read Entry (**tlbre**) instruction, and data is written to the TLBs from the MAS registers with a TLB Write Entry (**tlbwe**) instruction.

The implementation of the **tlbre**, **tlbwe**, **tlbsx**, **tlbivax**, and **tlbsync** instructions is summarized in this section. The extended (64-bit) forms of these instructions are invalid for the core complex. See Section 3.1.4, "Unsupported Book E Instructions." Although the **tlbre**, **tlbwe**, **tlbsx**, **tlbivax**, and **tlbsync** instructions are defined by Book E, their specific functions are defined by Freescale Book E.

12.4.1 TLB Read Entry (tlbre) Instruction

The **tlbre** instruction causes the contents of a single TLB entry to be extracted from the L2 MMU and placed in the corresponding fields of the MMU assist (MAS) registers. The entry extracted is specified by the TLBSEL, ESEL, and EPN fields of the MAS0, and MAS2 registers. The contents extracted from the L2 MMU are placed in MAS1, MAS2, and MAS3. Note that for the e500v2, if HID0[EN_MAS7_UPDATE] = 1, MAS7 is also updated with the four highest-order bits of physical address for the TLB entry. See Section 12.7.2, "MAS Register Updates," for details on which MAS register fields are updated.

The following RTL describes the e500 core complex **tlbre** implementation:

```
tlb_entry_id = MAS0(TLBSEL, ESEL) || MAS2(EPN)
result = L2MMU(tlb_entry_id)
MAS0, MAS1, MAS2, MAS3, (and MAS7 if HID0[EN_MAS7_UPDATE] = 1) = result
```

Note that architecturally, if the instruction specifies a TLB entry that is not found, the results placed in MAS0–MAS3 (and optionally, MAS7) are undefined. However, for the e500, the TLBSEL, ESEL and EPN fields always index to an existing L2 TLB entry and that indexed entry is read. Note that EPN bits are only used to index into TLB0. In the case of TLB1, the EPN field is unused for **tlbre**. See the EREF for information at the Freescale Book E level.

12.4.1.1 Reading Entries from the TLB1 Array

Entries in TLB1 can be read by first writing the necessary entry-identifying information into MAS0 using **mtspr** and then executing the **tlbre** instruction. To read an entry from TLB1, MAS0[TLBSEL] must be = 01 and MAS0[ESEL] must be set to point to the desired entry. After executing the **tlbre** instruction, MAS0–MAS3 (and optionally, MAS7 for the e500v2) are updated with the data from the selected TLB entry in TLB1.

12.4.1.2 Reading Entries from the TLB0 Array

Entries in TLB0 can be read by first writing the necessary entry-identifying information into MAS0 and MAS2 using **mtspr** and then executing the **tlbre** instruction. To read an entry from TLB0, MAS0[TLBSEL] must be = 00, MAS0[ESEL] must be set to point to the desired way, and EPN[45–51] in MAS2 must be loaded with the desired index. After executing the **tlbre** instruction, MAS0–MAS3 (and optionally, MAS7 for the e500v2) are updated with the data from the selected TLB entry in TLB0.

12.4.2 TLB Write Entry (tlbwe) Instruction

The **tlbwe** instruction causes the contents of certain fields of the MAS registers (MAS0, MAS1, MAS2, and MAS3) to be written into a single TLB entry in the L2 MMU. Execution of the **tlbwe** instruction on the e500v2 core also causes the upper 4 bits of the RPN that reside in MAS7 to be written to the selected TLB entry. The entry written is specified by the TLBSEL, ESEL, and EPN fields of the MAS0, and MAS2 registers.

The following RTL describes the e500 core complex **tlbwe** implementation:

```
tlb_entry_id = MAS0(TLBSEL, ESEL) || MAS2(EPN)
L2MMU(tlb_entry_id) = MAS0, MAS1, MAS2, MAS3, (and MAS7 on e500v2)
```

Note that when an L2 TLB entry is written, it may be displacing an already valid entry in the same L2 TLB location (a victim). If a valid L1 TLB entry corresponds to the L2 MMU victim entry, that L1 TLB entry is automatically invalidated. See the EREF for synchronization requirements defined at the Freescale Book E level for the use of **tlbwe**.

12.4.2.1 Writing to the TLB1 Array

TLB1 can be written by first writing the necessary information into MAS0–MAS3 (and MAS7 for the e500v2) using **mtspr** and then executing the **tlbwe** instruction. To write an entry into TLB1, MAS0[TLBSEL] must = 01, and MAS0[ESEL] must point to the desired entry. When the **tlbwe** instruction is executed, the TLB entry information stored in MAS0–MAS3 (and MAS7 for the e500v2) is written into the selected TLB entry in the TLB1 array.

12.4.2.2 Writing to the TLB0 Array

TLB0 can be written by first writing the necessary information into MAS0–MAS3 (and MAS7 for the e500v2) using **mtspr** and then executing the **tlbwe** instruction. To write an entry into TLB0, MAS0[TLBSEL] must = 00, MAS0[ESEL] must point to the desired way, and EPN[45–51] in MAS2 must be loaded with the desired index. When the **tlbwe** instruction is executed, the TLB entry information stored in MAS0–MAS3 (and MAS7 for the e500v2) is written into the selected TLB entry in TLB0.

12.4.3 TLB Search (tlbsx) Instruction—Searching the TLB1 and TLB0 Arrays

The **tlbsx** instruction updates the MAS registers conditionally based on the success or failure of a TLB lookup in the L2 MMU. The lookup is controlled by the effective address provided by GPR[**r**A] + GPR[**r**B] specified in the instruction encoding, as well as by the SAS and SPID0 search fields in MAS6. The values placed into MAS0, MAS1, MAS2, MAS3, and optionally, MAS7 differ, depending on whether a successful or unsuccessful search occurred. See Section 12.7.2, “MAS Register Updates,” for details on which MAS register fields are updated for these cases.

Note that **r**A = 0 is the preferred form for **tlbsx** and that some Freescale implementations, such as the e500, take an illegal instruction exception program interrupt if **r**A!=0.

The following RTL describes the e500 core complex **tlbsx** implementation:

```
if RA!=0 then generate illegal exception
EA = 32 0 || GPR(RB) 32:63
ProcessID = MAS6(SPID0), 0b0000_0000
AS = MAS6(SAS)
VA = AS || ProcessID || EA
if Valid_TLB_matching_entry_exists (VA)
   then result = see  Table 12-15, column labelled “tlbsx hit”
else result = see  Table 12-15, column labelled “tlbsx miss”
MAS0, MAS1, MAS2, MAS3 = result
```

The **tlbsx** instruction searches both the TLB1 and TLB0 arrays using EPN[32–51] from the GPR used as the instruction operand, and the SAS (search AS bit) and SPID0 (search PID) values from MAS6. If the search results in a hit, the information for the TLB entry that hit is loaded into MAS0–MAS3 and optionally, MAS7. The valid bit in MAS1 is used as the success flag as follows:

- If the search is successful, MAS1[V] is set.
- If the search is unsuccessful, MAS1[V] is cleared.

The **tlbsx** instruction is especially useful for finding the TLB entry that caused a DSI or ISI exception. In this case, at most three **tlbsx** instructions are required: one for each of the current PID values. Note that TID values of 0x00 always match with any PID value. Thus, if software only uses one PID register, only one search is required.

12.4.4 TLB Invalidate (tlbivax) Instruction

The following RTL describes the e500 core complex **tlbivax** implementation:

```
if RA = 0, a = 0
else, a = GPR(RA)
EA = a + GPR(RB)
if (valid_TLB_matching_entry exists or INV_ALL) and Entry_IPROT_not_set
then invalidate entry
```

A TLB invalidate operation is performed whenever a **tlbivax** instruction is executed. This instruction invalidates any TLB entry that corresponds to the virtual addresses calculated by this instruction. This operation includes invalidating TLB entries contained in TLBs on other processors and devices in addition to the processor executing the **tlbivax** instruction. Thus an invalidate operation is broadcast throughout the coherent domain of the processor executing this instruction.

Because the virtual address can be much larger than the physical address, the full virtual address specified by the **tlbivax** instruction cannot be broadcast to all devices. Instead, a subset address is broadcast that fits within the space of the implemented physical addressing model.

The address that is used by the processor executing the **tlbivax** instruction is detailed in Table 12-5. Note that this subset address is also the address broadcast to other processors. Thus, no other information except for that shown in Table 12-5 is used for the invalidation. As shown in the table, the bits of effective address used to perform the **tlbivax** invalidation of TLB1, TLB0, and the L1 TLBs are bits 32–51 of **r**A + **r**B.

Table 12-5. tlbivax EA Bit Definitions

Bits of (rA + rB) (preferred form is for rA = 0)	**Meaning**	**More Information Section/Page**
32–51	EA[32–51] for invalidation matching	—
52–59	Reserved; should be zero	—
60	TLBSEL. Selects which TLB is targeted for invalidation 0 TLB0 1 TLB1	12.4.4.1/12-21
61	INV_ALL command	12.4.4.2/12-22
62–63	Reserved	—

The limited virtual address used to invalidate TLB entries has the side effect that a single **tlbivax** instruction can invalidate more than a single entry in a targeted TLB. This is because the **tlbivax** does not compare the values of the PID or AS bits. A **tlbivax** targeted at TLB0 can invalidate either or both ways within an TLB0 index (for e500v1), up to all four ways for e500v2, and up to all four ways within an L1TLB4K index. Also, a **tlbivax** targeted at TLB1 can invalidate up to all 16 entries in the array, or up to all 8 entries of the L1VSPs (instruction and data).

The **tlbivax** instruction invalidates all matching entries in the instruction and data L1 TLBs simultaneously. Also, the core complex always snoops TLB invalidate transactions from other CCB bus masters (if any) and invalidates matching TLB entries accordingly.

Note that entries in TLB1 can be protected from invalidation by the **tlbivax** instruction by setting the IPROT bit for those entries. See the EREF for more information on the use of the IPROT bit defined for Freescale Book E processors.

12.4.4.1 TLB Selection for tlbivax Instruction

Because only a limited subset of the virtual address can be broadcast, extra information about the targeted TLB entries is encoded in two of the lower bits of the effective address calculated by the **tlbivax** instruction. Bit 60 of the **tlbivax** effective address is interpreted as the TLBSEL field. This bit indicates whether TLB1 or TLB0 is targeted by the invalidate operation. Because only a few bits (32–51) of address are broadcast and can be used in the invalidate comparison for TLB1, and most of those bits are masked out for larger page sizes, the TLBSEL field avoids unnecessary invalidations of large superpages in TLB1 when the **tlbivax** is targeting TLB0.

12.4.4.2 Invalidate All Address Encoding for tlbivax Instruction

Bit 61 of the **tlbivax** effective address is interpreted as the INV_ALL command. If this bit is set, it indicates that the invalidate operation should completely invalidate all entries of either TLB1 or TLB0 as indicated by the TLBSEL field, and invalidate all corresponding L1 TLB entries. Note that entries in TLB1 can be protected from this type of invalidation by setting the IPROT bit as described in Section 12.3.2.1, “IPROT Invalidation Protection in TLB1.”

12.4.4.3 TLB Invalidate Broadcast Enabling

In addition to invalidating the local matching TLB entries, the **tlbivax** instruction operation is also broadcast on the bus (causing a TLBINV address-only transaction) according to the value of the ABE (address broadcast enable) bit in the HID1 register as follows:

- If HID1[ABE] = 0, **tlbivax** instructions are not broadcast.
- If HID1[ABE] = 1, **tlbivax** instructions are broadcast.

12.4.5 TLB Synchronize (tlbsync) Instruction

The **tlbsync** instruction causes a TLBSYNC transaction on the CCB. This transaction is retried if any processor, including the one that executed the **tlbsync** instruction, has pending memory accesses that were issued before any previous **tlbivax** instructions were completed. 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.

12.5 TLB Entry Maintenance—Details

The TLB entries of the e500 core complex must be loaded and maintained by the system software, including performing the required table search operations in memory. However, the e500 provides some hardware assistance for these software tasks. Note that the system software cannot directly access the L1 TLBs, and the L1 TLBs are completely and automatically maintained in hardware as a subset of the contents of the L2 TLBs.

In addition to the resources described in Table 12-1, hardware assistance on the core complex for maintenance of TLB entries includes:

- Automatic loading of MAS0–2 based on the default values in MAS4 on 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 condition.
- Automatic loading of the data exception address register (DEAR) with the effective address of the load, store, or cache management instruction that caused an alignment, data TLB miss (data TLB error interrupt), or permissions violation (DSI interrupt).

- Automatic loading into SRR0 of the effective address of the instruction that causes a TLB miss exception or a permissions violation.
- 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 = 00. See Section 12.3.2.2, "Replacement Algorithms for L2 MMU."
- When **tlbwe** is executed, 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 is also resident in an L1 MMU, it is invalidated (or victimized) in the L1 MMU. This forces inclusion in the TLB hierarchy. Additionally, the new TLB entry contained in MAS0–MAS3 (and MAS7 on the e500v2) is written into the selected TLB.

Note that while the **tlbwe** instruction loads an entry in the L2 TLB array, it does not load an entry in the L1 TLB array. The 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 12.7.2, "MAS Register Updates," for a complete description of automatic fields loaded into the MAS registers on execution of TLB instructions and for various exception conditions.

The EREF provides more information on some of the actions taken by Freescale Book E devices on MMU exceptions.

The following subsections provide supplementary information that applies for the e500.

12.5.1 Automatic Updates—TLB Miss Exceptions

When a TLB miss exception occurs, MAS0–MAS2 are automatically updated using the defaults specified in MAS4, as well as the AS and EPN[32–51] values corresponding to the access that caused the exception, as described in Section 12.7.2, "MAS Register Updates."

In addition, if TLBSELD = 00 (selecting TLB0), MAS0[ESEL] is updated with the next victim information for TLB0. Finally, the MAS0[NV] field is updated with the incremented value of TLB0[NV]. Thus, ESEL points to the current victim (the entry to be replaced), while MAS0[NV] points to the next victim to be used if a TLB0 entry is replaced. See Section 12.3.2.2, "Replacement Algorithms for L2 MMU," for more information.

The process described above sets up all the TLB entry data necessary for a TLB write except for RPN[32–51] and RPN[28–31], the U0–U3 user attribute bits, and the UX, SX, UW, SW, UR, and SR permission bits for the new entry, all of which are stored in MAS3 (and MAS7). Thus, if the defaults stored in MAS4 are applicable to the TLB entry to be loaded, the TLB miss exception handler only has to update MAS3 (and MAS7) with an **mtspr** before executing **tlbwe**. If the defaults are not applicable to the TLB entry being loaded, then the TLB miss exception handler must update MAS0–MAS2 appropriately before performing the TLB write. See Section 12.5.2, "TLB Interrupt Routines," for more information on the handling of TLB miss exceptions.

12.5.2 TLB Interrupt Routines

When an exception is reported by the MMUs, the machine drains (that is, all instructions dispatched prior to the exception are executed). After all instructions are completed, the interrupt is acknowledged and MAS0–MAS2 are loaded as described in Section 12.5.1, "Automatic Updates—TLB Miss Exceptions."

As is recommended for most interrupt handler routines, the TLB miss, DSI, and ISI exception handlers must first save the values of enough GPRs so that the handler has enough GPRs available for its own use. The handler should then perform an **mfcr** to copy the CR data into one of the GPRs. Before exiting the handler, an **mtcrf** must be executed to restore the CR, and then the original GPR data must be restored.

The PID0–2 registers must also be restored (if modified) before exiting the handler. Note that PID register updates must be followed by an **isync**. This **isync** instruction must reside in an instruction page that is valid before the changes are made to the PID.

12.5.2.1 Permissions Violations (ISI, DSI) Interrupt Handlers

The only differences between the definition of actions on a permissions violation for Freescale Book E devices and for the e500 is that the e500 only uses MAS6[SPID0] and the e500 does not implement MAS5. Note that for a permissions violation case, software must explicitly load a value into MAS6[SPID0] (this value will most likely be the value of PID0).

The permissions violations handlers can use the **tlbsx** instruction to load all necessary information about the faulting access into the MAS registers and make the appropriate changes. If the access was an instruction or data access, the handler can load the following effective address into **r**B in order to load the faulting TLB entry into the MAS registers:

- Instruction access: load SRR0 value into **r**B
- Data access: load DEAR value into **r**B

See Section 12.4.3, "TLB Search **(tlbsx)** Instruction—Searching the TLB1 and TLB0 Arrays," for more information about the actions performed by the **tlbsx** instruction.

The guidelines for the saving and restoring of resources for permissions violations interrupt handlers are the same as that for TLB error interrupts.

12.6 TLB States after Reset

During reset, all TLB entries in the L1 and L2 MMUs are flash invalidated. Then entry 0 of TLB1 is loaded with the values shown in Table 12-6. Note that only the valid bits for other TLB entries are cleared. Other fields of TLB entries are set not set to a known state and software should be careful to insure that all fields of a TLB entry are appropriately initialized through the MAS registers before it is used for translation.

Note also that because the core complex fetches from effective address 0xFFFF_FFFC out of reset, the first access out of reset is automatically translated with this default TLB entry. The instruction located at 0xFFFF_FFFC should be a branch instruction to the beginning of this 4-Kbyte page.

Because this default entry only translates a 4-Kbyte page, the initial code in this page needs to set up more valid TLB entries (and pages) so that the program can branch out of this 4-Kbyte page 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 that exceptions can be handled early in the booting process.

Table 12-6. TLB1 Entry 0 Values after Reset

Field	Reset Value	Comments
V	1	Entry is valid
TS	0	Address space 0
TID[0–7]	0x00	TID value for shared (global) page
EPN[32–51]	0xFFFFF	Address of last 4-Kbyte page in address space
RPN[32–51]	0xFFFFF	Address of last 4-Kbyte page in address space
SIZE[0–3]	0001	4-Kbyte page size
SX/SR/SW	111	Full supervisor mode access allowed
UX/UR/UW	000	No user mode access allowed
WIMGE	01000	Caching-inhibited, non-coherent, big-endian
X0–X1	00	Reserved system attributes
U0–U3	0000	User attribute bits
IPROT	1	Page is protected from invalidation

12.7 Core Complex MMU Registers

Table 12-7 provides cross-references to other sections that have more detailed bit descriptions for the e500 registers related to the MMU. Also, the EREF lists the Freescale Book E definitions for these registers.

Table 12-7. Registers Used for MMU Functions

Registers	Comprehensive Reference (Section/Page)	Additional e500-Only Reference (Section/Page)
Process ID (PID0–PID2)	2.12.1/2-36	—
MMU control and status register (MMUCSR0)	2.12.2/2-36	—
MMU configuration register (MMUCFG)	2.12.3/2-37	—
TLB configuration registers (TLB0CFG–TLB1CFG)	2.12.4/2-37	—

Table 12-7. Registers Used for MMU Functions (continued)

Registers	Comprehensive Reference (Section/Page)	Additional e500-Only Reference (Section/Page)
MMU assist registers (MAS0–MAS4, MAS6 (and MAS7 for the e500v2))	2.12.5/2-39	12.7.1/12-26
Data exception address register (DEAR)	2.7.1.3/2-18	—

12.7.1 e500 MAS Registers

The core complex uses seven special purpose registers (MAS0, MAS1, MAS2, MAS3, MAS4, MAS6, and MAS7) to facilitate reading, writing, and searching the TLBs. The MAS registers can be read or written using the **mfspr** and **mtspr** instructions. The core complex does not implement the MAS5 register, because the **tlbsx** instruction on the e500 only searches based on a single PID value (the value of MAS6[SPID0]).

For the core complex, TLB0 is 2 (e500v1) or 4 (e500v2)-way set associative, so bits 45–51 of the effective address are used to index into TLB0 when it is accessed. For TLB0, ESEL is defined as a 2-bit field (bits 46–47) that identifies which of the indexed entries is to be referenced by the TLB operation (ESEL selects the way). For TLB1, ESEL selects one of the 16 entries in the array.

Figure 12-11 describes the format of MAS0 on the e500 core complex.

SPR 624 Access: Supervisor-only

	32–34	35	36–43	44–47	48–61	62–63
R/W	—	TLBSEL	—	ESEL	—	NV
Reset	All zeros					

Figure 12-11. MAS Register 0 (MAS0)

Table 12-8 shows the core complex MAS0 bit definitions.

Table 12-8. MAS0 Field Descriptions—MMU Read/Write and Replacement Control

Bits	Name	Descriptions
32–34	—	Reserved, should be cleared.
35	TLBSEL	Selects TLB for access 0 TLB0 1 TLB1
36–43	—	Reserved, should be cleared.

Table 12-8. MAS0 Field Descriptions—MMU Read/Write and Replacement Control

Bits	Name	Descriptions
44–47	ESEL	Entry select. Number of the entry in the selected array to be used for **tlbwe**. This field is also updated on TLB error exceptions (misses), and **tlbsx** hit and miss cases as shown in Table 12-15. For the e500, ESEL serves as the way select for the corresponding TLB as follows: When TLBSEL = 00 (TLB0 selected), bits 46–47 are used (and bits 44–45 should be cleared). This field selects between way 0, 1, 2, or 3 of TLB0. EA bits 45–51 from MAS2[EPN] are used to index into the TLB to further select the entry for the operation. Note that for the e500v1, bit 47 selects either way 0 or way 1, and bit 46 should remain cleared. When TLBSEL = 01 (TLB1 selected), all four bits are used to select one of 16 entries in the array.
48–61	—	Reserved, should be cleared.
62–63	NV	Next victim. 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 as shown in Table 12-15, 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 12.3.2.2, "Replacement Algorithms for L2 MMU"). Note that for the e500v1, bit 62 should remain cleared and only bit 63 has significance. Note that this field is not defined for operations that specify TLB1 (when TLBSEL = 01).

Figure 12-12 describes the format of MAS1 on the e500 core complex. Note that while Freescale Book E allows for a TID field of 12 bits, the TID field on the core complex is implemented as only 8 bits.

SPR 625 Access: Supervisor-only

	32	33	34–39	40–47	48–50	51	52–55	56–63
R/W	V	IPROT	—	TID	—	TS	TSIZE	—
Reset	All zeros							

Figure 12-12. MAS Register 1 (MAS1)

Table 12-9 shows the core complex MAS1 bit definitions.

Table 12-9. 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 due to the execution of **tlbiva**[**x**] (TLB1 only). 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. See Section 12.3.2.1, "IPROT Invalidation Protection in TLB1."
34–39	—	Reserved, should be cleared.
40–47	TID	Translation identity. An 8-bit field that defines the process ID for this TLB entry. TID is compared with the current process IDs of the three virtual address to be translated. A TID value of 0 defines an entry as global and matches with all process IDs.

Table 12-9. MAS1 Field Descriptions—Descriptor Context and Configuration Control (continued)

Bits	Name	Descriptions
48–50	—	Reserved, should be cleared.
51	TS	Translation space. This bit is compared with the IS or DS fields of the MSR (depending on the type of access) to determine if this TLB entry may be used for translation.
52–55	TSIZE	Translation size. Defines the page size of the TLB entry. For TLB arrays that contain fixed-size TLB entries, this field is ignored. For variable page size TLB arrays, the page size is 4^{TSIZE} Kbytes. Note that although the Freescale Book E standard supports all 16 page sizes defined in Book E, the e500 only supports the following page sizes: 0001 4 Kbyte 0010 16 Kbyte 0011 64 Kbyte 0100 256 Kbyte 0101 1 Mbyte 0110 4 Mbyte 0111 16 Mbyte 1000 64 Mbyte 1001 256 Mbyte 1010 1 Gbyte 1011 4 Gbyte
56–63	—	Reserved, should be cleared.

Figure 12-13 describes the format of MAS2 on the e500 core complex.

SPR 626 Access: Supervisor-only

	32–51	52–56	57	58	59	60	61	62	63
R/W	EPN	—	X0	X1	W	I	M	G	E
Reset	All zeros								

Figure 12-13. MAS Register 2 (MAS2)

Table 12-10 shows the core complex MAS2 bit definitions.

Table 12-10. MAS2 Field Descriptions—EPN and Page Attributes

Bits	Name	Description
32–51	EPN	Effective page number. Depending on page size, only the bits associated with a page boundary are valid. Bits that represent offsets within a page are ignored and should be cleared.
52–56	—	Reserved, should be cleared.
57	X0	Implementation-dependent page attribute
58	X1	Implementation-dependent page attribute
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.

Table 12-10. MAS2 Field Descriptions—EPN and Page Attributes (continued)

Bits	Name	Description
61	M	Memory coherence required 0 Memory coherence is not required. 1 Memory coherence 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 coherence 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 that miss in the L1 cache are performed without speculation (that is, they are known to be required). Speculative loads can be performed if they hit in the L1 cache. In addition, accesses to caching-inhibited pages are performed using only the memory element that is explicitly specified.
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 optionally available in previous devices that implement 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.

Figure 12-14 describes the format of MAS3. The core complex uses the same bit definitions as the Freescale Book E standard for MAS3 for 32-bit implementations.

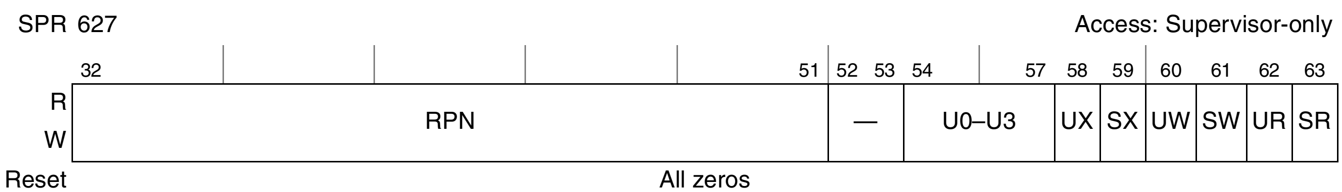


Figure 12-14. MAS Register 3 (MAS3)

Table 12-11 shows the core complex MAS3 bit definitions.

Table 12-11. 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 cleared. Note that, on the e500v2, additional bits of the RPN are contained in MAS7. See Section 12.7.1.1, "MAS Register 7 (MAS7)," for more information.
52–53	—	Reserved, should be cleared.
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	PERMIS	Permission bits (UX, SX, UW, SW, UR, SR). User and supervisor read, write, and execute permission bits. See the EREF:. for more information on the page permission bits as they are defined by Book E.

Figure 12-15 describes the format of MAS4 on the e500 core complex.

SPR 628 Access: Supervisor-only

	32 33	34 35	36 45	46 47	48	55	56	57	58	59	60	61	62	63
R W	—	TLBSELD	—	TIDSELD	—	TSIZED	—	X0D	X1D	WD	ID	MD	GD	ED
Reset	All zeros													

Figure 12-15. MAS Register 4 (MAS4)

Table 12-12 shows the core complex MAS4 bit definitions.

Table 12-12. MAS4 Field Descriptions—Hardware Replacement Assist Configuration

Bits	Name	Description
32–34	—	Reserved, should be cleared.
35	TLBSELD	TLBSEL default value. The default value to be loaded in MAS0[TLBSEL] on a TLB miss exception. See the EREF for more information. 0 TLB0 1 TLB1
36–45	—	Reserved, should be cleared.
46–47	TIDSELD	TID default selection value. A 2-bit field that specifies which of the current PID registers should be used to load the MAS1[TID] field on a TLB miss exception. The e500 implementation defines this field as follows: 00 PID0 01 PID1 10 PID2 11 TIDZ (0x00) (all zeros)
48–51	—	Reserved, should be cleared.
52–55	TSIZED	Default TSIZE value. Specifies the default value to be loaded into MAS1[TSIZE] on a TLB miss exception.
56	—	Reserved, should be cleared.
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.
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.

Note that MAS5 is not implemented in the e500 core complex.

Figure 12-16 shows the format of MAS6.

SPR 630 Access: Supervisor-only

	32–39	40–47	48–62	63
R/W	—	SPID0	—	SAS
Reset	All zeros			

Figure 12-16. MAS Register 6 (MAS6)

Table 12-13 shows the core complex MAS6 bit definitions. Note that while the Freescale Book E allows for a SPIDx field of 12 bits, SPID0 on the core complex is only an 8-bit field.

Table 12-13. MAS6—TLB Search Context Register 0

Bits	Name	Comments, or Function when Set
32–39	—	Reserved, should be cleared.
40–47	SPID0	Specifies the PID value (recent value of PID0) used when searching the TLB during execution of **tlbsx**.
48–62	—	Reserved, should be cleared.
63	SAS	Address space (AS) value for searches. Specifies the value of AS used when searching the TLB (during execution of **tlbsx**).

12.7.1.1 MAS Register 7 (MAS7)

The MAS7 register contains the high-order address bits of the RPN for implementations that support more than 32 bits of physical address. (It contains 4 bits in the case of the e500v2.) Implementations that do not support more than 32 bits of physical addressing do not implement MAS7. Note that MAS7 can be automatically updated as a result of execution of **tlbre** and **tlbsx** instructions (as is MAS3); this functionality is controlled by HID0[EN_MAS7_UPDATE].

Figure 12-17 shows the format of the MAS7 register.

SPR 944 Access: Supervisor-only

	32–59	60–63
R/W	—	RPN
Reset	All zeros	

Figure 12-17. MAS Register 7 (MAS7)

The MAS7 fields are described in Table 12-14.

Table 12-14. MAS7 Field Descriptions—High Order RPN

Bits	Name	Description
32–59	—	Reserved, should be cleared.
60–63	RPN	Real page number, 4 high-order bits. MAS3 holds only RPN[4–23]. The byte offset within the page is provided by the EA and is not present in MAS3 or MAS7.

12.7.2 MAS Register Updates

Table 12-15 summarizes the updates to each MAS register field for each update stimulus.

Table 12-15. MMU Assist Register Field Updates

MAS Register Bit/Field	Value Loaded for Each Case						
	Instr/Data TLB Error	**tlbsx Hit**	**tlbsx Miss**	**ISI**	**DSI**	**tlbre**	**tlbwe**
TLBSEL	TLBSELD	Which TLB hit	TLBSELD	—	—	—	—
ESEL	if TLBSELD = 0: TLB0[NV] else, undefined	Number of entry that hit	if TLBSELD = 0: TLB0[NV] else, undefined	—	—	—	—
NV	if TLBSELD = 0: ~TLB0[NV] else, undefined	if TLBSEL = 0: TLB0[NV] else, undefined	if TLBSELD = 0: ~TLB0[NV] else, undefined	—	—	if TLBSEL = 0: TLB0[NV] else, undefined	—
V	1	1	0	—	—	V(array)	—
IPROT	0	Matched IPROT if TLB1 hit; else 0	0	—	—	IPROT (array) if TLB1; else 0	—
TID[0–7]	Value of PID register selected by TIDSELD	TID (array)	SPID0	—	—	TID (array)	—
TS	MSR[IS/DS]	SAS	SAS	—	—	TS(array)	—
TSIZE[0–3]	TSIZED	TSIZE(array)	TSIZED	—	—	TSIZE(array)	—
EPN[32–51]	EPN of access	EPN (array)	—	—	—	EPN (array)	—
X0, X1 WIMGE	X0D, X1D WIMGED	X0, X1 (array) WIMGE (array)	X0D, X1D WIMGED	—	—	X0, X1 (array) WIMGE (array)	—
RPN[28–51]	Zeros	RPN (array)	Zeros	—	—	RPN (array)	—
Access (PERMIS + U0–U3)	Zeros	Access (array)	Zeros	—	—	Access (array)	—
TLBSELD	—	—	—	—	—	—	—
TIDSELD[0–1]	—	—	—	—	—	—	—
TSIZED[0–3]	—	—	—	—	—	—	—
WIMGED	—	—	—	—	—	—	—
SPID0	PID0	—	—	—	—	—	—
SAS	MSR[IS] for instruction access; MSR[DS] for data access	—	—	—	—	—	—

Chapter 13
Core Complex Bus (CCB)

This chapter provides a very general description of the core complex bus (CCB), which is the interface between the core and the integrating device. Because most of the behavior of the CCB is not directly programmable, or even visible, to the user, this chapter does not attempt to describe all aspects of the CCB or even the most important CCB signals.

Instead it describes only those aspects of the CCB that are configurable or that provide status information through the programming interface. It provides a glossary of those signals that are mentioned in other chapters to offer a clearer understanding of how the core is integrated as part of a larger device.

13.1 Overview

The CCB is the internal interface of the core complex and is derived from the 60x bus. The CCB allows a wide range of system-performance and system-complexity trade-offs, which are largely configured by the device that integrates the core. The CCB is defined as follows:

- High-speed, on-chip local bus interface
- 32-bit address bus
- Address protocol with address pipelining and retry/copyback derived from bus used by previous generations of PowerPC processors (referred to as the 60x bus)
- An address-out bus for mastering bus transactions
- An address-in bus for snooping internal resources
- Three tagged data buses

Two of the data buses are general-purpose data-in buses for reads, and the third is a data-out bus for writes. The two data-in buses feature support for out-of-order read transactions from two different sources simultaneously, and all three data buses may be operated concurrently. The address-in bus supports snooping for external management of the L1 caches and TLBs by other bus masters. The core complex broadcasts and snoops the cache and TLB management instructions accordingly. It is envisioned that a wide range of system implementations can be constructed from the defined interface.

The CCB derivation starts with the 60x bus, separates the bidirectional pins into unidirectional components (for system-on-chip use), and adds new attributes and capabilities to enhance data flow implementation or parallelism in certain system configurations. Note that this chapter does not attempt to characterize 60x bus behavior. Figure 13-1 shows the subset of CCB signals that are discussed in this document.

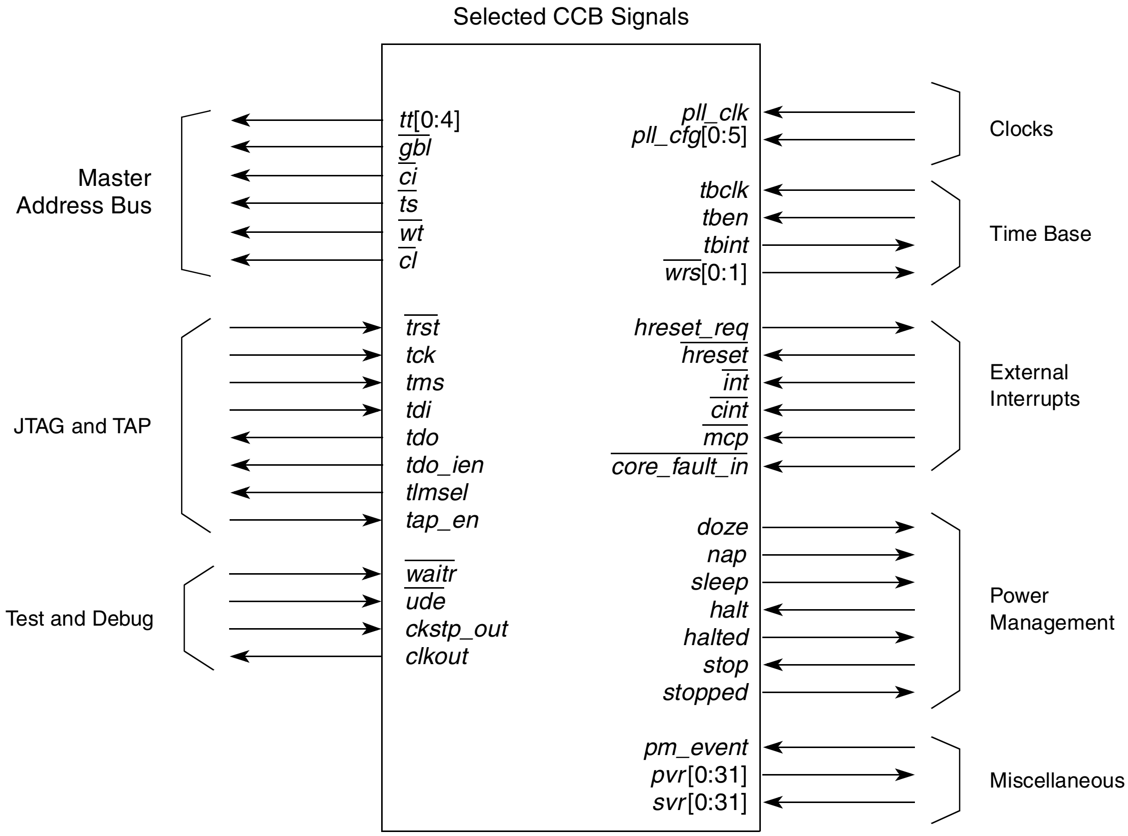


Figure 13-1. CCB Interface Signals

13.2 Signal Summary

Table 13-1 briefly describes selected internal signals of the CCB.

Table 13-1. Summary of Selected Internal Signals

Signal	I/O	Comments, or Meaning when Asserted
Bus Signals: Master Address Bus		
$\overline{ci}$	O	Cache inhibit. Normally reflected from the I bit of the WIMGE bits (regardless of whether the cache is enabled) For burst writes and address-only transactions, $\overline{ci}$ is always negated.
$\overline{cl}$	O	Cache lock. Indicates L2 (level 2) cache lock status for the transaction; also asserted during a burst write for **dcbf**

Table 13-1. Summary of Selected Internal Signals (continued)

Signal	I/O	Comments, or Meaning when Asserted
$\overline{gbl}$	O	Global. Normally reflected from the M bit of the WIMGE bits; asserted indicates transaction is enabled for snooping by other masters. • For burst writes, always negated • For lock-clear instructions to an L2 cache, always negated • For address-only transactions that bypass translation, always asserted
$\overline{ts}$	O	Transfer start. Asserted by the core to indicate a valid address with attributes.
tt[0:4]	O	Transfer type. Indicates the type of transaction (such as RWITM, WR w/Kill).
$\overline{wt}$	O	Write through. Used as a general-purpose information bit for the transaction. • For *tt*[0:4] = READ, 1 indicates instruction-side fetch; 0 indicates data-side read. • For *tt*[0:4] = RWITM/RCLAIM, 1 indicates intent-to-modify at the L1 level. • For single-beat writes, reflected from the EIMGE bits for that page • For burst writes, 0 indicates a push for **dcbf**/**dcbst** or for snoop. • For address-only transactions, always negated
Bus Signals: Snoop Address Bus		
$\overline{sgbl}$	I	Snoop global. Indicates the transaction is enabled for cache snooping. (Reservation-only snooping also occurs for non-global write transactions.)
$\overline{sts}$	I	Snoop transfer start. Asserted to indicate that the core complex should snoop the transaction this cycle
Bus Signals: Read-1 Data Bus (Read-2 Data Bus is Analogous)		
Test and Debug		
clkout	O	Clock out mux. Selects the appropriate e500 clock. Refer to Chapter 8, "Debug Support."
ckstp_out	O	Checkstop interrupt. Assertion of this signal by the e500 core is used by system to generate a chip-wide hard stop and to signal an external CKSTP_OUT.
$\overline{ude}$	I	Unconditional debug event interrupt. Asserting $\overline{ude}$ sets DBSR[UDE] and, if MSR[DE] is set, causes a debug interrupt to be taken. Several bits in the debug control registers can be used to override this behavior. See Section 2.13.1, "Debug Control Registers (DBCR0–DBCR2)," for more information. Provides extra COP functions when enabled by COP control bits.
$\overline{waitr}$	I	WAITR select. Assertion results in global $\overline{waitr}$ to be selected for the e500 core.
JTAG and TAP		
$\overline{trst}$	I	JTAG test reset. Asserted—This input causes asynchronous initialization of the internal JTAG test access port controller. Note that this signal must be asserted during the assertion of $\overline{hreset}$ to properly initialize the JTAG test access port.
tck	I	JTAG test clock. Driven by a free-running clock signal. Input signals to the test access port are sampled on the rising edge of *tck*. TAP output signal changes occur on the falling edge of *tck*. The test logic allows TCK to be stopped. asynchronously with respect to all other core complex clocks.
tms	I	JTAG test mode select. Decoded by the internal JTAG TAP controller to determine the primary operation of the test support circuitry
tdi	I	JTAG test data input. The value present on the rising edge of *tck* is loaded into the selected JTAG test instruction or data register.

Table 13-1. Summary of Selected Internal Signals (continued)

Signal	I/O	Comments, or Meaning when Asserted
tdo	O	JTAG test data output. The contents of the selected internal instruction or data register are shifted out onto this signal on the falling edge of *tck*.
tdo_ien	O	Test data out enable. *tdo* provides feedback to the external TAP linking module logic.
tlmsel	O	TLM selected. *tlmsel* provides feedback to the external TAP linking module logic.
tap_en	I	TAP enable. *tap_en* is used by the TAP linking module (TLM) logic external to the core complex.
Clocks		
pll_cfg[0:5]	I	PLL configuration select. Configurations are as follows: 00000_x PLL off 00001_0 or 00001_1 PLL 1x or 1.5x 00010_0 or 00010_1 PLL 2x or 2.5x 00011_0 or 00011_1 PLL 3x or 3.5x 00100_0 or 00100_1 PLL 4x or 4.5x ... similar pattern up to 24x for even multipliers, or 12.5x for odd multipliers.
pll_clk	I	PLL clock. Clock reference for the CCB.
Time Base		
tbclk	I	Sampled by the system logic to CCB clock. Required to be no more than 1/4 platform clock frequency. If selected, it can be a source of the time base.
tben	I	Asserted by the system logic to enable the time base
tbint	O	Asserted when a time base interrupt is signaled. This ordinarily prompts external logic to bring the core out of power-down mode by negating *stop* and then *halt* so the interrupt can be serviced.
$\overline{wrs}$[0:1]	O	Watchdog timer reset status. These two bits are set to one of three values when a reset is caused by the watchdog timer. These bits are undefined at power-up. 00 Implementation-dependent reset information. 01 Implementation-dependent reset information. 10 Implementation-dependent reset information. 11 Idle
External Interrupts		
hreset_req	O	Hard reset request. When DBCRO[RST] is set, the core sends an HRESET_REQ to the system. The system recognizes the assertion of this request and then stops the core using power management. With hreset_req being asserted and the core being in STOPPED state, $\overline{hreset}$ is asserted and core flushing starts.
$\overline{hreset}$	I	Hard reset. Assertion flushes the core. When $\overline{hreset}$ is negated, the 256 CCB clocks core flush starts.
$\overline{int}$	I	External interrupt. Initiates an external interrupt. If $\overline{int}$ is asserted and MSR[EE] is set, the e500 vectors to IVOR4.
$\overline{cint}$	I	Critical interrupt. Initiates a critical interrupt. If $\overline{cint}$ is asserted and MSR[CE] is set, the e500 vectors to IVOR0. If MSR[CE] is 0, critical interrupts are disabled and the e500 does not sample $\overline{cint}$.
$\overline{mcp}$	I	Machine check interrupt. Initiates a machine check operation. If MSR[ME] is set, the e500 vectors to IVOR1. If MSR[ME] is clear, then the e500 goes into checkstop state. MCSR is updated as defined in Section 2.7.2.4, “Machine Check Syndrome Register (MCSR).”

Table 13-1. Summary of Selected Internal Signals (continued)

Signal	I/O	Comments, or Meaning when Asserted
core_fault_in	I	Core bus fault input. When asserted, signals a bus fault. On the e500v2, prevents the core transaction from completing, protecting the code from executing with potentially bad data. Thus, the transaction stalls waiting for an interrupt. If HID1[RFXE] = 1 and MSR[ME] = 1, assertion of *core_fault_in* causes a machine check interrupt and if HID1[RFXE] = 1 and MSR[ME] = 0, it causes a checkstop. For more information about bus faults, see Section 13.8, "Proper Reporting of Bus Faults." For proper handling of bus faults, see Section 2.10.2, "Hardware Implementation-Dependent Register 1 (HID1)."
Power Management Signals for the Core Complex		
halt	I	Asserted by system logic to request the core complex to go into halted state. Negating *halt* causes the core complex to transition back into the full-on state. Once asserted, *halt* must not be negated until after the core complex has entered halted state (otherwise the negation may not be recognized).
stop	I	Asserted by system logic to request that the core complex go from the halted state into the power-down state. Negating this signal causes the core complex to transition back into the halted state. Once asserted, *stop* must not be negated until after the core complex has entered the stopped state (otherwise the negation may not be recognized). For power management purposes, *stop* must be asserted only while the core complex is in halted state.
halted	O	Asserted when the core complex is in the halted state. It is the indication that it is safe for e500 core to go into the power-down state.
stopped	O	Asserted any time the internal functional clocks of the core complex are stopped.
doze	O	Reflect the state of the corresponding HID0 DOZE, NAP, and SLEEP bits, further qualified with MSR[WE] = 1 (both must be 1 for the respective output to be asserted). The state of these signals has no effect on the power-down state of the core complex. They serve only as indicators to external logic of power management requests by software.
nap	O	
sleep	O	
Miscellaneous Signals		
pm_event	I	External event. A level-sensitive input to e500 performance monitor to count external events.
pvr[0:31]	O	Processor version. The processor version information is provided for reading through a system SPR. Static signals during functional mode.
svr[0:31]	I	System version. The system version information is directly readable through an SPR in the core complex. Static signals during functional mode.

13.3 Core Interface Behavior

This section describes the behavior of the core interface with respect to parity and the synchronizing instructions, **mbar** and **msync**.

13.3.1 Parity Specification

The CCB supports byte parity (odd parity) on each data bus. Parity checking for the read data buses is enabled by setting HID1[R1DPE,R2DPE].

For write transactions, the core complex always supplies correct data parity across all byte lanes of the write data bus. If an internal parity error is detected in the L1 data cache during a castout (burst write) operation from the core complex MCSR[DCP_PERR] is set. A front-side L2 does not cache the bad data. The system has enough information to prevent memory corruption.

The address attribute signal, $\overline{wt}$, is also asserted during the address tenure for that transaction. By setting L1CSR0[CPE], the core complex may be configured to also take a data cache parity error exception.

Parity error handling is described in Section 5.7.2, “Machine Check Interrupt.”

13.3.2 msync Operation and the Bus

The **msync** instruction provides a synchronization boundary for instruction execution. Its architectural intent is to guarantee that the effects of all instructions prior to the **msync** instruction have occurred before any subsequent instructions begin execution. It may be used, for example, to ensure that a control bit has finally been written to its destination control register in the system before the next instruction begins execution (such as to clear a pending interrupt). By its nature, it also provides an ordering boundary for pre- and post-**msync** memory transactions.

For the core complex, an **msync** does not finish execution until all memory transactions caused by prior instructions complete entirely in its caches and externally on the bus (address and data transactions complete, excluding instruction fetches). No subsequent instructions and associated memory transactions are initiated until such completion occurs. Execution of **msync** also generates a SYNC command on the bus (if HID1[ABE] is set through the *tt*[0:4] signals), which also must complete normally (without address retry) for the **msync** instruction to complete.

13.3.3 mbar Operation and the Bus

The **mbar** instruction provides an ordering boundary for memory operations. Its architectural intent is to guarantee that memory operations resulting from instructions prior to the **mbar** instruction occur before any subsequent memory operations occur (thereby ensuring an order between pre- and post-**mbar** memory operations). It may be used, for example, to ensure that reads and writes to an I/O device or between I/O devices occur in program order, or to ensure that memory updates occur before a semaphore is released.

The Book E architecture allows an implementation to support several classes of memory ordering, selected by the MO field of the **mbar** instruction. The core complex supports two classes for system flexibility. For MO ≥ 0, the core complex re-interprets and executes **mbar** as an **msync**, which by its nature guarantees an order between all pre- and post-**mbar** memory transactions.

For MO = 1, the core complex executes the **mbar** instruction as a pipelined or flowing ordering barrier for potentially higher performance. For this case, an ordering barrier is established by the **mbar** instruction and flows along with the pre- and post-**mbar** memory transactions through the memory hierarchy (L1 cache, bus, and system). On the bus, this ordering barrier is issued as an ORDER command (if HID1[ABE] is set through *tt*[0:4]).

The system ensures that the ordering barrier established by the ORDER command between any pre- and post-**mbar** bus transactions (excluding instruction fetches) is honored in any system queues and out to the transactions' destinations. If transaction ordering does not occur naturally or is not easily controlled in the system, a simple method could be to not complete the ORDER command on the bus (similar to the SYNC command) until all prior bus transactions have completed or to withhold bus grant for any further transactions until such completion.

13.4 Address Streaming Mode

Address streaming mode (selected by setting HID1[ASTE]) provides a way to increase address bus throughput on the CCB. Address streaming is useful for systems that must normally extend the address tenure by delaying address acknowledge after transfer start, thereby reducing bus transactions during a given period, as in the following examples:

- A system where addresses cannot be decoded or accepted immediately after transfer start by the system
- A snooping system where address acknowledge must be delayed to allow snooping caches (including the L1 caches of the core complex in certain clock modes) to process a snoop transaction

Note that address streaming, as defined here, differs from address pipelining, which is the issue of multiple address tenures independent of whether associated data tenures were started or completed.

Address streaming allows one additional bus transaction from the same bus master to start on the address bus during a current address tenure. This mode effectively overlaps and staggers two address tenures from the same bus master at any given time. It also effectively pipelines address tenures with respect to the address acknowledge/retry window.

13.5 L2 Cache Support

The e500 implements specific instructions to selectively lock and unlock lines in its L1 caches or in an L2 cache. To facilitate locking and unlocking of a front-side L2 cache (usually located directly on the CCB), the core complex provides an address lock attribute (CL) on the bus, which can be used in conjunction with the internal transfer type, *tt*[0:4], encodings to identify which addresses to lock or unlock.

13.5.1 L2 Locking

When the core complex executes an instruction to lock a line in an L2 cache (**dcbtls**, **dcbtstls**, or **icbtls**, with CT = 1), it performs the associated bus operation as a burst read transaction with the lock attribute asserted. A front-side L2 cache may recognize this transaction as a direction to establish the cache line (if not already valid) and to mark it as locked. Note that this is a complete address/data transaction by the core complex to memory that requires read data to be returned to the core complex. The read data, however, is not used or cached internally by the core complex. The purpose for the bus transaction is to establish a locked line in the L2 cache and to make data available from system memory for the L2 cache to capture.

Cache locking instructions targeted at an L2 cache may also hit to modified data in the L1 data cache when they are executed. In this case, the core complex pushes the line from the L1 data cache as a non-global burst write operation (similar to a regular L1 castout) and with the lock attribute set and the write-through attribute negated, rather than performing a read bus operation as described above. A front-side L2 cache may also recognize this transaction as a direction to establish and capture the cache line and mark it as locked.

13.5.2 L2 Unlocking

When the core complex executes an instruction (**dcblc**, **icblc**) to unlock an L2 cache line, it performs the associated bus operation as an address-only transaction with a *tt*[0:4] encoding of CLEAN and with the lock attribute asserted. A front-side L2 cache may recognize this transaction as a direction to unlock the specified address from its cache. This transaction is always performed as non-global because it is specifically targeted at an L2 cache.

An L2 cache may also use other bus transactions to cause locks to be cleared, such as bus transactions as a result of **dcbf** (identified on the bus as an address-only FLUSH) or as an L1 push due to **dcbf**.

13.5.3 L1 Overlock

A program can attempt to over-lock the core complex's L1 data cache by trying to establish a ninth locked entry at a cache index that already has all of its 8 ways locked. In this case, the core complex performs a reading transaction on the bus to initially bring in the ninth (newest) line and then immediately pushes that line out to the bus as a nonglobal burst write with the lock attribute asserted, rather than attempting to allocate that line in the L1 data cache. This write operation looks identical on the bus as the one described in Section 13.5.1, "L2 Locking," for hit-to-modified cases.

13.6 Reservation Management

The core complex supports standard reservation management through the **lwarx**/**stwcx.** instruction pair. This method of reservation management relies exclusively on bus snooping to detect whether an atomic access to a reservation granule was successful.

For systems that require the implementation of atomic accesses without a requirement for bus snooping, a following option is recommended. A system-defined atomic operation could be implemented directly in the memory subsystem and keyed off of a unique bus transaction (such as by *tt*[0:4] code and/or address decoding). By implementing such an operation directly in the memory system, a system may avoid the problems of having to lock multiple bus transactions by a processor throughout the system hierarchy, such as is typically done with the traditional LOCK pin of other bus protocols.

An example of a system-defined atomic operation that could be implemented directly in the memory system is an atomic set. For this operation, the memory system recognizes a unique read transaction on the bus, returns the read data from the specified field in memory, and then atomically writes the specified field to all ones. The field in memory might represent a high-true semaphore flag to indicate that a resource has been claimed. The atomic-set operation (as well as atomic-clear, atomic-increment, and atomic-decrement) is also defined for the RapidIO bus protocol.

The triggering of such an atomic transaction could be done, for instance, by the READ-atomic *tt*[0:4] code for a non-burst read, which occurs exclusively by the core complex for a cache-inhibited **lwarx**, or it could be triggered by simple address decoding or other mechanisms. Note that use of cache-inhibited **lwarx** would allow mixing of regular reads with atomic reads in a memory system for robustness; however, because it is not compatible with the usual **lwarx**/**stwcx.** behavior defined by the PowerPC architecture, such use would have to be carefully controlled by the system.

13.7 Remote Atomic Status Monitoring

For system convenience, the core complex provides a system-defined atomic status bit HID1[ATS] that a system may use for remote reservation management. If supported by the system, this bit could be monitored by a program internally until an atomic location in the memory system has been altered or cleared, thereby eliminating the bus bandwidth typically consumed by spinning on the bus waiting for the release of a semaphore as in traditional systems. This bit is automatically set whenever the core complex performs a **lwarx**(CI) transaction on the CCB. The memory system can clear this bit by asserting the atomic status clear (ATSC) input to the CCB according to a system-defined event. Such an event could be a write to a page of semaphore bits, indicating that a semaphore in the system has been released and that each processor may then attempt to claim a semaphore it is targeting.

13.8 Proper Reporting of Bus Faults

Except for one case in the e500v1 (described in the HID1[RFXE] bit description of Section 2.10.2, "Hardware Implementation-Dependent Register 1 (HID1)), the following applies for bus faults in the e500 core. When a bus fault is detected on a CCB transaction through the assertion of *core_fault_in* (and HID1[RFXE] = 0), the transaction stalls (to protect the register file and to avoid executing bad instructions), and does not complete until it receives an interrupt signalled by a peripheral block through the assertion of $\overline{int}$ or $\overline{cint}$, for example. This interrupt signalling typically occurs though an interrupt controller that is reporting enabled interrupts from either the peripheral block that detected the bus fault or from a watchdog timer.

Therefore, to ensure forward progress during normal operation, peripheral error-reporting logic must be configured to signal an interrupt (such as *int* or *cint*) for all possible sources of *core_fault_in*. Otherwise, the core stalls indefinitely on a bus fault, waiting for an interrupt.

However, during software or firmware development, when peripheral error-reporting may not yet be properly configured, the core can be configured (by setting HID1[RFXE]) to generate a machine check (or checkstop) on every assertion of *core_fault_in*. This forces bus faulted transactions to complete and allows processing to continue, even though little bus fault-specific information is saved that indicates the cause of the machine check. This is the only instance where RFXE should be set (except for the case for the e500v1, described in the HID1[RFXE] bit description of Section 2.10.2, “Hardware Implementation-Dependent Register 1 (HID1)).

Care must be taken if HID1[RFXE] is set = 1 during debug and some sources of *core_fault_in* are configured to signal an interrupt to the core (through *int* or *cint*), because in this case, two interrupts (machine check and external) could be reported on a bus fault, but the less-specific machine check interrupt enabled by RFXE = 1 (and MSR[ME] = 1) may occur first, giving little information about the cause of the fault.

Therefore, for normal operation, RFXE should always be cleared so that bus faults associated with peripheral devices do not generate a machine check interrupt or checkstop, but generate only the more useful interrupt provided by the peripheral. Thus, peripheral error reporting for all possible causes of *core_fault_in* should always be enabled for normal operation.

See Section 11.3.4.5, “Speculative Accesses to Guarded Memory,” for a cautionary statement regarding memory areas that are set up as both cacheable and guarded.

Appendix A
Programming Examples

This appendix gives examples of how memory synchronization instructions can be used to emulate various synchronization primitives and to provide more complex forms of synchronization. It also describes multiple-precision shifts.

A.1 Synchronization

Examples in this appendix have a common form. After possible initialization, a conditional sequence begins with a load and reserve instruction that may be followed by memory accesses and computations that include neither a load and reserve nor a store conditional. The sequence ends with a store conditional with the same target address as the initial load and reserve. In most of the examples, failure of the store conditional causes a branch back to the load and reserve for a repeated attempt. On the assumption that contention is low, the conditional branch in the examples is optimized for the case in which the store conditional succeeds, by setting the branch-prediction bit appropriately. These examples focus on techniques for the correct modification of shared storage locations: see note 4 in Section A.1.3.1, “Notes,” for a discussion of how the retry strategy can affect performance.

Load and reserve and store conditional instructions depend on the coherence mechanism of the system. Stores to a given location are coherent if they are serialized in some order, and no processor is able to observe a subset of those stores as occurring in a conflicting order. The “Memory and Cache Background” chapter of the EREF provides details about memory access ordering.

Each load operation, whether ordinary or load and reserve, returns a value that has a well-defined source. The source can be the store or store conditional instruction that wrote the value, an operation by some other mechanism that accesses storage (for example, an I/O device), or the initial state of storage.

The function of an atomic read/modify/write operation is to read a location and write its next value, possibly as a function of its current value, all as a single atomic operation. We assume that locations accessed by read/modify/write operations are accessed coherently, so the concept of a value being the next in the sequence of values for a location is well defined. The conditional sequence, as defined above, provides the effect of an atomic read/modify/write operation, but not with a single atomic instruction. Let *addr* be the location that is the common target of the load and reserve and store conditional instructions. Then the guarantee the architecture makes for the successful execution of the conditional sequence is that no store into *addr* by another processor or mechanism has intervened between the source of the load and reserve and the store conditional.

For each of these examples, it is assumed that a similar sequence of instructions is used by all processes requiring synchronization on the accessed data.

NOTE

Because memory synchronization instructions have implementation dependencies (for example, the granularity at which reservations are managed), they must be used with care. The operating system should provide system library programs that use these instructions to implement the high-level synchronization functions (such as test and set or compare and swap) needed by application programs. Application programs should use these library programs, rather than use storage synchronization instructions directly.

A.1.1 Synchronization Primitives

The following examples show how the **lwarx** and **stwcx.** instructions can be used to implement various synchronization primitives.

The sequences used to emulate the various primitives consist primarily of a loop using **lwarx** and **stwcx.**. No additional synchronization is necessary, because the **stwcx.** will fail, clearing EQ, if the word loaded by **lwarx** has changed before the **stwcx.** is executed: see Section 3.3.1.7, "Atomic Update Primitives Using **lwarx** and **stwcx.**," for details.

A.1.1.1 Fetch and No-op

The fetch and no-op primitive atomically loads the current value in a word in storage.

In this example, it is assumed that the address of the word to be loaded is in GPR3 and the data loaded is returned in GPR4.

```
loop:   lwarx   r4,0,r3         #load and reserve
        stwcx.  r4,0,r3         #store old value if still reserved
        bc      4,2,loop        #loop if lost reservation
```

If the **stwcx.** succeeds, it stores to the target location the same value that was loaded by the preceding **lwarx**. While the store is redundant with respect to the value in the location, its success ensures that the value loaded by the **lwarx** was the current value, that is, that the source of the value loaded by the **lwarx** was the last store to the location that preceded the **stwcx.** in the coherence order for the location.

A.1.1.2 Fetch and Store

The fetch and store primitive atomically loads and replaces a word in storage. In this example it is assumed that the address of the word to be loaded and replaced is in GPR3, the new value is in GPR4, and the old value is returned in GPR5.

```
loop:   lwarx   r5,0,r3         #load and reserve
        stwcx.  r4,0,r3         #store new value if still reserved
        bc      4,2,loop        #loop if lost reservation
```

A.1.1.3 Fetch and Add

The fetch and add primitive atomically increments a word in storage. In this example it is assumed that the address of the word to be incremented is in GPR3, the increment is in GPR4, and the old value is returned in GPR5.

```
loop:   lwarx   r5,0,r3         #load and reserve
        add     r0,r4,r5        #increment word
        stwcx.  r0,0,r3         #store new value if still reserved
        bc      4,2,loop        #loop if lost reservation
```

A.1.1.4 Fetch and AND

The Fetch and AND primitive atomically ANDs a value into a word in storage.

In this example it is assumed that the address of the word to be ANDed is in GPR3, the value to AND into it is in GPR4, and the old value is returned in GPR5.

```
loop:   lwarx   r5,0,r3         #load and reserve
        and     r0,r4,r5        #AND word
        stwcx.  r0,0,r3         #store new value if still reserved
        bc      4,2,loop        #loop if lost reservation
```

This sequence can be changed to perform another Boolean operation atomically on a word in memory by changing the **and** to the desired Boolean instruction (**or**, **xor**, etc.).

A.1.1.5 Test and Set

This version of the test and set primitive atomically loads a word from memory, sets the word in memory to a nonzero value if the value loaded is zero, and sets the EQ bit of CR Field 0 to indicate whether the value loaded is zero.

In this example it is assumed that the address of the word to be tested is in GPR3, the new value (nonzero) is in GPR4, and the old value is returned in GPR5.

```
loop:   lwarx   r5,0,r3         #load and reserve
        cmpwi   r5,0            #done if word
        bc      4,2,done        # not equal to 0
        stwcx.  r4,0,r3         #try to store non-0
        bc      4,2,loop        #loop if lost reservation
done:
```

A.1.1.6 Compare and Swap

The compare and swap primitive atomically compares a value in a register with a word in memory, if they are equal stores the value from a second register into the word in memory, if they are unequal loads the word from memory into the first register, and sets CR0[EQ] to indicate the result of the comparison.

In this example it is assumed that the address of the word to be tested is in GPR3, the comparand is in GPR4 and the old value is returned there, and the new value is in GPR5.

```
loop:   lwarx   r6,0,r3         #load and reserve
        cmpw    r4,r6           #1st 2 operands equal?
        bc      4,2,exit        #skip if not
        stwcx.  r5,0,r3         #store new value if still reserved
        bc      4,2,loop        #loop if lost reservation
exit:   or      r4,r6,r6        #return value from memory
```

A.1.1.7 Notes

1. The semantics given for compare and swap above are based on those of the IBM System/370 compare and swap instruction. Other architectures may define a compare and swap instruction differently.
2. Compare and swap is shown primarily for pedagogical reasons. It is useful on machines that lack the better synchronization facilities provided by **lwarx** and **stwcx.**. A major weakness of a System/370-style compare and swap instruction is that, although the instruction itself is atomic, it checks only that the old and current values of the word being tested are equal, with the result that programs that use such a compare and swap to control a shared resource can err if the word has been modified and the old value subsequently restored. The sequence shown above has the same weakness.

3. In some applications the second **bc** and/or the **or** can be omitted. The **bc** is needed only if the application requires that if CR0[EQ] on exit indicates not equal then GPR4 and GPR6 are not equal. The **or** is needed only if the application requires that if the comparands are not equal then the word from memory is loaded into the register with which it was compared (rather than into a third register). If any of these instructions is omitted, the resulting compare and swap does not obey System/370 semantics.

A.1.2 Lock Acquisition and Release

This example gives an algorithm for locking that demonstrates the use of synchronization with an atomic read/modify/write operation. A shared memory location, the address of which is an argument of the lock and unlock procedures given by GPR3, is used as a lock, to control access to some shared resource such as a shared data structure. The lock is open when its value is 0 and closed (locked) when its value is 1. Before accessing the shared resource the program executes the lock procedure, which sets the lock by changing its value from 0 to 1. To do this, the lock procedure calls test_and_set, which executes the code sequence shown in the test and set example of Section A.1.1, "Synchronization Primitives," thereby atomically loading the old value of the lock, writing to the lock the new value (1) given in GPR4, returning the old value in GPR5 (not used below), and setting the EQ bit of CR Field 0 according to whether the value loaded is 0. The lock procedure repeats the test_and_set until it succeeds in changing the value of the lock from 0 to 1.

Because the shared resource must not be accessed until the lock has been set, the lock procedure contains an **isync** after the **bc** that checks for the success of test_and_set. The **isync** delays all subsequent instructions until all preceding instructions have completed.

```
lock:   mfspr   r6,LR           #save Link Register
        addi    r4,r0,1         #obtain lock:
loop:   bl      test_and_set    # test-and-set
        bc      4,2,loop        # retry till old = 0
# Delay subsequent instructions till prior instructions finish
        isync
        mtspr   LR,r6           #restore Link Register
        blr                     #return
```

The unlock procedure stores a 0 to the lock location. Most applications that use locking require, for correctness, that if the access to the shared resource includes stores, the program must execute an **msync** before releasing the lock. The **msync** ensures that the program's modifications are performed with respect to other processors before the store that releases the lock is performed with respect to those processors. In this example, the unlock procedure begins with an **msync** for this purpose.

```
unlock: msync                   #order prior stores
        addi    r1,r0,0         #before lock release
        stw     r1,0(r3)        #store 0 to lock location
        blr                     #return
```

A.1.3 List Insertion

This example shows how **lwarx** and **stwcx.** can be used to implement simple insertion into a singly linked list. (Complicated list insertion, in which multiple values must be changed atomically, or in which the correct order of insertion depends on the contents of the elements, cannot be implemented in the manner shown below and requires a more complicated strategy such as using locks.)

The next element pointer from the list element after which the new element is to be inserted, here called the parent element, is stored into the new element, so that the new element points to the next element in the list: this store is performed unconditionally. Then the address of the new element is conditionally stored into the parent element, thereby adding the new element to the list.

In this example it is assumed that the address of the parent element is in GPR3, the address of the new element is in GPR4, and the next element pointer is at offset 0 from the start of the element. It is also assumed that the next element pointer of each list element is in a reservation granule separate from that of the next element pointer of all other list elements. See Section 3.3.1.7, "Atomic Update Primitives Using **lwarx** and **stwcx.**"

```
loop:   lwarx   r2,0,r3         #get next pointer
        stw     r2,0(r4)        #store in new element
        msync                   #order stw before stwcx.(can omit if not MP)
        stwcx.  r4,0,r3         #add new element to list
        bc      4,2,loop        #loop if stwcx. failed
```

In the preceding example, if two list elements have next element pointers in the same reservation granule then, in a multiprocessor, livelock can occur. (Livelock is a state in which processors interact in a way such that no processor makes progress.)

If it is not possible to allocate list elements such that each element's next element pointer is in a different reservation granule, livelock can be avoided by using the following, more complicated, sequence.

```
        lwz     r2,0(r3)        #get next pointer
loop1:  or      r5,r2,r2        #keep a copy
        stw     r2,0(r4)        #store in new element
        msync                   #order stw before stwcx.
loop2:  lwarx   r2,0,r3         #get it again
        cmpw    r2,r5           #loop if changed (someone
        bc      4,2,loop1       # else progressed)
        stwcx.  r4,0,r3         #add new element to list
        bc      4,2,loop        #loop if failed
```

A.1.3.1 Notes

1. In general, **lwarx** and **stwcx.** should be paired, with the same effective address used for both. The only exception is that an unpaired **stwcx.** to any (scratch) effective address can be used to clear any reservation held by the processor.
2. It is acceptable to execute a **lwarx** for which no **stwcx.** is executed. For example, this occurs in the test and set sequence shown above if the value loaded is not zero.
3. To increase the likelihood that forward progress is made, it is important that looping on **lwarx**/**stwcx.** pairs be minimized. For example, in the sequence shown above for test and set, this is achieved by testing the old value before attempting the store: were the order reversed, more **stwcx.** instructions might be executed, and reservations might more often be lost between the **lwarx** and the **stwcx.**.
4. The manner in which **lwarx** and **stwcx.** are communicated to other processors and mechanisms, and between levels of the memory subsystem within a given processor is implementation dependent (see Section 3.3.1.7, “Atomic Update Primitives Using **lwarx** and **stwcx.**”). In some implementations performance may be improved by minimizing looping on a **lwarx** instruction that fails to return a desired value. For example, in the test and set example shown above, if the programmer wishes to stay in the loop until the word loaded is zero, he could change the bne- $+12 to bne- loop. However, in some implementations better performance may be obtained by using an ordinary load instruction to do the initial checking of the value, as follows.

```
loop:   lwz     r5,0(r3)        #load the word
        cmpi    cr0,0,r5,0      #loop back if word
        bc      4,2,loop        # not equal to 0
        lwarx   r5,0,r3         #try again, reserving
        cmpi    cr0,0,r5,0      # (likely to succeed)
        bc      4,2,loop
        stwcx.  r4,0,r3         #try to store non-0
        bc      4,2,loop        #loop if lost reservation
```

5. In a multiprocessor, livelock is possible if a loop containing a **lwarx**/**stwcx.** pair also contains an ordinary store instruction for which any byte of the affected memory area is in the reservation granule: see Section 3.3.1.7, “Atomic Update Primitives Using **lwarx** and **stwcx.**” For example, the first code sequence shown in Section A.1.3, “List Insertion,” can cause livelock if two list elements have next element pointers in the same reservation granule.

Appendix B
Guidelines for 32-Bit Book E

This appendix provides guidelines used by 32-bit Book E implementations. Likewise, a set of guidelines is also outlined for software developers. Application software written to these guidelines can be labelled 32-bit Book E applications and can expect to execute properly on all implementations of Book E, both 32-bit and 64-bit implementations.

32-bit Book E implementations execute applications that adhere to the software guidelines for 32-bit Book E software outlined in this appendix and are not expected to properly execute 64-bit Book E applications or any applications not adhering to these guidelines (that is, 64-bit Book E applications).

B.1 64-Bit–Specific Book E Instructions

A subset of Book E instructions are restricted to 64-bit Book E processing. A 32-bit Book E implementation need not implement any of the following instructions. Likewise, neither should 32-bit Book E applications use any of these instructions. All other Book E instructions are either supported directly by the implementation or sufficient infrastructure is provided to enable software emulation of the instructions.

The 64-bit Book E instructions are as follows:

- 64-bit integer arithmetic, compare, shift and rotate instructions
  - **adde64**[**o**], **addme64**[**o**], **addze64**[**o**]
  - **subfe64**[**o**], **subfme64**[**o**], **subfze64**[**o**]
  - **mulhd**, **mulhdu**, **mulld**[**o**], **divd**, **divdu**, **extsw**
  - **cmp** (L=1), **cmpi** (L=1), **cmpl** (L=1), **cmpli** (L=1)
  - **rldcl**, **rldcr**, **rldic**, **rldicl**, **rldicr**, **rldimi**, **sld**, **srad**, **sradi**, **srd**
  - **cntlzd**, **td**, **tdi**
- 64-bit extended addressing branch instructions—**bcctre**[**l**], **bce**[**l**][**a**], **bclre**[**l**], **be**[**l**][**a**]
- 64-bit extended addressing cache management instructions—**dcbae**, **dcbfe**, **dcbie**, **dcbste**, **dcbte**, **dcbtste**, **dcbze**, **icbie**, **icbte**

- 64-bit extended addressing load instructions—**lbze**, **lbzue**, **lbzxe**, **lbzxue**, **ldarxe**, **lde**, **ldue**, **ldxe**, **ldxue**, **lfde**, **lfdue**, **lfdxe**, **lfdxue**, **lfse**, **lfsue**, **lfsxe**, **lfsxue**, **lhae**, **lhaue**, **lhaxe**, **lhaxue**, **lhbrxe**, **lhze**, **lhzue**, **lhzxe**, **lhzxue**, **lwarxe**, **lwbrxe**, **lwze**, **lwzue**, **lwzxe**, **lwzxue**
- 64-bit extended addressing store instructions—**stbe**, **stbue**, **stbxe**, **stbxue**, **stdcxe.**, **stde**, **stdue**, **stdxe**, **stdxue**, **stfde**, **stfdue**, **stfdxe**, **stfdxue**, **stfiwxe**, **stfse**, **stfsue**, **stfsxe**, **stfsxue**, **sthbrxe**, **sthe**, **sthue**, **sthxe**, **sthxue**, **stwbrxe**, **stwcxe.**, **stwe**, **stwue**, **stwxe**, **stwxue**

B.2 Registers on 32-Bit Book E Implementations

Book E defines 32- and 64-bit registers. All 32-bit registers are supported as defined in Book E. However, only bits 32–63 of Book E's 64-bit registers are required to be implemented in hardware in 32-bit Book E implementation. Such 64-bit registers include LR, CTR, 32 GPRs, SRR0, and CSRR0. Book E makes no restrictions regarding implementing a subset of the 64-bit floating-point architecture.

Likewise, other than floating-point instructions, all instructions defined to return a 64-bit result return only bits 32–63 of the result on a 32-bit Book E implementation.

B.3 Addressing on 32-Bit Book E Implementations

Only bits 32–63 of the 64-bit Book E instruction and data memory effective addresses need to be calculated and presented to main memory. Given that only branch and data memory access instructions not included in Section B.1, "64-Bit–Specific Book E Instructions," are defined to prepend 32 zeros to bits 32–63 of the effective address computation, a 32-bit implementation can bypass the prepending of the 32 zeros when implementing these instructions. For branch to LR and branch to CR instructions, given that LR and CTR are implemented as 32-bit registers, concatenating only 2 zeros to the right of bits 32–61 of these registers is necessary to form the 32-bit branch target address.

The simplest implementation of next sequential instruction address computation suggests allowing effective address computations to wrap from 0xFFFF_FFFC to 0x0000_0000. This wrapping is required of PowerPC implementations. For 32-bit Book E applications, there appears little if any benefit to allowing this wrapping behavior. Book E specifies that the situation where the computation of the next sequential instruction address after address 0xFFFF_FFFC is undefined (note that the next sequential instruction address after address 0xFFFF_FFFC on a 64-bit Book E implementation is 0x0000_0001_0000_0000).

B.4 TLB Fields on 32-bit Book E Implementations

32-bit Book E implementations should support bits 32–53 of the effective page number (EPN) field in the TLB. This size provides support for a 32-bit effective address, which PowerPC ABIs may have come to expect to be available. 32-bit Book E implementations may support greater than

32-bit real addresses by supporting more than bits 32–53 of the real page number (RPN) field in the TLB.

B.5 32-Bit Book E Software Guidelines

This section describes instruction selection and addressing of 32-bit software.

B.5.1 32-Bit Instruction Selection

Any Book E software that uses any of the instructions listed in Section B.1, “64-Bit–Specific Book E Instructions,” is considered 64-bit Book E software, and correct execution cannot be guaranteed on 32-bit Book E implementations. Generally speaking, 32-bit software should avoid instructions that depend on any particular setting of bits 0–31 of any 64-bit application-accessible system register, including GPRs, for producing the correct 32-bit results. Context switching is not required to preserve the upper 32 bits of application-accessible 64-bit system registers and insertion of arbitrary settings of those upper 32 bits at arbitrary times during the execution of the 32-bit application must not affect the final result.

B.5.2 32-Bit Addressing

Book E provides a complete set of data memory access instructions that perform a modulo 2^{32} on the computed effective address and then prepend 32 zeros to produce the full 64-bit address. Book E also provides a complete set of branch instructions that perform a modulo 2^{32} on the computed branch target effective address and then prepend 32 zeros to produce the full 64-bit branch target address. On a 32-bit Book E implementation, these instructions are executed as defined, but without prepending the 32 zeros (only the low-order 32 bits of the address are calculated). On a 64-bit implementation, executing these instructions as defined provides the effect of restricting the application to lowest 32-bit address space.

However, there is one exception. Next sequential instruction address computations (not a taken branch) are not defined for 32-bit Book E applications when the current instruction address is 0xFFFF_FFFC. On a 32-bit Book E implementation, the instruction address could simply wrap to 0x0000_0000, providing the same effect that is required in the PowerPC Architecture. However, when the 32-bit Book E application is executed on a 64-bit Book E implementation, the next sequential instruction address calculated will be 0x0000_0001_0000_0000 and not 0x0000_0000_0000_0000. To avoid this problem the 32-bit Book E application must either avoid this situation by not allowing code to span this address boundary, or requiring a branch absolute to address 0 be placed at address 0xFFFF_FFFC to emulate the wrap. Either of these approaches allows the application to execute on 32-bit and 64-bit Book E implementations.

Appendix C
Simplified Mnemonics for PowerPC Instructions

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 PowerPC™ architecture and by implementations of and extensions to the PowerPC architecture.

Most of this information is also provided in the appendixes of reference manuals and the *Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture* (referred to as the *Programming Environment Manual*). However, Section C.11, "Comprehensive List of Simplified Mnemonics," provides an alphabetical listing of simplified mnemonics that are used by a variety of processors. Some assemblers may define additional simplified mnemonics not included here. The simplified mnemonics listed here should be supported by all compilers.

C.1 Overview

Simplified (or extended) mnemonics allow an assembly-language programmer to program using more intuitive mnemonics and symbols than the instructions and syntax defined by the instruction set architecture. For example, to code the conditional call "branch to an absolute target if CR4 specifies a greater than condition, setting the LR without simplified mnemonics, the programmer would write the branch conditional instruction, **bc 12,17,***target*. The simplified mnemonic, branch if greater than, **bgt cr4,***target*, incorporates the conditions. Not only is it easier to remember the symbols than the numbers when programming, it is also easier to interpret simplified mnemonics when reading existing code.

Although the original PowerPC architecture 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 (for example, AltiVec instructions and Book E auxiliary processing units (APUs)). 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.

C.2 Subtract Simplified Mnemonics

This section describes simplified mnemonics for subtract instructions.

C.2.1 Subtract Immediate

There is no subtract immediate instruction, however, its effect is achieved by negating the immediate operand of an Add Immediate instruction, **addi**. Simplified mnemonics include this negation, making the intent of the computation more clear. These are listed in Table C-1.

Table C-1. Subtract Immediate Simplified Mnemonics

Simplified Mnemonic	Standard Mnemonic
subi **r**D,**r**A,value	**addi** **r**D,**r**A,–value
subis **r**D,**r**A,value	**addis** **r**D,**r**A,–value
subic **r**D,**r**A,value	**addic** **r**D,**r**A,–value
subic. **r**D,**r**A,value	**addic.** **r**D,**r**A,–value

C.2.2 Subtract

Subtract from instructions subtract the second operand (**r**A) from the third (**r**B). The simplified mnemonics in Table C-2 use the more common order in which the third operand is subtracted from the second.

Table C-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.

C.3 Rotate and Shift Simplified Mnemonics

Rotate and shift instructions provide powerful, general ways to manipulate register contents, but can be difficult to understand. Simplified mnemonics are provided for the following operations:

- Extract—Select a field of *n* bits starting at bit position *b* in the source register; left or right justify this field in the target register; clear all other bits of the target register.
- Insert—Select a left- or right-justified field of *n* bits in the source register; insert this field starting at bit position *b* of the target register; leave other bits of the target register unchanged.
- Rotate—Rotate the contents of a register right or left *n* bits without masking.

- Shift—Shift the contents of a register right or left *n* bits, clearing vacated bits (logical shift).
- Clear—Clear the leftmost or rightmost *n* bits of a register.
- Clear left and shift left—Clear the leftmost *b* bits of a register, then shift the register left by *n* bits. This operation can be used to scale a (known non-negative) array index by the width of an element.

C.3.1 Operations on Words

The simplified mnemonics in Table C-3 can be coded with a dot (**.**) suffix to cause the Rc bit to be set in the underlying instruction.

Table C-3. Word Rotate and Shift Simplified Mnemonics

Operation	Simplified Mnemonic	Equivalent to:
Extract and left justify word immediate	**extlwi** r**A**,**r**S,*n*,*b* (*n* > 0)	**rlwinm** r**A**,**r**S,*b*,**0**,*n* – 1
Extract and right justify word immediate	**extrwi** r**A**,**r**S,*n*,*b* (*n* > 0)	**rlwinm** r**A**,**r**S,*b* + *n*, 32 – *n*,**31**
Insert from left word immediate	**inslwi** r**A**,**r**S,*n*,*b* (*n* > 0)	**rlwimi** r**A**,**r**S,32 – *b*,*b*,(*b* + *n*) – 1
Insert from right word immediate	**insrwi** r**A**,**r**S,*n*,*b* (*n* > 0)	**rlwimi** r**A**,**r**S,32 – (*b* + *n*),*b*,(*b* + *n*) – 1
Rotate left word immediate	**rotlwi** r**A**,**r**S,*n*	**rlwinm** r**A**,**r**S,*n*,**0**,**31**
Rotate right word immediate	**rotrwi** r**A**,**r**S,*n*	**rlwinm** r**A**,**r**S,32 – *n*,**0**,**31**
Rotate word left	**rotlw** r**A**,**r**S,**r**B	**rlwnm** r**A**,**r**S,**r**B,**0**,**31**
Shift left word immediate	**slwi** r**A**,**r**S,*n* (*n* < 32)	**rlwinm** r**A**,**r**S,*n*,**0**,31 – *n*
Shift right word immediate	**srwi** r**A**,**r**S,*n* (*n* < 32)	**rlwinm** r**A**,**r**S,32 – *n*,*n*,**31**
Clear left word immediate	**clrlwi** r**A**,**r**S,*n* (*n* < 32)	**rlwinm** r**A**,**r**S,**0**,*n*,**31**
Clear right word immediate	**clrrwi** r**A**,**r**S,*n* (*n* < 32)	**rlwinm** r**A**,**r**S,**0**,**0**,31 – *n*
Clear left and shift left word immediate	**clrlslwi** r**A**,**r**S,*b*,*n* (*n* ≤ *b* ≤ 31)	**rlwinm** r**A**,**r**S,*n*,*b* – *n*,31 – *n*

Examples using word mnemonics follow:

1. Extract the sign bit (bit 0) of **r**S and place the result right-justified into **r**A.
   extrwi rA**,r**S**,1,0** equivalent to **rlwinm r**A**,r**S**,1,31,31**
2. Insert the bit extracted in (1) into the sign bit (bit 0) of **r**B.
   insrwi rB**,r**A**,1,0** equivalent to **rlwimi r**B**,r**A**,31,0,0**
3. Shift the contents of **r**A left 8 bits.
   slwi rA**,r**A**,8** equivalent to **rlwinm r**A**,r**A**,8,0,23**
4. Clear the high-order 16 bits of **r**S and place the result into **r**A.
   clrlwi rA**,r**S**,16** equivalent to **rlwinm r**A**,r**S**,0,16,31**

C.4 Branch Instruction Simplified Mnemonics

Branch conditional instructions can be coded with the operations, a condition to be tested, and a prediction, as part of the instruction mnemonic rather than as numeric operands (the BO and BI operands). Table C-4 shows the four general types of branch instructions. Simplified mnemonics are defined only for branch instructions that include BO and BI operands; there is no need to simplify unconditional branch mnemonics.

Table C-4. Branch Instructions

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

The BO and BI operands correspond to two fields in the instruction opcode, as Figure C-1 shows for Branch Conditional (**bc**, **bca**, **bcl**, and **bcla**) instructions.

0–5	6–10	11–15	16–29	30	31
0 0 1 0 0 0	BO	BI	BD	AA	LK

Figure C-1. Branch Conditional (bc) Instruction Format

The BO operand specifies branch operations that involve decrementing CTR. It is also used to determine whether testing a CR bit causes a branch to occur if the condition is true or false.

The BI operand identifies a CR bit to test (whether a comparison is less than or greater than, for example). The simplified mnemonics avoid the need to memorize the numerical values for BO and BI.

For example, **bc 16,0,***target* is a conditional branch that, as a BO value of 16 (0b1_0000) indicates, decrements the 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 C-6); the CR0 equal field is indicated by a BI value of 2 (as shown in Table C-7). Incorporating the branch-if-true condition adds a '**t**' to the simplified mnemonic, **bdnzt.** The equal condition that is specified by a BI value of 2 (indicating the EQ bit in CR0) is replaced by the **eq** symbol. Using the simplified mnemonic and the **eq** operand, the expression becomes **bdnzt eq,***target*.

This example tests CR0[EQ]; however, to test the equal condition in CR5 (CR bit 22), the expression becomes **bc 8,22,***target*. The BI operand of 22 indicates CR[22] (CR5[2], or BI field 0b10110), as shown in Table C-7. This can be expressed as the simplified mnemonic. **bdnzt 4 * cr5 + eq,***target.*

The notation, **4 * cr5 + eq** may at first seem awkward, but it eliminates computing the value of the CR bit. It can be seen that (4 * 5) + 2 = 22. Note that although 32-bit registers in Book E processors are numbered 32–63, only values 0–31 are valid (or possible) for BI operands. As shown in Table C-8, a Book E–compliant processor automatically translates the bit values; specifying a BI value of 22 selects bit 54 on a Book E processor, or CR5[2] = CR5[EQ].

C.4.1 Key Facts about Simplified Branch Mnemonics

The following key points are helpful in understanding how to use simplified branch mnemonics:

- All simplified branch mnemonics eliminate the BO operand, so if any operand is present in a branch simplified mnemonic, it is the BI operand (or a reduced form of it).
- If the CR is not involved in the branch, the BI operand can be deleted
- If the CR is involved in the branch, the BI operand can be treated in the following ways:
  - It can be specified as a numeric value, just as it is in the architecturally defined instruction, or it can be indicated with an easier to remember formula, **4** * **cr***n* **+** [test bit symbol], where *n* indicates the CR field number.
  - The condition of the test bit (eq, lt, gt, and so) can be incorporated into the mnemonic, leaving the need for an operand that defines only the CR field.
    - If the test bit is in CR0, no operand is needed.
    - If the test bit is in CR1–CR7, the BI operand can be replaced with a **cr**S operand (that is, **cr1**, **cr2**, **cr3**, and so forth.

C.4.2 Eliminating the BO Operand

The 5-bit BO field, shown in Figure C-2, encodes the following operations in conditional branch instructions:

- Decrement count register (CTR)
  - And test if result is equal to zero
  - And test if result is not equal to zero

- Test condition register (CR)
  - Test condition true
  - Test condition false
- Branch prediction (taken, fall through). If the prediction bit, *y*, is needed, it is signified by appending a plus or minus sign as described in Section C.4.3, "Incorporating the BO Branch Prediction."

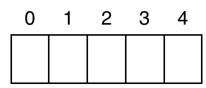


Figure C-2. BO Field (Bits 6–10 of the Instruction Encoding)

BO bits can be interpreted individually as described in Table C-5.

Table C-5. BO Bit Encodings

BO Bit	Description
0	If set, ignore the CR bit comparison.
1	If set, the CR bit comparison is against true, if not set the CR bit comparison is against false
2	If set, the CTR is not decremented.
3	If BO[2] is set, this bit determines whether the CTR comparison is for equal to zero or not equal to zero.
4	The *y* bit. If set, reverse the static prediction. Use of the this bit is optional and independent from the interpretation of the rest of the BO operand. Because simplified branch mnemonics eliminate the BO operand, this bit is programmed by adding a plus or minus sign to the simplified mnemonic, as described in Section C.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 C-6. A *z* bit indicates that the bit is ignored. However, these bits should be cleared, as they may be assigned a meaning in a future version of the architecture.

As shown in Table C-6, the '**c**' in the standard mnemonic is replaced with the operations otherwise specified in the BO field, (**d** for decrement, **z** for zero, **nz** for non-zero, **t** for true, and **f** for false).

Table C-6. BO Operand Encodings

BO Field	Value [1] (Decimal)	Description	Symbol
0000*y*	0	Decrement the CTR, then branch if the decremented CTR ≠ 0; condition is FALSE.	**dnzf**
0001*y*	2	Decrement the CTR, then branch if the decremented CTR = 0; condition is FALSE.	**dzf**
001*zy*	4	Branch if the condition is FALSE. [2] Note that 'false' and 'four' both start with 'f'.	**f**
0100*y*	8	Decrement the CTR, then branch if the decremented CTR ≠ 0; condition is TRUE.	**dnzt**
0101*y*	10	Decrement the CTR, then branch if the decremented CTR = 0; condition is TRUE.	**dzt**

Table C-6. BO Operand Encodings (continued)

BO Field	Value [1] (Decimal)	Description	Symbol
011*z* [3]*y*	12	Branch if the condition is TRUE. [2] Note that ‘true’ and ‘twelve’ both start with ‘t’.	**t**
1*z*00*y* [4]	16	Decrement the CTR, then branch if the decremented CTR ≠ 0.	**dnz** [5]
1*z*01*y* [4]	18	Decrement the CTR, then branch if the decremented CTR = 0.	**dz** [5]
1*z*1*zz* [4]	20	Branch always.	—

[1] Assumes y = z = 0. Section C.4.3, “Incorporating the BO Branch Prediction,” describes how to use simplified mnemonics to program the *y* bit for static prediction.

[2] 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 C.4.6, “Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO and Replaces BI with crS).”

[3] A *z* bit indicates a bit that is ignored. However, these bits should be cleared, as they may be assigned a meaning in a future version of the architecture.

[4] 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.

[5] 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.

C.4.3 Incorporating the BO Branch Prediction

As shown in Table C-6, the low-order bit (*y* bit) of the BO field provides a hint about whether the branch is likely to be taken (static branch prediction). Assemblers should clear this bit 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 *y* bit. That is, ‘+’ indicates that the branch is to be taken and ‘–’ indicates that the branch is not to be taken. This suffix can be added to any branch conditional mnemonic, either standard or simplified.

For relative and absolute branches (**bc**[**l**][**a**]), the setting of the *y* bit 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 non-negative 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 *y* bit to be set, and coding the suffix ‘–’ causes the bit to be cleared.

Examples of branch prediction follow:

1. Branch if CR0 reflects less than condition, specifying that the branch should be predicted as taken.

   blt+ *target*

2. Same as (1), but target address is in the LR and the branch should be predicted as not taken.

   bltlr–

C.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 C.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 C.4, "Branch Instruction Simplified Mnemonics," where **bdnzt 4 * cr5 + eq,***target* could be used instead of **bdnzt 22,***target*. This is described in Section C.4.4.1.1, "Specifying a CR Bit."

  The simplified mnemonics in Section C.4.5, "Simplified Mnemonics that Incorporate the BO Operand," use one of these two methods to specify a CR bit.

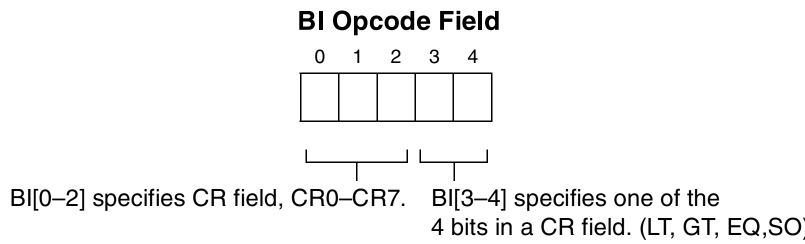
- Additional simplified mnemonics are specified that incorporate CR conditions that would otherwise be specified by the BI operand, so the BI operand is replaced by the **cr**S operand to specify the CR field, CR0–CR7. See Section C.4.4.1, "BI Operand Instruction Encoding."

  These mnemonics are described in Section C.4.6, "Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO and Replaces BI with crS)."

C.4.4.1 BI Operand Instruction Encoding

The entire 5-bit BI field, shown in Figure C-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 C.4.6, "Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO and Replaces BI with crS)," the BI operand is replaced by a **cr**S operand. To understand this, it is useful to view the BI operand as comprised of two parts. As Figure C-3 shows, BI[0–2] indicates the CR field and BI[3–4] represents the condition to test.



Simplified mnemonics based on CR conditions but not CTR values—BO = 12 (branch if true) and BO = 4 branch if false)	Specified by a separate, reduced BI operand (**cr**S)	Incorporated into the simplified mnemonic.
Standard branch mnemonics and simplified mnemonics based on CTR values	The BI operand specifies the entire 5-bit field. If CR0 is used, the bit can be identified by LT, GT, EQ, or SO. If CR1–CR7 are used, the form 4 * **cr**S + LT|GT|EQ|SO can be used.	

Figure C-3. BI Field (Bits 11–14 of the Instruction Encoding)

Integer record-form instructions update CR0, as described in Table C-7.

C.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 C-7 and Table C-8.

Table C-7. CR0 and CR1 Fields as Updated by Integer Instructions

CR*n* Bit	CR Bits		BI		Description
	AIM	Book E	0–2	3–4	
CR0[0]	0	32	000	00	Negative (LT)—Set when the result is negative.
CR0[1]	1	33	000	01	Positive (GT)—Set when the result is positive (and not zero).
CR0[2]	2	34	000	10	Zero (EQ)—Set when the result is zero.
CR0[3]	3	35	000	11	Summary overflow (SO). Copy of XER[SO] at the instruction’s completion.

Some simplified mnemonics incorporate only the BO field (as described Section C.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 C-8.

Compare word instructions (described in Section C.5, “Compare Word Simplified Mnemonics”), 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 C-7. CR logical instructions, described in Section C.6, “Condition Register Logical Simplified Mnemonics,” can update individual CR bits.

Table C-8. BI Operand Settings for CR Fields for Branch Comparisons

CRn Bit	Bit Expression	CR Bits: AIM (BI Operand)	CR Bits: Book E	BI 0–2	BI 3–4	Description
CRn[0]	**4 * cr0 + lt** (or **lt**)	0	32	000	00	Less than (LT). For integer compare instructions: **r**A < SIMM or **r**B (signed comparison) or **r**A < UIMM or **r**B (unsigned comparison).
	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		
CRn[1]	**4 * cr0 + gt** (or **gt**)	1	33	000	01	Greater than (GT). For integer compare instructions: rA > SIMM or rB (signed comparison) or **r**A > UIMM or **r**B (unsigned comparison).
	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		
CRn[2]	**4 * cr0 + eq** (or **eq**)	2	34	000	10	Equal (EQ). For integer compare instructions: **r**A = SIMM, UIMM, or **r**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		
CRn[3]	**4 * cr0 + so/un** (or **so/un**)	3	35	000	11	Summary overflow (SO). For integer compare instructions, this is a copy of XER[SO] at instruction completion.
	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 C.4.5, "Simplified Mnemonics that Incorporate the BO Operand," covers useful cases. Unusual cases can be coded using a standard branch conditional syntax.

C.4.4.1.2 The crS Operand

The **cr**S symbols are shown in Table C-9. Note that either the symbol or the operand value can be used in the syntax used with the simplified mnemonic.

Table C-9. CR Field Identification Symbols

Symbol	BI[0–2]	CR Bits
cr0 (default, can be eliminated from syntax)	000	32–35
cr1	001	36–39
cr2	010	40–43
cr3	011	44–47
cr4	100	48–51
cr5	101	52–55
cr6	110	56–59
cr7	111	60–63

To identify a CR bit, an expression in which a CR field symbol is multiplied by 4 and then added to a bit-number-within-CR-field symbol can be used, (for example, **cr0 * 4 + eq**).

C.4.5 Simplified Mnemonics that Incorporate the BO Operand

The mnemonics in Table C-10 allow common BO operand encodings to be specified as part of the mnemonic, along with the absolute address (AA) and set link register bits (LK). There are no simplified mnemonics for relative and absolute unconditional branches. For these, the basic mnemonics **b**, **ba**, **bl**, and **bla** are used.

Table C-10. Branch Simplified Mnemonics

Branch Semantics	LR Update Not Enabled				LR Update Enabled			
	bc	**bca**	**bclr**	**bcctr**	**bcl**	**bcla**	**bclrl**	**bcctrl**
Branch unconditionally [1]	—	—	**blr**	**bctr**	—	—	**blrl**	**bctrl**
Branch if condition true	**bt**	**bta**	**btlr**	**btctr**	**btl**	**btla**	**btlrl**	**btctrl**
Branch if condition false	**bf**	**bfa**	**bflr**	**bfctr**	**bfl**	**bfla**	**bflrl**	**bfctrl**
Decrement CTR, branch if CTR ≠ 0 [1]	**bdnz**	**bdnza**	**bdnzlr**	—	**bdnzl**	**bdnzla**	**bdnzlrl**	—
Decrement CTR, branch if CTR ≠ 0 and condition true	**bdnzt**	**bdnzta**	**bdnztlr**	—	**bdnztl**	**bdnztla**	**bdnztlrl**	—
Decrement CTR, branch if CTR ≠ 0 and condition false	**bdnzf**	**bdnzfa**	**bdnzflr**	—	**bdnzfl**	**bdnzfla**	**bdnzflrl**	—
Decrement CTR, branch if CTR = 0 [1]	**bdz**	**bdza**	**bdzlr**	—	**bdzl**	**bdzla**	**bdzlrl**	—
Decrement CTR, branch if CTR = 0 and condition true	**bdzt**	**bdzta**	**bdztlr**	—	**bdztl**	**bdztla**	**bdztlrl**	—
Decrement CTR, branch if CTR = 0 and condition false	**bdzf**	**bdzfa**	**bdzflr**	—	**bdzfl**	**bdzfla**	**bdzflrl**	—

[1] Simplified mnemonics for branch instructions that do not test CR bits should specify only a target. Otherwise a programming error may occur.

Table C-10 shows the syntax for basic simplified branch mnemonics

Table C-11. Branch Instructions

Instruction	Standard Mnemonic	Syntax	Simplified Mnemonic	Syntax
Branch	**b** (**ba bl bla**)	target_addr	N/A, syntax does not include BO	
Branch Conditional	**bc** (**bca bcl bcla**)	BO,BI,target_addr	**b***x* [1](**b***x***a b***x***l b***x***la**)	BI [2],target_addr
Branch Conditional to Link Register	**bclr** (**bclrl**)	BO,BI	**b***x***lr** (**b***x***lrl**)	BI
Branch Conditional to Count Register	**bcctr** (**bcctrl**)	BO,BI	**b***x***ctr** (**b***x***ctrl**)	BI

[1] *x* stands for one of the symbols in Table C-6, where applicable.

[2] BI can be a numeric value or an expression as shown in Table C-9.

The simplified mnemonics in Table C-10 that test a condition require a corresponding CR bit as the first operand (as examples 2–5 below illustrate). The symbols in Table C-9 can be used in place of a numeric value.

C.4.5.1 Examples that Eliminate the BO Operand

The simplified mnemonics in Table C-10 are used in the following examples:

1. Decrement CTR and branch if it is still nonzero (closure of a loop controlled by a count loaded into CTR) (note that no CR bits are tested).
   bdnz *target* equivalent to **bc 16,0,***target*

   Because this instruction does not test a CR bit, the simplified mnemonic should specify only a target operand. Specifying a CR (for example, **bdnz** 0,*target* or **bdnz cr0,***target*) may be considered a programming error. Subsequent examples test conditions).
2. Same as (1) but branch only if CTR is nonzero and equal condition in CR0.
   bdnzt eq,*target* equivalent to **bc 8,2,***target*

   Other equivalents include **bdnzt 2,***target* or the unlikely **bdnzt 4*cr0+eq,***target*
3. Same as (2), but equal condition is in CR5.
   bdnzt 4 * cr5 + eq,*target* equivalent to **bc 8,22,***target*

   bdnzt 22,*target* would also work
4. Branch if bit 59 of CR is false.
   bf 27,*target* equivalent to **bc 4,27,***target*

   bf 4*cr6+so,*target* would also work
5. Same as (4), but set the link register. This is a form of conditional call.
   bfl 27,*target* equivalent to **bcl 4,27,***target*

Table C-12 lists simplified mnemonics and syntax for **bc** and **bca** without LR updating.

Table C-12. Simplified Mnemonics for bc and bca without LR Update

Branch Semantics	bc	Simplified Mnemonic	bca	Simplified Mnemonic
Branch unconditionally	—	—	—	—
Branch if condition true [1]	**bc 12,BI**,target	**bt BI**,target	**bca 12,BI**,target	**bta BI**,target
Branch if condition false [1]	**bc 4,BI**,target	**bf BI**,target	**bca 4,BI**,target	**bfa BI**,target
Decrement CTR, branch if CTR ≠ 0	**bc 16,0**,target	**bdnz** target [2]	**bca 16,0**,target	**bdnza** target [2]
Decrement CTR, branch if CTR ≠ 0 and condition true	**bc 8,BI**,target	**bdnzt BI**,target	**bca 8,BI**,target	**bdnzta BI**,target
Decrement CTR, branch if CTR ≠ 0 and condition false	**bc 0,BI**,target	**bdnzf BI**,target	**bca 0,BI**,target	**bdnzfa BI**,target
Decrement CTR, branch if CTR = 0	**bc 18,0**,target	**bdz** target[2]	**bca 18,0**,target	**bdza** target[2]
Decrement CTR, branch if CTR = 0 and condition true	**bc 10,BI**,target	**bdzt BI**,target	**bca 10,BI**,target	**bdzta BI**,target
Decrement CTR, branch if CTR = 0 and condition false	**bc 2,BI**,target	**bdzf BI**,target	**bca 2,BI**,target	**bdzfa BI**,target

1 Instructions for which B0 is either 12 (branch if condition true) or 4 (branch if condition false) do not depend on the CTR value and can be alternately coded by incorporating the condition specified by the BI field, as described in Section C.4.6, "Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO and Replaces BI with crS)."

2 Simplified mnemonics for branch instructions that do not test CR bits should specify only a target. Otherwise a programming error may occur.

Table C-13 lists simplified mnemonics and syntax for **bclr** and **bcctr** without LR updating.

Table C-13. Simplified Mnemonics for bclr and bcctr without LR Update

Branch Semantics	bclr	Simplified Mnemonic	bcctr	Simplified Mnemonic
Branch unconditionally	**bclr 20,0**	**blr** [1]	**bcctr 20,0**	**bctr** [1]
Branch if condition true [2]	**bclr 12,BI**	**btlr BI**	**bcctr 12,BI**	**btctr BI**
Branch if condition false [2]	**bclr 4,BI**	**bflr BI**	**bcctr 4,BI**	**bfctr BI**
Decrement CTR, branch if CTR ≠ 0	**bclr 16,BI**	**bdnzlr BI**	—	—
Decrement CTR, branch if CTR ≠ 0 and condition true	**bclr 8,BI**	**bdnztlr BI**	—	—
Decrement CTR, branch if CTR ≠ 0 and condition false	**bclr 0,BI**	**bdnzflr BI**	—	—
Decrement CTR, branch if CTR = 0	**bclr 18,0**	**bdzlr** [1]	—	—
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 C.4.6, "Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO and Replaces BI with crS)."

Table C-14 provides simplified mnemonics and syntax for **bcl** and **bcla**.

Table C-14. Simplified Mnemonics for bcl and bcla with LR Update

Branch Semantics	bcl	Simplified Mnemonic	bcla	Simplified Mnemonic
Branch unconditionally	—	—	—	—
Branch if condition true [1]	**bcl 12,BI,**target	**btl BI,**target	**bcla 12,BI,**target	**btla BI,**target
Branch if condition false [1]	**bcl 4,BI,**target	**bfl BI,**target	**bcla 4,BI,**target	**bfla BI,**target
Decrement CTR, branch if CTR ≠ 0	**bcl 16,0,**target	**bdnzl** target [2]	**bcla 16,0,**target	**bdnzla** target [2]
Decrement CTR, branch if CTR ≠ 0 and condition true	**bcl 8,0,**target	**bdnztl BI,**target	**bcla 8,BI,**target	**bdnztla BI,**target
Decrement CTR, branch if CTR ≠ 0 and condition false	**bcl 0,BI,**target	**bdnzfl BI,**target	**bcla 0,BI,**target	**bdnzfla BI,**target
Decrement CTR, branch if CTR = 0	**bcl 18,BI,**target	**bdzl** target [2]	**bcla 18,BI,**target	**bdzla** target [2]
Decrement CTR, branch if CTR = 0 and condition true	**bcl 10,BI,**target	**bdztl BI,**target	**bcla 10,BI,**target	**bdztla BI,**target
Decrement CTR, branch if CTR = 0 and condition false	**bcl 2,BI,**target	**bdzfl BI,**target	**bcla 2,BI,**target	**bdzfla BI,**target

[1] Instructions for which B0 is either 12 (branch if condition true) or 4 (branch if condition false) do not depend on the CTR value and can be alternately coded by incorporating the condition specified by the BI field. See Section C.4.6, "Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO and Replaces BI with crS)."

[2] Simplified mnemonics for branch instructions that do not test CR bits should specify only a target. A programming error may occur.

Table C-15 provides simplified mnemonics and syntax for **bclrl** and **bcctrl** with LR updating.

Table C-15. Simplified Mnemonics for bclrl and bcctrl with LR Update

Branch Semantics	bclrl	Simplified Mnemonic	bcctrl	Simplified Mnemonic
Branch unconditionally	**bclrl 20,0**	**blrl** [1]	**bcctrl 20,0**	**bctrl** [1]
Branch if condition true	**bclrl 12,BI**	**btlrl BI**	**bcctrl 12,BI**	**btctrl BI**
Branch if condition false	**bclrl 4,BI**	**bflrl BI**	**bcctrl 4,BI**	**bfctrl BI**
Decrement CTR, branch if CTR ≠ 0	**bclrl 16,0**	**bdnzlrl** [1]	—	—
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.

C.4.6 Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO and Replaces BI with crS)

The mnemonics in Table C-18 are variations of the branch-if-condition-true (BO = 12) and branch-if-condition-false (BO = 4) encodings. Because these instructions do not depend on the CTR, the true/false conditions specified by BO can be combined with the CR test bit specified by BI to create a different set of simplified mnemonics that eliminates the BO operand and the portion of the BI operand (BI[3–4]) that specifies one of the four possible test bits. However, the simplified mnemonic cannot specify in which of the eight CR fields the test bit falls, so the BI operand is replaced by a **cr**S operand.

The standard codes shown in Table C-16 are used for the most common combinations of branch conditions. Note that for ease of programming, these codes include synonyms; for example, less than or equal (**le**) and not greater than (**ng**) achieve the same result.

NOTE

A CR field symbol, **cr0**–**cr7**, is used as the first operand after the simplified mnemonic. If the default, CR0, is used, no **cr**S is necessary,

Table C-16. Standard Coding for Branch Conditions

Code	Description	Equivalent	Bit Tested
lt	Less than	—	LT
le	Less than or equal (equivalent to **ng**)	**ng**	GT
eq	Equal	—	EQ
ge	Greater than or equal (equivalent to **nl**)	**nl**	LT
gt	Greater than	—	GT
nl	Not less than (equivalent to **ge**)	**ge**	LT
ne	Not equal	—	EQ
ng	Not greater than (equivalent to **le**)	**le**	GT
so	Summary overflow	—	SO
ns	Not summary overflow	—	SO
un	Unordered (after floating-point comparison)	—	SO
nu	Not unordered (after floating-point comparison)	—	SO

Table C-17 shows the syntax for simplified branch mnemonics that incorporate CR conditions. Here, **cr**S replaces a BI operand to specify only a CR field (because the specific CR bit within the field is now part of the simplified mnemonic. Note that the default is CR0; if no **cr**S is specified, CR0 is used.

Table C-17. Branch Instructions and Simplified Mnemonics that Incorporate CR Conditions

Instruction	Standard Mnemonic	Syntax	Simplified Mnemonic	Syntax
Branch	**b** (**ba bl bla**)	target_addr	—	
Branch Conditional	**bc** (**bca bcl bcla**)	BO,BI,target_addr	**b***x* [1](**b***x***a b***x***l b***x***la**)	**cr**S [2],target_addr
Branch Conditional to Link Register	**bclr** (**bclrl**)	BO,BI	**b***x***lr** (**b***x***lrl**)	**cr**S
Branch Conditional to Count Register	**bcctr** (**bcctrl**)	BO,BI	**b***x***ctr** (**b***x***ctrl**)	**cr**S

[1] *x* stands for one of the symbols in Table C-16, where applicable.

[2] BI can be a numeric value or an expression as shown in Table C-9.

Table C-18 shows the simplified branch mnemonics incorporating conditions.

Table C-18. Simplified Mnemonics with Comparison Conditions

Branch Semantics	LR Update Not Enabled				LR Update Enabled			
	bc	**bca**	**bclr**	**bcctr**	**bcl**	**bcla**	**bclrl**	**bcctrl**
Branch if less than	**blt**	**blta**	**bltlr**	**bltctr**	**bltl**	**bltla**	**bltlrl**	**bltctrl**
Branch if less than or equal	**ble**	**blea**	**blelr**	**blectr**	**blel**	**blela**	**blelrl**	**blectrl**
Branch if equal	**beq**	**beqa**	**beqlr**	**beqctr**	**beql**	**beqla**	**beqlrl**	**beqctrl**
Branch if greater than or equal	**bge**	**bgea**	**bgelr**	**bgectr**	**bgel**	**bgela**	**bgelrl**	**bgectrl**
Branch if greater than	**bgt**	**bgta**	**bgtlr**	**bgtctr**	**bgtl**	**bgtla**	**bgtlrl**	**bgtctrl**
Branch if not less than	**bnl**	**bnla**	**bnllr**	**bnlctr**	**bnll**	**bnlla**	**bnllrl**	**bnlctrl**
Branch if not equal	**bne**	**bnea**	**bnelr**	**bnectr**	**bnel**	**bnela**	**bnelrl**	**bnectrl**
Branch if not greater than	**bng**	**bnga**	**bnglr**	**bngctr**	**bngl**	**bngla**	**bnglrl**	**bngctrl**
Branch if summary overflow	**bso**	**bsoa**	**bsolr**	**bsoctr**	**bsol**	**bsola**	**bsolrl**	**bsoctrl**
Branch if not summary overflow	**bns**	**bnsa**	**bnslr**	**bnsctr**	**bnsl**	**bnsla**	**bnslrl**	**bnsctrl**
Branch if unordered	**bun**	**buna**	**bunlr**	**bunctr**	**bunl**	**bunla**	**bunlrl**	**bunctrl**
Branch if not unordered	**bnu**	**bnua**	**bnulr**	**bnuctr**	**bnul**	**bnula**	**bnulrl**	**bnuctrl**

Instructions using the mnemonics in Table C-18 indicate the condition bit, but not the CR field. If no field is specified, CR0 is used. The CR field symbols defined in Table C-9 (**cr0**–**cr7**) are used for this operand, as shown in examples 2–4 below.

C.4.6.1 Branch Simplified Mnemonics that Incorporate CR Conditions: Examples

The following examples use the simplified mnemonics shown in Table C-18:

1. Branch if CR0 reflects not-equal condition.
   bne *target* equivalent to **bc 4,2,***target*
2. Same as (1) but condition is in CR3.
   bne cr3,*target* equivalent to **bc 4,14,***target*
3. Branch to an absolute target if CR4 specifies greater than condition, setting the LR. This is a form of conditional call.
   bgtla cr4,*target* equivalent to **bcla 12,17,***target*
4. Same as (3), but target address is in the CTR.
   bgtctrl cr4 equivalent to **bcctrl 12,17**

C.4.6.2 Branch Simplified Mnemonics that Incorporate CR Conditions: Listings

Table C-19 shows simplified branch mnemonics and syntax for **bc** and **bca** without LR updating.

Table C-19. Simplified Mnemonics for bc and bca without Comparison Conditions or LR Updating

Branch Semantics	bc	Simplified Mnemonic	bca	Simplified Mnemonic
Branch if less than	**bc 12**,BI [1],target	**blt cr**S target	**bca 12**,BI[1],target	**blta cr**S target
Branch if less than or equal	**bc 4**,BI [2],target	**ble cr**S target	**bca 4**,BI[2],target	**blea cr**S target
Branch if not greater than		**bng cr**S target		**bnga cr**S target
Branch if equal	**bc 12**,BI [3],target	**beq cr**S target	**bca 12**,BI[3],target	**beqa cr**S target
Branch if greater than or equal	**bc 4**,BI[1],target	**bge cr**S target	**bca 4**,BI[1],target	**bgea cr**S target
Branch if not less than		**bnl cr**S target		**bnla cr**S target
Branch if greater than	**bc 12**,BI[2],target	**bgt cr**S target	**bca 12**,BI[2],target	**bgta cr**S target
Branch if not equal	**bc 4**,BI[3],target	**bne cr**S target	**bca 4**,BI[3],target	**bnea cr**S target
Branch if summary overflow	**bc 12**,BI [4],target	**bso cr**S target	**bca 12**,BI[4],target	**bsoa cr**S target
Branch if unordered		**bun cr**S target		**buna cr**S target
Branch if not summary overflow	**bc 4**,BI[4],target	**bns cr**S target	**bca 4**,BI[4],target	**bnsa cr**S target
Branch if not unordered		**bnu cr**S target		**bnua cr**S target

[1] The value in the BI operand selects CR*n*[0], the LT bit.
[2] The value in the BI operand selects CR*n*[1], the GT bit.
[3] The value in the BI operand selects CR*n*[2], the EQ bit.
[4] The value in the BI operand selects CR*n*[3], the SO bit.

Table C-20 shows simplified branch mnemonics and syntax for **bclr** and **bcctr** without LR updating.

Table C-20. Simplified Mnemonics for bclr and bcctr without Comparison Conditions and LR Updating

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.

Table C-21 shows simplified branch mnemonics and syntax for **bcl** and **bcla**.

Table C-21. Simplified Mnemonics for bcl and bcla with Comparison Conditions and LR Updating

Branch Semantics	bcl	Simplified Mnemonic	bcla	Simplified Mnemonic
Branch if less than	**bcl 12,**BI [1],target	**bltl cr**S target	**bcla 12,**BI[1],target	**bltla cr**S target
Branch if less than or equal	**bcl 4,**BI [2],target	**blel cr**S target	**bcla 4,**BI[2],target	**blela cr**S target
Branch if not greater than		**bngl cr**S target		**bngla cr**S target
Branch if equal	**bcl 12,**BI [3],target	**beql cr**S target	**bcla 12,**BI[3],target	**beqla cr**S target
Branch if greater than or equal	**bcl 4,**BI[1],target	**bgel cr**S target	**bcla 4,**BI[1],target	**bgela cr**S target
Branch if not less than		**bnll cr**S target		**bnlla cr**S target
Branch if greater than	**bcl 12,**BI[2],target	**bgtl cr**S target	**bcla 12,**BI[2],target	**bgtla cr**S target
Branch if not equal	**bcl 4,**BI[3],target	**bnel cr**S target	**bcla 4,**BI[3],target	**bnela cr**S target
Branch if summary overflow	**bcl 12,**BI [4],target	**bsol cr**S target	**bcla 12,**BI[4],target	**bsola cr**S target
Branch if unordered		**bunl cr**S target		**bunla cr**S target

Table C-21. Simplified Mnemonics for bcl and bcla with Comparison Conditions and LR Updating (continued)

Branch Semantics	bcl	Simplified Mnemonic	bcla	Simplified Mnemonic
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.

Table C-22 shows the simplified branch mnemonics and syntax for **bclrl** and **bcctrl** with LR updating.

Table C-22. Simplified Mnemonics for bclrl and bcctrl with Comparison Conditions and LR Updating

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,**BI [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.

C.5 Compare Word Simplified Mnemonics

In compare word instructions, the L operand indicates a word (L = 0) or double-word (L = 1). Simplified mnemonics in Table C-23 eliminate the L operand for word comparisons.

Table C-23. Word Compare Simplified Mnemonics

Operation	Simplified Mnemonic	Equivalent to:
Compare Word Immediate	**cmpwi cr**D,**r**A,SIMM	**cmpi cr**D,**0**,**r**A,SIMM
Compare Word	**cmpw cr**D,**r**A,**r**B	**cmp cr**D,**0**,**r**A,**r**B
Compare Logical Word Immediate	**cmplwi cr**D,**r**A,UIMM	**cmpli cr**D,**0**,**r**A,UIMM
Compare Logical Word	**cmplw cr**D,**r**A,**r**B	**cmpl cr**D,**0**,**r**A,**r**B

As with branch mnemonics, the **cr**D field of a compare instruction can be omitted if CR0 is used, as shown in examples 1 and 3 below. Otherwise, the target CR field must be specified as the first operand. The following examples use word compare mnemonics:

1. Compare **r**A with immediate value 100 as signed 32-bit integers and place result in CR0.
   cmpwi rA**,100** equivalent to **cmpi 0,0,r**A**,100**
2. Same as (1), but place results in CR4.
   cmpwi cr4,rA**,100** equivalent to **cmpi 4,0,r**A**,100**
3. Compare **r**A and **r**B as unsigned 32-bit integers and place result in CR0.
   cmplw rA**,r**B equivalent to **cmpl 0,0,r**A**,r**B

C.6 Condition Register Logical Simplified Mnemonics

The CR logical instructions, shown in Table C-24, 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 C-8 can be used to identify the CR bit.

Table C-24. Condition Register Logical Simplified Mnemonics

Operation	Simplified Mnemonic	Equivalent to
Condition register set	**crset b**x	**creqv b**x,**b**x,**b**x
Condition register clear	**crclr b**x	**crxor b**x,**b**x,**b**x
Condition register move	**crmove b**x,**b**y	**cror b**x,**b**y,**b**y
Condition register not	**crnot b**x,**b**y	**crnor b**x,**b**y,**b**y

Examples using the CR logical mnemonics follow:

1. Set CR[57].
   crset 25 equivalent to **creqv 25,25,25**
2. Clear CR0[SO].
   crclr so equivalent to **crxor 3,3,3**
3. Same as (2), but clear CR3[SO].
   crclr 4 * cr3 + so equivalent to **crxor 15,15,15**
4. Invert the CR0[EQ].
   crnot eq,eq equivalent to **crnor 2,2,2**
5. Same as (4), but CR4[EQ] is inverted and the result is placed into CR5[EQ].
   crnot 4 * cr5 + eq, 4 * cr4 + eq equivalent to **crnor 22,18,18**

C.7 Trap Instructions Simplified Mnemonics

The codes in Table C-25 have been adopted for the most common combinations of trap conditions.

Table C-25. Standard Codes for Trap Instructions

Code	Description	TO Encoding	<	>	=	<U [1]	>U [2]
lt	Less than	16	1	0	0	0	0
le	Less than or equal	20	1	0	1	0	0
eq	Equal	4	0	0	1	0	0
ge	Greater than or equal	12	0	1	1	0	0
gt	Greater than	8	0	1	0	0	0
nl	Not less than	12	0	1	1	0	0
ne	Not equal	24	1	1	0	0	0
ng	Not greater than	20	1	0	1	0	0
llt	Logically less than	2	0	0	0	1	0
lle	Logically less than or equal	6	0	0	1	1	0
lge	Logically greater than or equal	5	0	0	1	0	1
lgt	Logically greater than	1	0	0	0	0	1
lnl	Logically not less than	5	0	0	1	0	1
lng	Logically not greater than	6	0	0	1	1	0
—	Unconditional	31	1	1	1	1	1

[1] The symbol ‘<U’ indicates an unsigned less-than evaluation is performed.
[2] The symbol ‘>U’ indicates an unsigned greater-than evaluation is performed.

The mnemonics in Table C-26 are variations of trap instructions, with the most useful TO values represented in the mnemonic rather than specified as a numeric operand.

Table C-26. Trap Simplified Mnemonics

Trap Semantics	32-Bit Comparison	
	twi Immediate	tw Register
Trap unconditionally	—	**trap**
Trap if less than	**twlti**	**twlt**
Trap if less than or equal	**twlei**	**twle**
Trap if equal	**tweqi**	**tweq**
Trap if greater than or equal	**twgei**	**twge**
Trap if greater than	**twgti**	**twgt**
Trap if not less than	**twnli**	**twnl**
Trap if not equal	**twnei**	**twne**
Trap if not greater than	**twngi**	**twng**
Trap if logically less than	**twllti**	**twllt**
Trap if logically less than or equal	**twllei**	**twlle**
Trap if logically greater than or equal	**twlgei**	**twlge**
Trap if logically greater than	**twlgti**	**twlgt**
Trap if logically not less than	**twlnli**	**twlnl**
Trap if logically not greater than	**twlngi**	**twlng**

The following examples use the trap mnemonics shown in Table C-26:

1. Trap if **r**A is not zero.
   twnei rA**,0** equivalent to **twi 24,r**A**,0**
2. Trap if **r**A is not equal to **r**B.
   twne rA**, r**B equivalent to **tw 24,r**A**,r**B
3. Trap if **r**A is logically greater than 0x7FF.
   twlgti rA**,** 0x7FF equivalent to **twi 1,r**A**,** 0x7FF
4. Trap unconditionally.
   trap equivalent to **tw 31,0,0**

Trap instructions evaluate a trap condition as follows: The contents of **r**A are compared with either the sign-extended SIMM field or the contents of **r**B, depending on the trap instruction.

The comparison results in five conditions that are ANDed with operand TO. If the result is not 0, the trap exception handler is invoked. See Table C-27 for these conditions.

Table C-27. 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

C.8 Simplified Mnemonics for Accessing SPRs

The **mtspr** and **mfspr** instructions specify a special-purpose register (SPR) as a numeric operand. Simplified mnemonics are provided that represent the SPR in the mnemonic rather than requiring it to be coded as a numeric operand. The pattern for **mtspr** and **mfspr** simplified mnemonics is straightforward: replace the **-spr** portion of the mnemonic with the abbreviation for the spr (for example XER, SRR0, or LR), eliminate the SPRN operand, leaving the source or destination GPR operand, **r**S or **r**D.

Following are examples using the SPR simplified mnemonics:

1. Copy the contents of **r**S to the XER.
   mtxer rS equivalent to **mtspr 1,r**S
2. Copy the contents of the LR to **r**S.
   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 examples above show simplified mnemonics for accessing SPRs defined by the AIM version of the PowerPC architecture; however, the same formula is used for Book E, EIS, and implementation-specific SPRs, as shown in the following examples:

1. Copy the contents of **r**S to CSRR0.
   mtcsrr0 rS equivalent to **mtspr 58,r**S
2. Copy the contents of IVOR0 to **r**S.
   mfivor0 rD equivalent to **mfspr r**D**,400**
3. Copy the contents of **r**S to the MAS1.
   mtmas1 rS equivalent to **mtspr 625,r**S

There is an additional simplified mnemonic formula for accessing SPRGs, although not all of these more complicated simplified mnemonics are supported by all assemblers. These are shown in Table C-28 along with the equivalent simplified mnemonic using the formula described above.

Table C-28. 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	

C.9 Recommended Simplified Mnemonics

This section describes commonly-used operations (such as no-op, load immediate, load address, move register, and complement register).

C.9.1 No-Op (nop)

Many instructions can be coded in a way that, effectively, no operation is performed. An additional 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**

C.9.2 Load Immediate (li)

The **addi** and **addis** instructions can be used to load an immediate value into a register. Additional mnemonics are provided to convey the idea that no addition is being performed but that data is being moved from the immediate operand of the instruction to a register.

1. Load a 16-bit signed immediate value into **r**D.
   li rD,value equivalent to **addi r**D**,0,**value
2. Load a 16-bit signed immediate value, shifted left by 16 bits, into **r**D.
   lis rD,value equivalent to **addis r**D**,0,**value

C.9.3 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*

C.9.4 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

C.9.5 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

C.9.6 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

C.10 EIS-Specific Simplified Mnemonics

This section describes simplified mnemonics for instructions defines by auxiliary processing units (APUs) defined as part of the Motorola Book E implementation standards (EIS).

C.10.1 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**

C.10.2 SPE Mnemonics

The following mnemonic handles moving of the full 64-bit SPE GPR:

Vector Move		
evmr rD**,r**A	equivalent to	**evor r**D**,r**A**,r**A

The following mnemonic performs a complement register:

Vector Not		
evnot rD**,r**A	equivalent to	**evnor r**D**,r**A**,r**A

C.11 Comprehensive List of Simplified Mnemonics

Table C-29 lists simplified mnemonics that are supported by the e500 processor. Note that compiler designers may implement additional simplified mnemonics not listed here.

Table C-29. Simplified Mnemonics

Simplified Mnemonic	Mnemonic	Instruction
bctr [1]	**bcctr 20,0**	Branch unconditionally (**bcctr** without LR update)
bctrl [1]	**bcctrl 20,0**	Branch unconditionally (**bcctrl** with LR Update)
bdnz target [1]	**bc 16,0,**target	Decrement CTR, branch if CTR ≠ 0 (**bc** without LR update)
bdnza target [1]	**bca 16,0,**target	Decrement CTR, branch if CTR ≠ 0 (**bca** without LR update)

Table C-29. Simplified Mnemonics (continued)

Simplified Mnemonic	Mnemonic	Instruction
bdnzf BI,target	**bc 0,BI,**target	Decrement CTR, branch if CTR ≠ 0 and condition false (**bc** without LR update)
bdnzfa BI,target	**bca 0,BI,**target	Decrement CTR, branch if CTR ≠ 0 and condition false (**bca** without LR update)
bdnzfl BI,target	**bcl 0,BI,**target	Decrement CTR, branch if CTR ≠ 0 and condition false (**bcl** with LR update)
bdnzfla BI,target	**bcla 0,BI,**target	Decrement CTR, branch if CTR ≠ 0 and condition false (**bcla** with LR update)
bdnzflr BI	**bclr 0,BI**	Decrement CTR, branch if CTR ≠ 0 and condition false (**bclr** without LR update)
bdnzflrl BI	**bclrl 0,BI**	Decrement CTR, branch if CTR ≠ 0 and condition false (**bclrl** with LR Update)
bdnzl target [1]	**bcl 16,0,**target	Decrement CTR, branch if CTR ≠ 0 (**bcl** with LR update)
bdnzla target [1]	**bcla 16,0,**target	Decrement CTR, branch if CTR ≠ 0 (**bcla** with LR update)
bdnzlr BI	**bclr 16,BI**	Decrement CTR, branch if CTR ≠ 0 (**bclr** without LR update)
bdnzlrl [1]	**bclrl 16,0**	Decrement CTR, branch if CTR ≠ 0 (**bclrl** with LR Update)
bdnzt BI,target	**bc 8,BI,**target	Decrement CTR, branch if CTR ≠ 0 and condition true (**bc** without LR update)
bdnzta BI,target	**bca 8,BI,**target	Decrement CTR, branch if CTR ≠ 0 and condition true (**bc**a without LR update)
bdnztl BI,target	**bcl 8,0,**target	Decrement CTR, branch if CTR ≠ 0 and condition true (**bcl** with LR update)
bdnztla BI,target	**bcla 8,BI,**target	Decrement CTR, branch if CTR ≠ 0 and condition true (**bcla** with LR update)
bdnztlr BI	**bclr 8,BI**	Decrement CTR, branch if CTR ≠ 0 and condition true (**bclr** without LR update)
bdnztlr BI	**bclr 8,BI**	Decrement CTR, branch if CTR = 0 and condition true (**bclr** without LR update)
bdnztlrl BI	**bclrl 8,BI**	Decrement CTR, branch if CTR ≠ 0 and condition true (**bclrl** with LR Update)
bdz target [1]	**bc 18,0,**target	Decrement CTR, branch if CTR = 0 (**bc** without LR update)
bdza target [1]	**bca 18,0,**target	Decrement CTR, branch if CTR = 0 (**bca** without LR update)
bdzf BI,target	**bc 2,BI,**target	Decrement CTR, branch if CTR = 0 and condition false (**bc** without LR update)
bdzfa BI,target	**bca 2,BI,**target	Decrement CTR, branch if CTR = 0 and condition false (**bca** without LR update)

Table C-29. Simplified Mnemonics (continued)

Simplified Mnemonic	Mnemonic	Instruction
bdzfl BI,target	**bcl 2,BI,**target	Decrement CTR, branch if CTR = 0 and condition false (**bcl** with LR update)
bdzfla BI,target	**bcla 2,BI,**target	Decrement CTR, branch if CTR = 0 and condition false (**bcla** with LR update)
bdzflr BI	**bclr 2,BI**	Decrement CTR, branch if CTR = 0 and condition false (**bclr** without LR update)
bdzflrl BI	**bclrl 2,BI**	Decrement CTR, branch if CTR = 0 and condition false (**bclrl** with LR Update)
bdzl target [1]	**bcl 18,BI,**target	Decrement CTR, branch if CTR = 0 (**bcl** with LR update)
bdzla target [1]	**bcla 18,BI,**target	Decrement CTR, branch if CTR = 0 (**bcla** with LR update)
bdzlr [1]	**bclr 18,0**	Decrement CTR, branch if CTR = 0 (**bclr** without LR update)
bdzlrl [1]	**bclrl 18,0**	Decrement CTR, branch if CTR = 0 (**bclrl** with LR Update)
bdzt BI,target	**bc 10,BI,**target	Decrement CTR, branch if CTR = 0 and condition true (**bc** without LR update)
bdzta BI,target	**bca 10,BI,**target	Decrement CTR, branch if CTR = 0 and condition true (**bca** without LR update)
bdztl BI,target	**bcl 10,BI,**target	Decrement CTR, branch if CTR = 0 and condition true (**bcl** with LR update)
bdztla BI,target	**bcla 10,BI,**target	Decrement CTR, branch if CTR = 0 and condition true (**bcla** with LR update)
bdztlrl BI	**bclrl 10, BI**	Decrement CTR, branch if CTR = 0 and condition true (**bclrl** with LR Update)
beq crS target	**bc 12,**BI [2]**,**target	Branch if equal (**bc** without comparison conditions or LR updating)
beqa crS target	**bca 12,**BI[2]**,**target	Branch if equal (**bca** without comparison conditions or LR updating)
beqctr crS target	**bcctr 12,**BI[2]**,**target	Branch if equal (**bcctr** without comparison conditions and LR updating)
beqctrl crS target	**bcctrl 12,**BI[2]**,**target	Branch if equal (**bcctrl** with comparison conditions and LR update)
beql crS target	**bcl 12,**BI[2]**,**target	Branch if equal (**bcl** with comparison conditions and LR updating)
beqla crS target	**bcla 12,**BI[2]**,**target	Branch if equal (**bcla** with comparison conditions and LR updating)
beqlr crS target	**bclr 12,**BI[2]**,**target	Branch if equal (**bclr** without comparison conditions and LR updating)
beqlrl crS target	**bclrl 12,**BI[2]**,**target	Branch if equal (**bclrl** with comparison conditions and LR update)
bf BI,target	**bc 4,BI,**target	Branch if condition false [3] (**bc** without LR update)

Table C-29. Simplified Mnemonics (continued)

Simplified Mnemonic	Mnemonic	Instruction
bfa BI,target	**bca 4,BI,**target	Branch if condition false [3] (**bca** without LR update)
bfctr BI	**bcctr 4,BI**	Branch if condition false [3] (**bcctr** without LR update)
bfctrl BI	**bcctrl 4,BI**	Branch if condition false [3](**bcctrl** with LR Update)
bfl BI,target	**bcl 4,BI,**target	Branch if condition false [3] (**bcl** with LR update)
bfla BI,target	**bcla 4,BI,**target	Branch if condition false [3] (**bcla** with LR update)
bflr BI	**bclr 4,BI**	Branch if condition false [3] (**bclr** without LR update)
bflrl BI	**bclrl 4,BI**	Branch if condition false [3](**bclrl** with LR Update)
bge crS target	**bc 4,**BI [4]**,**target	Branch if greater than or equal (**bc** without comparison conditions or LR updating)
bgea crS target	**bca 4,**BI[4]**,**target	Branch if greater than or equal (**bca** without comparison conditions or LR updating)
bgectr crS target	**bcctr 4,**BI[4],target	Branch if greater than or equal (**bcctr** without comparison conditions and LR updating)
bgectrl crS target	**bcctrl 4,**BI[4]**,**target	Branch if greater than or equal (**bcctrl** with comparison conditions and LR update)
bgel crS target	**bcl 4,**BI[4],target	Branch if greater than or equal (**bcl** with comparison conditions and LR updating)
bgela crS target	**bcla 4,**BI[4]**,**target	Branch if greater than or equal (**bcla** with comparison conditions and LR updating)
bgelr crS target	**bclr 4,**BI[4]**,**target	Branch if greater than or equal (**bclr** without comparison conditions and LR updating)
bgelrl crS target	**bclrl 4,**BI[4]**,**target	Branch if greater than or equal (**bclrl** with comparison conditions and LR update)
bgt crS target	**bc 12,**BI [5],target	Branch if greater than (**bc** without comparison conditions or LR updating)
bgta crS target	**bca 12,**BI[5],target	Branch if greater than (**bca** without comparison conditions or LR updating)
bgtctr crS target	**bcctr 12,**BI[5],target	Branch if greater than (**bcctr** without comparison conditions and LR updating)
bgtctrl crS target	**bcctrl 12,**BI[5],target	Branch if greater than (**bcctrl** with comparison conditions and LR update)
bgtl crS target	**bcl 12,**BI[5],target	Branch if greater than (**bcl** with comparison conditions and LR updating)
bgtla crS target	**bcla 12,**BI[5]**,**target	Branch if greater than (**bcla** with comparison conditions and LR updating)
bgtlr crS target	**bclr 12,**BI[5],target	Branch if greater than (**bclr** without comparison conditions and LR updating)
bgtlrl crS target	**bclrl 12,**BI[5]**,**target	Branch if greater than (**bclrl** with comparison conditions and LR update)
ble crS target	**bc 4,**BI[5]**,**target	Branch if less than or equal (**bc** without comparison conditions or LR updating)

Table C-29. Simplified Mnemonics (continued)

Simplified Mnemonic	Mnemonic	Instruction
blea crS target	**bca 4,**BI[5]**,**target	Branch if less than or equal (**bca** without comparison conditions or LR updating)
blectr crS target	**bcctr 4,**BI[5]**,**target	Branch if less than or equal (**bcctr** without comparison conditions and LR updating)
blectrl crS target	**bcctrl 4,**BI[5]**,**target	Branch if less than or equal (**bcctrl** with comparison conditions and LR update)
blel crS target	**bcl 4,**BI[5]**,**target	Branch if less than or equal (**bcl** with comparison conditions and LR updating)
blela crS target	**bcla 4,**BI[5]**,**target	Branch if less than or equal (**bcla** with comparison conditions and LR updating)
blelr crS target	**bclr 4,**BI[5]**,**target	Branch if less than or equal (**bclr** without comparison conditions and LR updating)
blelrl crS target	**bclrl 4,**BI[5]**,**target	Branch if less than or equal (**bclrl** with comparison conditions and LR update)
blr [1]	**bclr 20,0**	Branch unconditionally (**bclr** without LR update)
blrl [1]	**bclrl 20,0**	Branch unconditionally (**bclrl** with LR Update)
blt crS target	**bc 12,**BI**,**target	Branch if less than (**bc** without comparison conditions or LR updating)
blta crS target	**bca 12,**BI[4]**,**target	Branch if less than (**bca** without comparison conditions or LR updating)
bltctr crS target	**bcctr 12,**BI[4]**,**target	Branch if less than (**bcctr** without comparison conditions and LR updating)
bltctrl crS target	**bcctrl 12,**BI[4]**,**target	Branch if less than (**bcctrl** with comparison conditions and LR update)
bltl crS target	**bcl 12,**BI[4]**,**target	Branch if less than (**bcl** with comparison conditions and LR updating)
bltla crS target	**bcla 12,**BI[4]**,**target	Branch if less than (**bcla** with comparison conditions and LR updating)
bltlr crS target	**bclr 12,**BI[4]**,**target	Branch if less than (**bclr** without comparison conditions and LR updating)
bltlrl crS target	**bclrl 12,**BI[4]**,**target	Branch if less than (**bclrl** with comparison conditions and LR update)
bne crS target	**bc 4,**BI[3]**,**target	Branch if not equal (**bc** without comparison conditions or LR updating)
bnea crS target	**bca 4,**BI[3]**,**target	Branch if not equal (**bca** without comparison conditions or LR updating)
bnectr crS target	**bcctr 4,**BI[3]**,**target	Branch if not equal (**bcctr** without comparison conditions and LR updating)
bnectrl crS target	**bcctrl 4,**BI[3]**,**target	Branch if not equal (**bcctrl** with comparison conditions and LR update)
bnel crS target	**bcl 4,**BI[3]**,**target	Branch if not equal (**bcl** with comparison conditions and LR updating)

Table C-29. Simplified Mnemonics (continued)

Simplified Mnemonic	Mnemonic	Instruction
bnela crS target	**bcla 4,**BI[3],target	Branch if not equal (**bcla** with comparison conditions and LR updating)
bnelr crS target	**bclr 4,**BI[3],target	Branch if not equal (**bclr** without comparison conditions and LR updating)
bnelrl crS target	**bclrl 4,**BI[3],target	Branch if not equal (**bclrl** with comparison conditions and LR update)
bng crS target	**bc 4,**BI[5],target	Branch if not greater than (**bc** without comparison conditions or LR updating)
bnga crS target	**bca 4,**BI[5],target	Branch if not greater than (**bca** without comparison conditions or LR updating)
bngctr crS target	**bcctr 4,**BI[5],target	Branch if not greater than (**bcctr** without comparison conditions and LR updating)
bngctrl crS target	**bcctrl 4,**BI[5],target	Branch if not greater than (**bcctrl** with comparison conditions and LR update)
bngl crS target	**bcl 4,**BI[5],target	Branch if not greater than (**bcl** with comparison conditions and LR updating)
bngla crS target	**bcla 4,**BI[5],target	Branch if not greater than (**bcla** with comparison conditions and LR updating)
bnglr crS target	**bclr 4,**BI[5],target	Branch if not greater than (**bclr** without comparison conditions and LR updating)
bnglrl crS target	**bclrl 4,**BI[5],target	Branch if not greater than (**bclrl** with comparison conditions and LR update)
bnl crS target	**bc 4,**BI[4],target	Branch if not less than (**bc** without comparison conditions or LR updating)
bnla crS target	**bca 4,**BI[4],target	Branch if not less than (**bca** without comparison conditions or LR updating)
bnlctr crS target	**bcctr 4,**BI[4],target	Branch if not less than (**bcctr** without comparison conditions and LR updating)
bnlctrl crS target	**bcctrl 4,**BI[4],target	Branch if not less than (**bcctrl** with comparison conditions and LR update)
bnll crS target	**bcl 4,**BI[4],target	Branch if not less than (**bcl** with comparison conditions and LR updating)
bnlla crS target	**bcla 4,**BI[4],target	Branch if not less than (**bcla** with comparison conditions and LR updating)
bnllr crS target	**bclr 4,**BI[4],target	Branch if not less than (**bclr** without comparison conditions and LR updating)
bnllrl crS target	**bclrl 4,**BI[4],target	Branch if not less than (**bclrl** with comparison conditions and LR update)
bns crS target	**bc 4,**BI [6],target	Branch if not summary overflow (**bc** without comparison conditions or LR updating)
bnsa crS target	**bca 4,**BI[6],target	Branch if not summary overflow (**bca** without comparison conditions or LR updating)

Table C-29. Simplified Mnemonics (continued)

Simplified Mnemonic	Mnemonic	Instruction
bnsctr crS target	**bcctr 4,**BI[6],target	Branch if not summary overflow (**bcctr** without comparison conditions and LR updating)
bnsctrl crS target	**bcctrl 4,**BI[6]**,**target	Branch if not summary overflow (**bcctrl** with comparison conditions and LR update)
bnsl crS target	**bcl 4,**BI[6]**,**target	Branch if not summary overflow (**bcl** with comparison conditions and LR updating)
bnsla crS target	**bcla 4,**BI[6],target	Branch if not summary overflow (**bcla** with comparison conditions and LR updating)
bnslr crS target	**bclr 4,**BI[6]**,**target	Branch if not summary overflow (**bclr** without comparison conditions and LR updating)
bnslrl crS target	**bclrl 4,**BI[6],target	Branch if not summary overflow (**bclrl** with comparison conditions and LR update)
bnu crS target	**bc 4,**BI[6],target	Branch if not unordered (**bc** without comparison conditions or LR updating)
bnua crS target	**bca 4,**BI[6]**,**target	Branch if not unordered (**bca** without comparison conditions or LR updating)
bnuctr crS target	**bcctr 4,**BI[6],target	Branch if not unordered (**bcctr** without comparison conditions and LR updating)
bnuctrl crS target	**bcctrl 4,**BI[6]**,**target	Branch if not unordered (**bcctrl** with comparison conditions and LR update)
bnul crS target	**bcl 4,**BI[6]**,**target	Branch if not unordered (**bcl** with comparison conditions and LR updating)
bnula crS target	**bcla 4,**BI[6],target	Branch if not unordered (**bcla** with comparison conditions and LR updating)
bnulr crS target	**bclr 4,**BI[6]**,**target	Branch if not unordered (**bclr** without comparison conditions and LR updating)
bnulrl crS target	**bclrl 4,**BI[6],target	Branch if not unordered (**bclrl** with comparison conditions and LR update)
bso crS target	**bc 12,**BI[6]**,**target	Branch if summary overflow (**bc** without comparison conditions or LR updating)
bsoa crS target	**bca 12,**BI[6],target	Branch if summary overflow (**bca** without comparison conditions or LR updating)
bsoctr crS target	**bcctr 12,**BI[6]**,**target	Branch if summary overflow (**bcctr** without comparison conditions and LR updating)
bsoctrl crS target	**bcctrl 12,**BI[6]**,**target	Branch if summary overflow (**bcctrl** with comparison conditions and LR update)
bsol crS target	**bcl 12,**BI[6],target	Branch if summary overflow (**bcl** with comparison conditions and LR updating)
bsola crS target	**bcla 12,**BI[6]**,**target	Branch if summary overflow (**bcla** with comparison conditions and LR updating)
bsolr crS target	**bclr 12,**BI[6],target	Branch if summary overflow (**bclr** without comparison conditions and LR updating)

Table C-29. Simplified Mnemonics (continued)

Simplified Mnemonic	Mnemonic	Instruction
bsolrl crS target	**bclrl 12,**BI[6]**,**target	Branch if summary overflow (**bclrl** with comparison conditions and LR update)
bt BI,target	**bc 12,BI,**target	Branch if condition true[3] (**bc** without LR update)
bta BI,target	**bca 12,BI,**target	Branch if condition true[3] (**bca** without LR update)
btctr BI	**bcctr 12,BI**	Branch if condition true [3] (**bcctr** without LR update)
btctrl BI	**bcctrl 12,BI**	Branch if condition true [3] (**bcctrl** with LR Update)
btl BI,target	**bcl 12,BI,**target	Branch if condition true [3] (**bcl** with LR update)
btla BI,target	**bcla 12,BI,**target	Branch if condition true [3] (**bcla** with LR update)
btlr BI	**bclr 12,BI**	Branch if condition true [3] (**bclr** without LR update)
btlrl BI	**bclrl 12,BI**	Branch if condition true [3] (**bclrl** with LR Update)
bun crS target	**bc 12,**BI[6]**,**target	Branch if unordered (**bc** without comparison conditions or LR updating)
buna crS target	**bca 12,**BI[6]**,**target	Branch if unordered (**bca** without comparison conditions or LR updating)
bunctr crS target	**bcctr 12,**BI[6]**,**target	Branch if unordered (**bcctr** without comparison conditions and LR updating)
bunctrl crS target	**bcctrl 12,**BI[6]**,**target	Branch if unordered (**bcctrl** with comparison conditions and LR update)
bunl crS target	**bcl 12,**BI[6]**,**target	Branch if unordered (**bcl** with comparison conditions and LR updating)
bunla crS target	**bcla 12,**BI[6]**,**target	Branch if unordered (**bcla** with comparison conditions and LR updating)
bunlr crS target	**bclr 12,**BI[6]**,**target	Branch if unordered (**bclr** without comparison conditions and LR updating)
bunlrl crS target	**bclrl 12,**BI[6]**,**target	Branch if unordered (**bclrl** with comparison conditions and LR update)
clrlslwi rA**,r**S**,***b***,***n* ($n \le b \le 31$)	**rlwinm r**A**,r**S**,***n***,***b* – *n*,31 – *n*	Clear left and shift left word immediate
clrlwi rA**,r**S**,***n* ($n < 32$)	**rlwinm r**A**,r**S**,0,***n***,31**	Clear left word immediate
clrrwi rA**,r**S**,***n* ($n < 32$)	**rlwinm r**A**,r**S**,0,0,**31 – *n*	Clear right word immediate
cmplw crD**,r**A**,r**B	**cmpl cr**D**,0,r**A**,r**B	Compare logical word
cmplwi crD**,r**A,UIMM	**cmpli cr**D**,0,r**A,UIMM	Compare logical word immediate
cmpw crD**,r**A**,r**B	**cmp cr**D**,0,r**A**,r**B	Compare word
cmpwi crD**,r**A,SIMM	**cmpi cr**D**,0,r**A,SIMM	Compare word immediate
crclr bx	**crxor** bx,bx,bx	Condition register clear
crmove bx,by	**cror** bx,by,by	Condition register move
crnot bx,by	**crnor** bx,by,by	Condition register not
crset bx	**creqv** bx,bx,bx	Condition register set
evmr rD**,r**A	**evor r**D**,r**A**,r**A	Vector Move Register
evnot rD**,r**A	**evnor r**D**,r**A**,r**A	Vector Complement Register

Table C-29. Simplified Mnemonics (continued)

Simplified Mnemonic	Mnemonic	Instruction
evsubiw rD,rB,UIMM	**evsubifw** rD,UIMM,rB	Vector subtract word immediate
evsubw rD,rB,rA	**evsubfw** rD,rA,rB	Vector subtract word
extlwi rA,rS,*n*,*b* (*n* > 0)	**rlwinm** rA,rS,*b*,**0**,*n* – 1	Extract and left justify word immediate
extrwi rA,rS,*n*,*b* (*n* > 0)	**rlwinm** rA,rS,*b* + *n*, 32 – *n*,**31**	Extract and right justify word immediate
inslwi rA,rS,*n*,*b* (*n* > 0)	**rlwimi** rA,rS,32 – *b*,*b*,(*b* + *n*) – 1	Insert from left word immediate
insrwi rA,rS,*n*,*b* (*n* > 0)	**rlwimi** rA,rS,32 – (*b* + *n*),*b*,(*b* + *n*) – 1	Insert from right word immediate
iseleq rD,rA,rB	**isel** rD,rA,rB,**2**	Integer Select Equal
iselgt rD,rA,rB	**isel** rD,rA,rB,**1**	Integer Select Greater Than
isellt rD,rA,rB	**isel** rD,rA,rB,**0**	Integer Select Less Than
la rD,d(rA)	**addi** rD,rA,d	Load address
li rD,value	**addi** rD,**0**,value	Load immediate
lis rD,value	**addis** rD,**0**,value	Load immediate signed
mf*spr* rD	**mfspr** rD,SPRN	Move from SPR (see Section C.8, “Simplified Mnemonics for Accessing SPRs.”)
mr rA,rS	**or** rA,rS,rS	Move register
mtcr rS	**mtcrf** 0xFF,rS	Move to Condition Register
mt*spr* rS	**mfspr** SPRN,rS	Move to SPR (see Section C.8, “Simplified Mnemonics for Accessing SPRs.”)
nop	**ori 0,0,0**	No-op
not rA,rS	**nor** rA,rS,rS	NOT
not rA,rS	**nor** rA,rS,rS	Complement register
rotlw rA,rS,rB	**rlwnm** rA,rS,rB,**0,31**	Rotate left word
rotlwi rA,rS,*n*	**rlwinm** rA,rS,*n*,**0,31**	Rotate left word immediate
rotrwi rA,rS,*n*	**rlwinm** rA,rS,32 – *n*,**0,31**	Rotate right word immediate
slwi rA,rS,*n* (*n* < 32)	**rlwinm** rA,rS,*n*,**0**,31 – *n*	Shift left word immediate
srwi rA,rS,*n* (*n* < 32)	**rlwinm** rA,rS,32 – *n*,*n*,**31**	Shift right word immediate
sub rD,rA,rB	**subf** rD,rB,rA	Subtract from
subc rD,rA,rB	**subfc** rD,rB,rA	Subtract from carrying
subi rD,rA,value	**addi** rD,rA,–value	Subtract immediate
subic rD,rA,value	**addic** rD,rA,–value	Subtract immediate carrying
subic. rD,rA,value	**addic.** rD,rA,–value	Subtract immediate carrying
subis rD,rA,value	**addis** rD,rA,–value	Subtract immediate signed
tweq rA,SIMM	**tw 4**,rA,SIMM	Trap if equal
tweqi rA,SIMM	**twi 4**,rA,SIMM	Trap immediate if equal
twge rA,SIMM	**tw 12**,rA,SIMM	Trap if greater than or equal
twgei rA,SIMM	**twi 12**,rA,SIMM	Trap immediate if greater than or equal
twgt rA,SIMM	**tw 8**,rA,SIMM	Trap if greater than

Table C-29. Simplified Mnemonics (continued)

Simplified Mnemonic	Mnemonic	Instruction
twgti rA,SIMM	**twi 8,r**A,SIMM	Trap immediate if greater than
twle rA,SIMM	**tw 20,r**A,SIMM	Trap if less than or equal
twlei rA,SIMM	**twi 20,r**A,SIMM	Trap immediate if less than or equal
twlge rA,SIMM	**tw 12,r**A,SIMM	Trap if logically greater than or equal
twlgei rA,SIMM	**twi 12,r**A,SIMM	Trap immediate if logically greater than or equal
twlgt rA,SIMM	**tw 1,r**A,SIMM	Trap if logically greater than
twlgti rA,SIMM	**twi 1,r**A,SIMM	Trap immediate if logically greater than
twlle rA,SIMM	**tw 6,r**A,SIMM	Trap if logically less than or equal
twllei rA,SIMM	**twi 6,r**A,SIMM	Trap immediate if logically less than or equal
twllt rA,SIMM	**tw 2,r**A,SIMM	Trap if logically less than
twllti rA,SIMM	**twi 2,r**A,SIMM	Trap immediate if logically less than
twlng rA,SIMM	**tw 6,r**A,SIMM	Trap if logically not greater than
twlngi rA,SIMM	**twi 6,r**A,SIMM	Trap immediate if logically not greater than
twlnl rA,SIMM	**tw 5,r**A,SIMM	Trap if logically not less than
twlnli rA,SIMM	**twi 5,r**A,SIMM	Trap immediate if logically not less than
twlt rA,SIMM	**tw 16,r**A,SIMM	Trap if less than
twlti rA,SIMM	**twi 16,r**A,SIMM	Trap immediate if less than
twne rA,SIMM	**tw 24,r**A,SIMM	Trap if not equal
twnei rA,SIMM	**twi 24,r**A,SIMM	Trap immediate if not equal
twng rA,SIMM	**tw 20,r**A,SIMM	Trap if not greater than
twngi rA,SIMM	**twi 20,r**A,SIMM	Trap immediate if not greater than
twnl rA,SIMM	**tw 12,r**A,SIMM	Trap if not less than
twnli rA,SIMM	**twi 12,r**A,SIMM	Trap immediate if not less than

1 Simplified mnemonics for branch instructions that do not test a CR bit should not specify one; a programming error may occur.

2 The value in the BI operand selects CR*n*[2], the EQ bit.

3 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 C.4.6, "Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO and Replaces BI with crS)."

4 The value in the BI operand selects CR*n*[0], the LT bit.

5 The value in the BI operand selects CR*n*[1], the GT bit.

6 The value in the BI operand selects CR*n*[3], the SO bit.

Appendix D
Opcode Listings

This appendix lists instructions as follows:

- Table D-1 lists opcodes alphabetically by mnemonic. It also includes simplified mnemonics showing the syntax for their standard mnemonic equivalents.
- Table D-2 lists opcodes in numerical order, showing both the decimal and the hexadecimal value for the primary opcodes.
- Table D-3 lists opcodes by form, showing the opcodes in binary.

D.1 Instructions (Binary) by Mnemonic

Table D-1 lists e500 instructions by mnemonic.

Table D-1. Instructions (Binary) by Mnemonic

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31	Form	Mnemonic
add	0 1 1 1 1 1	rD	rA	rB	0 1 0 0 0 0 1 0 1 0	0	X	**add**
add.	0 1 1 1 1 1	rD	rA	rB	0 1 0 0 0 0 1 0 1 0	1	X	**add.**
addc	0 1 1 1 1 1	rD	rA	rB	0 0 0 0 0 0 1 0 1 0	0	X	**addc**
addc.	0 1 1 1 1 1	rD	rA	rB	0 0 0 0 0 0 1 0 1 0	1	X	**addc.**
addco	0 1 1 1 1 1	rD	rA	rB	1 0 0 0 0 0 1 0 1 0	0	X	**addco**
addco.	0 1 1 1 1 1	rD	rA	rB	1 0 0 0 0 0 1 0 1 0	1	X	**addco.**
adde	0 1 1 1 1 1	rD	rA	rB	0 0 1 0 0 0 1 0 1 0	0	X	**adde**
adde.	0 1 1 1 1 1	rD	rA	rB	0 0 1 0 0 0 1 0 1 0	1	X	**adde.**
addeo	0 1 1 1 1 1	rD	rA	rB	1 0 1 0 0 0 1 0 1 0	0	X	**addeo**
addeo.	0 1 1 1 1 1	rD	rA	rB	1 0 1 0 0 0 1 0 1 0	1	X	**addeo.**
addi	0 0 1 1 1 0	rD	rA	SIMM			D	**addi**
addic	0 0 1 1 0 0	rD	rA	SIMM			D	**addic**
addic.	0 0 1 1 0 1	rD	rA	SIMM			D	**addic.**
addis	0 0 1 1 1 1	rD	rA	SIMM			D	**addis**
addme	0 1 1 1 1 1	rD	rA	///	0 0 1 1 1 0 1 0 1 0	0	X	**addme**
addme.	0 1 1 1 1 1	rD	rA	///	0 0 1 1 1 0 1 0 1 0	1	X	**addme.**
addmeo	0 1 1 1 1 1	rD	rA	///	1 0 1 1 1 0 1 0 1 0	0	X	**addmeo**
addmeo.	0 1 1 1 1 1	rD	rA	///	1 0 1 1 1 0 1 0 1 0	1	X	**addmeo.**

Table D-1. Instructions (Binary) by Mnemonic

Mnemonic	0–5	6–10	11–15	16–20	21–29	30	31	Form	Mnemonic
addo	0 1 1 1 1 1	rD	rA	rB	1 1 0 0 0 0 1 0 1	0	0	X	**addo**
addo.	0 1 1 1 1 1	rD	rA	rB	1 1 0 0 0 0 1 0 1	0	1	X	**addo.**
addze	0 1 1 1 1 1	rD	rA	///	0 0 1 1 0 0 1 0 1	0	0	X	**addze**
addze.	0 1 1 1 1 1	rD	rA	///	0 0 1 1 0 0 1 0 1	0	1	X	**addze.**
addzeo	0 1 1 1 1 1	rD	rA	///	1 0 1 1 0 0 1 0 1	0	0	X	**addzeo**
addzeo.	0 1 1 1 1 1	rD	rA	///	1 0 1 1 0 0 1 0 1	0	1	X	**addzeo.**
and	0 1 1 1 1 1	rS	rA	rB	0 0 0 0 0 1 1 1 0	0	0	X	**and**
and.	0 1 1 1 1 1	rS	rA	rB	0 0 0 0 0 1 1 1 0	0	1	X	**and.**
andc	0 1 1 1 1 1	rS	rA	rB	0 0 0 0 1 1 1 1 0	0	0	X	**andc**
andc.	0 1 1 1 1 1	rS	rA	rB	0 0 0 0 1 1 1 1 0	0	1	X	**andc.**
andi.	0 1 1 1 0 0	rS	rA	UIMM				D	**andi.**
andis.	0 1 1 1 0 1	rS	rA	UIMM				D	**andis.**
b	0 1 0 0 1 0	LI				0	0	I	**b**
ba	0 1 0 0 1 0	LI				1	0	I	**ba**
bbelr	0 1 1 1 1 1	///			1 0 0 0 1 0 0 1 1	0	0	X	**bbelr**
bblels	0 1 1 1 1 1	///			1 0 0 0 1 0 0 1 1	0	0	X	**bblels**
bc	0 1 0 0 0 0	BO	BI	BD		0	0	B	**bc**
bca	0 1 0 0 0 0	BO	BI	BD		1	0	B	**bca**
bcctr	0 1 0 0 1 1	BO	BI	///	1 0 0 0 0 1 0 0 0	0	0	XL	**bcctr**
bcctrl	0 1 0 0 1 1	BO	BI	///	1 0 0 0 0 1 0 0 0	0	1	XL	**bcctrl**
bcl	0 1 0 0 0 0	BO	BI	BD		0	1	B	**bcl**
bcla	0 1 0 0 0 0	BO	BI	BD		1	1	B	**bcla**
bclr	0 1 0 0 1 1	BO	BI	///	0 0 0 0 0 1 0 0 0	0	0	XL	**bclr**
bclrl	0 1 0 0 1 1	BO	BI	///	0 0 0 0 0 1 0 0 0	0	1	XL	**bclrl**

Mnemonic				Mnemonic
bctr	**bctr** [1]	equivalent to	**bcctr 20,0**	**bctr**
bctrl	**bctrl** [1]	equivalent to	**bcctrl 20,0**	**bctrl**
bdnz	**bdnz** target [1]	equivalent to	**bc 16,0,**target	**bdnz**
bdnza	**bdnza** target [1]	equivalent to	**bca 16,0,**target	**bdnza**
bdnzf	**bdnzf BI,**target	equivalent to	**bc 0,BI,**target	**bdnzf**
bdnzfa	**bdnzfa BI,**target	equivalent to	**bca 0,BI,**target	**bdnzfa**
bdnzfl	**bdnzfl BI,**target	equivalent to	**bcl 0,BI,**target	**bdnzfl**
bdnzfla	**bdnzfla BI,**target	equivalent to	**bcla 0,BI,**target	**bdnzfla**
bdnzflr	**bdnzflr BI**	equivalent to	**bclr 0,BI**	**bdnzflr**

Table D-1. Instructions (Binary) by Mnemonic

Mnemonic	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31			Form	Mnemonic
bdnzflrl	**bdnzflrl BI**	equivalent to	**bclrl 0,BI**		**bdnzflrl**
bdnzl	**bdnzl** target [1]	equivalent to	**bcl 16,0,**target		**bdnzl**
bdnzla	**bdnzla** target [1]	equivalent to	**bcla 16,0,**target		**bdnzla**
bdnzlr	**bdnzlr BI**	equivalent to	**bclr 16,BI**		**bdnzlr**
bdnzlrl	**bdnzlrl** [1]	equivalent to	**bclrl 16,0**		**bdnzlrl**
bdnzt	**bdnzt BI,**target	equivalent to	**bc 8,BI,**target		**bdnzt**
bdnzta	**bdnzta BI,**target	equivalent to	**bca 8,BI,**target		**bdnzta**
bdnztl	**bdnztl BI,**target	equivalent to	**bcl 8,0,**target		**bdnztl**
bdnztla	**bdnztla BI,**target	equivalent to	**bcla 8,BI,**target		**bdnztla**
bdnztlr	**bdnztlr BI**	equivalent to	**bclr 8,BI**		**bdnztlr**
bdnztlr	**bdnztlr BI**	equivalent to	**bclr 8,BI**		**bdnztlr**
bdnztlrl	**bdnztlrl BI**	equivalent to	**bclrl 8,BI**		**bdnztlrl**
bdz	**bdz** target [1]	equivalent to	**bc 18,0,**target		**bdz**
bdza	**bdza** target [1]	equivalent to	**bca 18,0,**target		**bdza**
bdzf	**bdzf BI,**target	equivalent to	**bc 2,BI,**target		**bdzf**
bdzfa	**bdzfa BI,**target	equivalent to	**bca 2,BI,**target		**bdzfa**
bdzfl	**bdzfl BI,**target	equivalent to	**bcl 2,BI,**target		**bdzfl**
bdzfla	**bdzfla BI,**target	equivalent to	**bcla 2,BI,**target		**bdzfla**
bdzflr	**bdzflr BI**	equivalent to	**bclr 2,BI**		**bdzflr**
bdzflrl	**bdzflrl BI**	equivalent to	**bclrl 2,BI**		**bdzflrl**
bdzl	**bdzl** target [1]	equivalent to	**bcl 18,BI,**target		**bdzl**
bdzla	**bdzla** target [1]	equivalent to	**bcla 18,BI,**target		**bdzla**
bdzlr	**bdzlr** [1]	equivalent to	**bclr 18,0**		**bdzlr**
bdzlrl	**bdzlrl** [1]	equivalent to	**bclrl 18,0**		**bdzlrl**
bdzt	**bdzt BI,**target	equivalent to	**bc 10,BI,**target		**bdzt**
bdzta	**bdzta BI,**target	equivalent to	**bca 10,BI,**target		**bdzta**
bdztl	**bdztl BI,**target	equivalent to	**bcl 10,BI,**target		**bdztl**
bdztla	**bdztla BI,**target	equivalent to	**bcla 10,BI,**target		**bdztla**
bdztlrl	**bdztlrl BI**	equivalent to	**bclrl 10, BI**		**bdztlrl**
beq	**beq cr**S,target	equivalent to	**bc 12,**BI[2]**,**target		**beq**
beqa	**beqa cr**S,target	equivalent to	**bca 12,**BI[2]**,**target		**beqa**
beqctr	**beqctr cr**S,target	equivalent to	**bcctr 12,**BI[2]**,**target		**beqctr**
beqctrl	**beqctrl cr**S,target	equivalent to	**bcctrl 12,**BI[2]**,**target		**beqctrl**

Table D-1. Instructions (Binary) by Mnemonic

Mnemonic	0 1 2 3 4 5	6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29	30	31	Form	Mnemonic
beql	**beql cr**S,target	equivalent to **bcl 12**,BI[2],target				**beql**
beqla	**beqla cr**S,target	equivalent to **bcla 12**,BI[2],target				**beqla**
beqlr	**beqlr cr**S,target	equivalent to **bclr 12**,BI[2],target				**beqlr**
beqlrl	**beqlrl cr**S,target	equivalent to **bclrl 12**,BI[2],target				**beqlrl**
bf	**bf BI**,target	equivalent to **bc 4,BI**,target				**bf**
bfa	**bfa BI**,target	equivalent to **bca 4,BI**,target				**bfa**
bfctr	**bfctr BI**	equivalent to **bcctr 4,BI**				**bfctr**
bfctrl	**bfctrl BI**	equivalent to **bcctrl 4,BI**				**bfctrl**
bfl	**bfl BI**,target	equivalent to **bcl 4,BI**,target				**bfl**
bfla	**bfla BI**,target	equivalent to **bcla 4,BI**,target				**bfla**
bflr	**bflr BI**	equivalent to **bclr 4,BI**				**bflr**
bflrl	**bflrl BI**	equivalent to **bclrl 4,BI**				**bflrl**
bge	**bge cr**S,target	equivalent to **bc 4**,BI[3],target				**bge**
bgea	**bgea cr**S,target	equivalent to **bca 4**,BI[3],target				**bgea**
bgectr	**bgectr cr**S,target	equivalent to **bcctr 4**,BI[3],target				**bgectr**
bgectrl	**bgectrl cr**S,target	equivalent to **bcctrl 4**,BI[3],target				**bgectrl**
bgel	**bgel cr**S,target	equivalent to **bcl 4**,BI[3],target				**bgel**
bgela	**bgela cr**S,target	equivalent to **bcla 4**,BI[3],target				**bgela**
bgelr	**bgelr cr**S,target	equivalent to **bclr 4**,BI[3],target				**bgelr**
bgelrl	**bgelrl cr**S,target	equivalent to **bclrl 4**,BI[3],target				**bgelrl**
bgt	**bgt cr**S,target	equivalent to **bc 12**,BI[4],target				**bgt**
bgta	**bgta cr**S,target	equivalent to **bca 12**,BI[4],target				**bgta**
bgtctr	**bgtctr cr**S,target	equivalent to **bcctr 12**,BI[4],target				**bgtctr**
bgtctrl	**bgtctrl cr**S,target	equivalent to **bcctrl 12**,BI[4],target				**bgtctrl**
bgtl	**bgtl cr**S,target	equivalent to **bcl 12**,BI[4],target				**bgtl**
bgtla	**bgtla cr**S,target	equivalent to **bcla 12**,BI[4],target				**bgtla**
bgtlr	**bgtlr cr**S,target	equivalent to **bclr 12**,BI[4],target				**bgtlr**
bgtlrl	**bgtlrl cr**S,target	equivalent to **bclrl 12**,BI[4],target				**bgtlrl**
bl	0 1 0 0 1 0	LI	0	1	I	**bl**
bla	0 1 0 0 1 0	LI	1	1	I	**bla**
ble	**ble cr**S,target	equivalent to **bc 4**,BI[4],target				**ble**
blea	**blea cr**S,target	equivalent to **bca 4**,BI[4],target				**blea**
blectr	**blectr cr**S,target	equivalent to **bcctr 4**,BI[4],target				**blectr**

Table D-1. Instructions (Binary) by Mnemonic

Mnemonic	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31			Form	Mnemonic
blectrl	**blectrl cr**S,target	equivalent to	**bcctrl 4,**BI[4]**,**target		**blectrl**
blel	**blel cr**S,target	equivalent to	**bcl 4,**BI[4],target		**blel**
blela	**blela cr**S,target	equivalent to	**bcla 4,**BI[4],target		**blela**
blelr	**blelr cr**S,target	equivalent to	**bclr 4,**BI[4],target		**blelr**
blelrl	**blelrl cr**S,target	equivalent to	**bclrl 4,**BI[4],target		**blelrl**
blr	**blr** [1]	equivalent to	**bclr 20,0**		**blr**
blrl	**blrl** [1]	equivalent to	**bclrl 20,0**		**blrl**
blt	**blt cr**S,target	equivalent to	**bc 12,**BI,target		**blt**
blta	**blta cr**S,target	equivalent to	**bca 12,**BI[3],target		**blta**
bltctr	**bltctr cr**S,target	equivalent to	**bcctr 12,**BI[3],target		**bltctr**
bltctrl	**bltctrl cr**S,target	equivalent to	**bcctrl 12,**BI[3],target		**bltctrl**
bltl	**bltl cr**S,target	equivalent to	**bcl 12,**BI[3],target		**bltl**
bltla	**bltla cr**S,target	equivalent to	**bcla 12,**BI[3],target		**bltla**
bltlr	**bltlr cr**S,target	equivalent to	**bclr 12,**BI[3],target		**bltlr**
bltlrl	**bltlrl cr**S,target	equivalent to	**bclrl 12,**BI[3],target		**bltlrl**
bne	**bne cr**S,target	equivalent to	**bc 4,**BI[3],target		**bne**
bnea	**bnea cr**S,target	equivalent to	**bca 4,**BI[3],target		**bnea**
bnectr	**bnectr cr**S,target	equivalent to	**bcctr 4,**BI[3],target		**bnectr**
bnectrl	**bnectrl cr**S,target	equivalent to	**bcctrl 4,**BI[3],target		**bnectrl**
bnel	**bnel cr**S,target	equivalent to	**bcl 4,**BI[3],target		**bnel**
bnela	**bnela cr**S,target	equivalent to	**bcla 4,**BI[3],target		**bnela**
bnelr	**bnelr cr**S,target	equivalent to	**bclr 4,**BI[3],target		**bnelr**
bnelrl	**bnelrl cr**S,target	equivalent to	**bclrl 4,**BI[3],target		**bnelrl**
bng	**bng cr**S,target	equivalent to	**bc 4,**BI[4],target		**bng**
bnga	**bnga cr**S,target	equivalent to	**bca 4,**BI[4],target		**bnga**
bngctr	**bngctr cr**S,target	equivalent to	**bcctr 4,**BI[4],target		**bngctr**
bngctrl	**bngctrl cr**S,target	equivalent to	**bcctrl 4,**BI[4],target		**bngctrl**
bngl	**bngl cr**S,target	equivalent to	**bcl 4,**BI[4],target		**bngl**
bngla	**bngla cr**S,target	equivalent to	**bcla 4,**BI[4],target		**bngla**
bnglr	**bnglr cr**S,target	equivalent to	**bclr 4,**BI[4],target		**bnglr**
bnglrl	**bnglrl cr**S,target	equivalent to	**bclrl 4,**BI[4],target		**bnglrl**
bnl	**bnl cr**S,target	equivalent to	**bc 4,**BI[3],target		**bnl**
bnla	**bnla cr**S,target	equivalent to	**bca 4,**BI[3],target		**bnla**

Table D-1. Instructions (Binary) by Mnemonic

Mnemonic	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31					Form	Mnemonic
bnlctr	**bnlctr cr**S,target		equivalent to	**bcctr 4,**BI[3]**,**target			**bnlctr**
bnlctrl	**bnlctrl cr**S,target		equivalent to	**bcctrl 4,**BI[3]**,**target			**bnlctrl**
bnll	**bnll cr**S,target		equivalent to	**bcl 4,**BI[3]**,**target			**bnll**
bnlla	**bnlla cr**S,target		equivalent to	**bcla 4,**BI[3]**,**target			**bnlla**
bnllr	**bnllr cr**S,target		equivalent to	**bclr 4,**BI[3]**,**target			**bnllr**
bnllrl	**bnllrl cr**S,target		equivalent to	**bclrl 4,**BI[3]**,**target			**bnllrl**
bns	**bns cr**S,target		equivalent to	**bc 4,**BI[5]**,**target			**bns**
bnsa	**bnsa cr**S,target		equivalent to	**bca 4,**BI[5]**,**target			**bnsa**
bnsctr	**bnsctr cr**S,target		equivalent to	**bcctr 4,**BI[5]**,**target			**bnsctr**
bnsctrl	**bnsctrl cr**S,target		equivalent to	**bcctrl 4,**BI[5]**,**target			**bnsctrl**
bnsl	**bnsl cr**S,target		equivalent to	**bcl 4,**BI[5]**,**target			**bnsl**
bnsla	**bnsla cr**S,target		equivalent to	**bcla 4,**BI[5]**,**target			**bnsla**
bnslr	**bnslr cr**S,target		equivalent to	**bclr 4,**BI[5]**,**target			**bnslr**
bnslrl	**bnslrl cr**S,target		equivalent to	**bclrl 4,**BI[5]**,**target			**bnslrl**
bnu	**bnu cr**S,target		equivalent to	**bc 4,**BI[5]**,**target			**bnu**
bnua	**bnua cr**S,target		equivalent to	**bca 4,**BI[5]**,**target			**bnua**
bnuctr	**bnuctr cr**S,target		equivalent to	**bcctr 4,**BI[5]**,**target			**bnuctr**
bnuctrl	**bnuctrl cr**S,target		equivalent to	**bcctrl 4,**BI[5]**,**target			**bnuctrl**
bnul	**bnul cr**S,target		equivalent to	**bcl 4,**BI[5]**,**target			**bnul**
bnula	**bnula cr**S,target		equivalent to	**bcla 4,**BI[5]**,**target			**bnula**
bnulr	**bnulr cr**S,target		equivalent to	**bclr 4,**BI[5]**,**target			**bnulr**
bnulrl	**bnulrl cr**S,target		equivalent to	**bclrl 4,**BI[5]**,**target			**bnulrl**
brinc	0 0 0 1 0 0	rD	rA	rB	0 1 0 0 0 0 0 1 1 1 1	EVX	**brinc**
bso	**bso cr**S,target		equivalent to	**bc 12,**BI[5]**,**target			**bso**
bsoa	**bsoa cr**S,target		equivalent to	**bca 12,**BI[5]**,**target			**bsoa**
bsoctr	**bsoctr cr**S,target		equivalent to	**bcctr 12,**BI[5]**,**target			**bsoctr**
bsoctrl	**bsoctrl cr**S,target		equivalent to	**bcctrl 12,**BI[5]**,**target			**bsoctrl**
bsol	**bsol cr**S,target		equivalent to	**bcl 12,**BI[5]**,**target			**bsol**
bsola	**bsola cr**S,target		equivalent to	**bcla 12,**BI[5]**,**target			**bsola**
bsolr	**bsolr cr**S,target		equivalent to	**bclr 12,**BI[5]**,**target			**bsolr**
bsolrl	**bsolrl cr**S,target		equivalent to	**bclrl 12,**BI[5]**,**target			**bsolrl**
bt	**bt BI,**target		equivalent to	**bc 12,BI,**target			**bt**
bta	**bta BI,**target		equivalent to	**bca 12,BI,**target			**bta**

Table D-1. Instructions (Binary) by Mnemonic

Mnemonic	0–5	6–10	11–15	16–20	21–30	31	Form	Mnemonic
btctr	**btctr BI**	equivalent to	**bcctr 12,BI**					**btctr**
btctrl	**btctrl BI**	equivalent to	**bcctrl 12,BI**					**btctrl**
btl	**btl BI,**target	equivalent to	**bcl 12,BI,**target					**btl**
btla	**btla BI,**target	equivalent to	**bcla 12,BI,**target					**btla**
btlr	**btlr BI**	equivalent to	**bclr 12,BI**					**btlr**
btlrl	**btlrl BI**	equivalent to	**bclrl 12,BI**					**btlrl**
bun	**bun cr**S,target	equivalent to	**bc 12,**BI[5]**,**target					**bun**
buna	**buna cr**S,target	equivalent to	**bca 12,**BI[5]**,**target					**buna**
bunctr	**bunctr cr**S,target	equivalent to	**bcctr 12,**BI[5]**,**target					**bunctr**
bunctrl	**bunctrl cr**S,target	equivalent to	**bcctrl 12,**BI[5]**,**target					**bunctrl**
bunl	**bunl cr**S,target	equivalent to	**bcl 12,**BI[5]**,**target					**bunl**
bunla	**bunla cr**S,target	equivalent to	**bcla 12,**BI[5]**,**target					**bunla**
bunlr	**bunlr cr**S,target	equivalent to	**bclr 12,**BI[5]**,**target					**bunlr**
bunlrl	**bunlrl cr**S,target	equivalent to	**bclrl 12,**BI[5]**,**target					**bunlrl**
clrlslwi	**clrlslwi r**A,**r**S,*b*,*n* (*n* ≤ *b* ≤ 31)	equivalent to	**rlwinm r**A,**r**S,*n*,*b* – *n*,31 – *n*					**clrlslwi**
clrlwi	**clrlwi r**A,**r**S,*n* (n < 32)	equivalent to	**rlwinm r**A,**r**S,**0,***n***,31**					**clrlwi**
clrrwi	**clrrwi r**A,**r**S,*n* (n < 32)	equivalent to	**rlwinm r**A,**r**S,**0,0,**31 – *n*					**clrrwi**
cmp	0 1 1 1 1 1	crfD / L	rA	rB	0 0 0 0 0 0 0 0 0 0	/	X	**cmp**
cmpi	0 0 1 0 1 1	crfD / L	rA	SIMM			D	**cmpi**
cmpl	0 1 1 1 1 1	crfD / L	rA	rB	0 0 0 0 1 0 0 0 0 0	/	X	**cmpl**
cmpli	0 0 1 0 1 0	crfD / L	rA	UIMM			D	**cmpli**
cmplw	**cmplw cr**D,**r**A,**r**B	equivalent to	**cmpl cr**D,**0**,**r**A,**r**B					**cmplw**
cmplwi	**cmplwi cr**D,**r**A,**UIMM**	equivalent to	**cmpli cr**D,**0**,**r**A,UIMM					**cmplwi**
cmpw	**cmpw cr**D,**r**A,**r**B	equivalent to	**cmp cr**D,**0**,**r**A,**r**B					**cmpw**
cmpwi	**cmpwi cr**D,**r**A,**SIMM**	equivalent to	**cmpi cr**D,**0**,**r**A,SIMM					**cmpwi**
cntlzw	0 1 1 1 1 1	rS	rA	///	0 0 0 0 0 1 1 0 1 0	0	X	**cntlzw**
cntlzw.	0 1 1 1 1 1	rS	rA	///	0 0 0 0 0 1 1 0 1 0	1	X	**cntlzw.**
crand	0 1 0 0 1 1	crbD	crbA	crbB	0 1 0 0 0 0 0 0 0 1	/	XL	**crand**
crandc	0 1 0 0 1 1	crbD	crbA	crbB	0 0 1 0 0 0 0 0 0 1	/	XL	**crandc**
crclr	**crclr bx**	equivalent to	**crxor** bx,bx,bx					**crclr**
creqv	0 1 0 0 1 1	crbD	crbA	crbB	0 1 0 0 1 0 0 0 0 1	/	XL	**creqv**
crmove	**crmove bx,by**	equivalent to	**cror** bx,by,by					**crmove**
crnand	0 1 0 0 1 1	crbD	crbA	crbB	0 0 1 1 1 0 0 0 0 1	/	XL	**crnand**

Table D-1. Instructions (Binary) by Mnemonic

Mnemonic	0–5	6–10	11–15	16–20	21–30	31	Form	Mnemonic
crnor	0 1 0 0 1 1	crbD	crbA	crbB	0 0 0 0 1 0 0 0 0 1	/	XL	**crnor**
crnot	**crnot bx,by**		equivalent to	**crnor** bx,by,by				**crnot**
cror	0 1 0 0 1 1	crbD	crbA	crbB	0 1 1 1 0 0 0 0 0 1	/	XL	**cror**
crorc	0 1 0 0 1 1	crbD	crbA	crbB	0 1 1 0 1 0 0 0 0 1	/	XL	**crorc**
crset	**crset bx**		equivalent to	**creqv** bx,bx,bx				**crset**
crxor	0 1 0 0 1 1	crbD	crbA	crbB	0 0 1 1 0 0 0 0 0 1	/	XL	**crxor**
dcba	0 1 1 1 1 1	///	rA	rB	1 0 1 1 1 1 0 1 1 0	/	X	**dcba**
dcbf	0 1 1 1 1 1	///	rA	rB	0 0 0 1 0 1 0 1 1 0	/	X	**dcbf**
dcbi	0 1 1 1 1 1	///	rA	rB	0 1 1 1 0 1 0 1 1 0	/	X	**dcbi**
dcblc	0 1 1 1 1 1	CT	rA	rB	0 1 1 0 0 0 0 1 1 0	0	X	**dcblc**
dcbst	0 1 1 1 1 1	///	rA	rB	0 0 0 0 1 1 0 1 1 0	/	X	**dcbst**
dcbt	0 1 1 1 1 1	CT	rA	rB	0 1 0 0 0 1 0 1 1 0	/	X	**dcbt**
dcbtls	0 1 1 1 1 1	CT	rA	rB	0 0 1 0 1 0 0 1 1 0	0	X	**dcbtls**
dcbtst	0 1 1 1 1 1	CT	rA	rB	0 0 1 1 1 1 0 1 1 0	/	X	**dcbtst**
dcbtstls	0 1 1 1 1 1	CT	rA	rB	0 0 1 0 0 0 0 1 1 0	0	X	**dcbtstls**
dcbz	0 1 1 1 1 1	///	rA	rB	1 1 1 1 1 1 0 1 1 0	/	X	**dcbz**
divw	0 1 1 1 1 1	rD	rA	rB	0 1 1 1 1 0 1 0 1 1	0	X	**divw**
divw.	0 1 1 1 1 1	rD	rA	rB	0 1 1 1 1 0 1 0 1 1	1	X	**divw.**
divwo	0 1 1 1 1 1	rD	rA	rB	1 1 1 1 1 0 1 0 1 1	0	X	**divwo**
divwo.	0 1 1 1 1 1	rD	rA	rB	1 1 1 1 1 0 1 0 1 1	1	X	**divwo.**
divwu	0 1 1 1 1 1	rD	rA	rB	0 1 1 1 0 0 1 0 1 1	0	X	**divwu**
divwu.	0 1 1 1 1 1	rD	rA	rB	0 1 1 1 0 0 1 0 1 1	1	X	**divwu.**
divwuo	0 1 1 1 1 1	rD	rA	rB	1 1 1 1 0 0 1 0 1 1	0	X	**divwuo**
divwuo.	0 1 1 1 1 1	rD	rA	rB	1 1 1 1 0 0 1 0 1 1	1	X	**divwuo.**
dss	**dss STRM**		equivalent to	**dss** STRM,**0**				**dss**
efdabs	0 0 0 1 0 0	rD	rA	///	0 1 0 1 1 1 0 0 1 0	0	EFX	**efdabs**
efdadd	0 0 0 1 0 0	rD	rA	rB	0 1 0 1 1 1 0 0 0 0	0	EFX	**efdadd**
efdcfs	0 0 0 1 0 0	rD	0 0 0 0 0	rB	0 1 0 1 1 1 0 1 1 1	1	EFX	**efdcfs**
efdcfsf	0 0 0 1 0 0	rD	///	rB	0 1 0 1 1 1 1 0 0 1	1	EFX	**efdcfsf**
efdcfsi	0 0 0 1 0 0	rD	///	rB	0 1 0 1 1 1 1 0 0 0	1	EFX	**efdcfsi**
efdcfuf	0 0 0 1 0 0	rD	///	rB	0 1 0 1 1 1 1 0 0 1	0	EFX	**efdcfuf**
efdcfui	0 0 0 1 0 0	rD	///	rB	0 1 0 1 1 1 1 0 0 0	0	EFX	**efdcfui**
efdcmpeq	0 0 0 1 0 0	crfD (6–8), / / (9–10)	rA	rB	0 1 0 1 1 1 0 1 1 1	0	EFX	**efdcmpeq**

Table D-1. Instructions (Binary) by Mnemonic

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30 31	Form	Mnemonic
efdcmpgt	0 0 0 1 0 0	crfD / /	rA	rB	0 1 0 1 1 1 0 1 1 0 0	EFX	**efdcmpgt**
efdcmplt	0 0 0 1 0 0	crfD / /	rA	rB	0 1 0 1 1 1 0 1 1 0 1	EFX	**efdcmplt**
efdctsf	0 0 0 1 0 0	rD	///	rB	0 1 0 1 1 1 1 0 1 1 1	EFX	**efdctsf**
efdctsi	0 0 0 1 0 0	rD	///	rB	0 1 0 1 1 1 1 0 1 0 1	EFX	**efdctsi**
efdctsiz	0 0 0 1 0 0	rD	///	rB	0 1 0 1 1 1 1 1 0 1 0	EFX	**efdctsiz**
efdctuf	0 0 0 1 0 0	rD	///	rB	0 1 0 1 1 1 1 0 1 1 0	EFX	**efdctuf**
efdctui	0 0 0 1 0 0	rD	///	rB	0 1 0 1 1 1 1 0 1 0 0	EFX	**efdctui**
efdctuiz	0 0 0 1 0 0	rD	///	rB	0 1 0 1 1 1 1 1 0 0 0	EFX	**efdctuiz**
efddiv	0 0 0 1 0 0	rD	rA	rB	0 1 0 1 1 1 0 1 0 0 1	EFX	**efddiv**
efdmul	0 0 0 1 0 0	rD	rA	rB	0 1 0 1 1 1 0 1 0 0 0	EFX	**efdmul**
efdnabs	0 0 0 1 0 0	rD	rA	///	0 1 0 1 1 1 0 0 1 0 1	EFX	**efdnabs**
efdneg	0 0 0 1 0 0	rD	rA	///	0 1 0 1 1 1 0 0 1 1 0	EFX	**efdneg**
efdsub	0 0 0 1 0 0	rD	rA	rB	0 1 0 1 1 1 0 0 0 0 1	EFX	**efdsub**
efdtsteq	0 0 0 1 0 0	crfD / /	rA	rB	0 1 0 1 1 1 1 1 1 1 0	EFX	**efdtsteq**
efdtstgt	0 0 0 1 0 0	crfD / /	rA	rB	0 1 0 1 1 1 1 1 1 0 0	EFX	**efdtstgt**
efdtstlt	0 0 0 1 0 0	crfD / /	rA	rB	0 1 0 1 1 1 1 1 1 0 1	EFX	**efdtstlt**
efsabs	0 0 0 1 0 0	rD	rA	///	0 1 0 1 1 0 0 0 1 0 0	EFX	**efsabs**
efsadd	0 0 0 1 0 0	rD	rA	rB	0 1 0 1 1 0 0 0 0 0 0	EFX	**efsadd**
efscfd	0 0 0 1 0 0	rD	0 0 0 0 0	rB	0 1 0 1 1 0 0 1 1 1 1	EFX	**efscfd**
efscfsf	0 0 0 1 0 0	rD	///	rB	0 1 0 1 1 0 1 0 0 1 1	EFX	**efscfsf**
efscfsi	0 0 0 1 0 0	rD	///	rB	0 1 0 1 1 0 1 0 0 0 1	EFX	**efscfsi**
efscfuf	0 0 0 1 0 0	rD	///	rB	0 1 0 1 1 0 1 0 0 1 0	EFX	**efscfuf**
efscfui	0 0 0 1 0 0	rD	///	rB	0 1 0 1 1 0 1 0 0 0 0	EFX	**efscfui**
efscmpeq	0 0 0 1 0 0	crfD / /	rA	rB	0 1 0 1 1 0 0 1 1 1 0	EFX	**efscmpeq**
efscmpgt	0 0 0 1 0 0	crfD / /	rA	rB	0 1 0 1 1 0 0 1 1 0 0	EFX	**efscmpgt**
efscmplt	0 0 0 1 0 0	crfD / /	rA	rB	0 1 0 1 1 0 0 1 1 0 1	EFX	**efscmplt**
efsctsf	0 0 0 1 0 0	rD	///	rB	0 1 0 1 1 0 1 0 1 1 1	EFX	**efsctsf**
efsctsi	0 0 0 1 0 0	rD	///	rB	0 1 0 1 1 0 1 0 1 0 1	EFX	**efsctsi**
efsctsiz	0 0 0 1 0 0	rD	///	rB	0 1 0 1 1 0 1 1 0 1 0	EFX	**efsctsiz**
efsctuf	0 0 0 1 0 0	rD	///	rB	0 1 0 1 1 0 1 0 1 1 0	EFX	**efsctuf**
efsctui	0 0 0 1 0 0	rD	///	rB	0 1 0 1 1 0 1 0 1 0 0	EFX	**efsctui**
efsctuiz	0 0 0 1 0 0	rD	///	rB	0 1 0 1 1 0 1 1 0 0 0	EFX	**efsctuiz**
efsdiv	0 0 0 1 0 0	rD	rA	rB	0 1 0 1 1 0 0 1 0 0 1	EFX	**efsdiv**

Table D-1. Instructions (Binary) by Mnemonic

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30 31	Form	Mnemonic
efsmul	0 0 0 1 0 0	rD	rA	rB	0 1 0 1 1 0 0 1 0 0 0	EFX	**efsmul**
efsnabs	0 0 0 1 0 0	rD	rA	///	0 1 0 1 1 0 0 0 1 0 1	EFX	**efsnabs**
efsneg	0 0 0 1 0 0	rD	rA	///	0 1 0 1 1 0 0 0 1 1 0	EFX	**efsneg**
efssub	0 0 0 1 0 0	rD	rA	rB	0 1 0 1 1 0 0 0 0 0 1	EFX	**efssub**
efststeq	0 0 0 1 0 0	crfD / /	rA	rB	0 1 0 1 1 0 1 1 1 1 0	EFX	**efststeq**
efststgt	0 0 0 1 0 0	crfD / /	rA	rB	0 1 0 1 1 0 1 1 1 0 0	EFX	**efststgt**
efststlt	0 0 0 1 0 0	crfD / /	rA	rB	0 1 0 1 1 0 1 1 1 0 1	EFX	**efststlt**
eqv	0 1 1 1 1 1	rD	rA	rB	0 1 0 0 0 1 1 1 0 0 0	X	**eqv**
eqv.	0 1 1 1 1 1	rD	rA	rB	0 1 0 0 0 1 1 1 0 0 1	X	**eqv.**
evabs	0 0 0 1 0 0	rD	rA	///	0 1 0 0 0 0 0 1 0 0 0	EVX	**evabs**
evaddiw	0 0 0 1 0 0	rD	UIMM	rB	0 1 0 0 0 0 0 0 0 1 0	EVX	**evaddiw**
evaddsmiaaw	0 0 0 1 0 0	rD	rA	///	1 0 0 1 1 0 0 1 0 0 1	EVX	**evaddsmiaaw**
evaddssiaaw	0 0 0 1 0 0	rD	rA	///	1 0 0 1 1 0 0 0 0 0 1	EVX	**evaddssiaaw**
evaddumiaaw	0 0 0 1 0 0	rD	rA	///	1 0 0 1 1 0 0 1 0 0 0	EVX	**evaddumiaaw**
evaddusiaaw	0 0 0 1 0 0	rD	rA	///	1 0 0 1 1 0 0 0 0 0 0	EVX	**evaddusiaaw**
evaddw	0 0 0 1 0 0	rD	rA	rB	0 1 0 0 0 0 0 0 0 0 0	EVX	**evaddw**
evand	0 0 0 1 0 0	rD	rA	rB	0 1 0 0 0 0 1 0 0 0 1	EVX	**evand**
evandc	0 0 0 1 0 0	rD	rA	rB	0 1 0 0 0 0 1 0 0 1 0	EVX	**evandc**
evcmpeq	0 0 0 1 0 0	crfD / /	rA	rB	0 1 0 0 0 1 1 0 1 0 0	EVX	**evcmpeq**
evcmpgts	0 0 0 1 0 0	crfD / /	rA	rB	0 1 0 0 0 1 1 0 0 0 1	EVX	**evcmpgts**
evcmpgtu	0 0 0 1 0 0	crfD / /	rA	rB	0 1 0 0 0 1 1 0 0 0 0	EVX	**evcmpgtu**
evcmplts	0 0 0 1 0 0	crfD / /	rA	rB	0 1 0 0 0 1 1 0 0 1 1	EVX	**evcmplts**
evcmpltu	0 0 0 1 0 0	crfD / /	rA	rB	0 1 0 0 0 1 1 0 0 1 0	EVX	**evcmpltu**
evcntlsw	0 0 0 1 0 0	rD	rA	///	0 1 0 0 0 0 0 1 1 1 0	EVX	**evcntlsw**
evcntlzw	0 0 0 1 0 0	rD	rA	///	0 1 0 0 0 0 0 1 1 0 1	EVX	**evcntlzw**
evdivws	0 0 0 1 0 0	rD	rA	rB	1 0 0 1 1 0 0 0 1 1 0	EVX	**evdivws**
evdivwu	0 0 0 1 0 0	rD	rA	rB	1 0 0 1 1 0 0 0 1 1 1	EVX	**evdivwu**
eveqv	0 0 0 1 0 0	rD	rA	rB	0 1 0 0 0 0 1 1 0 0 1	EVX	**eveqv**
evextsb	0 0 0 1 0 0	rD	rA	///	0 1 0 0 0 0 0 1 0 1 0	EVX	**evextsb**
evextsh	0 0 0 1 0 0	rD	rA	///	0 1 0 0 0 0 0 1 0 1 1	EVX	**evextsh**
evfsabs	0 0 0 1 0 0	rD	rA	///	0 1 0 1 0 0 0 0 1 0 0	EVX	**evfsabs**
evfsadd	0 0 0 1 0 0	rD	rA	rB	0 1 0 1 0 0 0 0 0 0 0	EVX	**evfsadd**
evfscfsf	0 0 0 1 0 0	rD	///	rB	0 1 0 1 0 0 1 0 0 1 1	EVX	**evfscfsf**

Table D-1. Instructions (Binary) by Mnemonic

Mnemonic	0 1 2 3 4 5	6 7 8	9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30 31	Form	Mnemonic
evfscfsi	0 0 0 1 0 0	rD		///	rB	0 1 0 1 0 0 1 0 0 0 1	EVX	**evfscfsi**
evfscfuf	0 0 0 1 0 0	rD		///	rB	0 1 0 1 0 0 1 0 0 1 0	EVX	**evfscfuf**
evfscfui	0 0 0 1 0 0	rD		///	rB	0 1 0 1 0 0 1 0 0 0 0	EVX	**evfscfui**
evfscmpeq	0 0 0 1 0 0	crfD	/ /	rA	rB	0 1 0 1 0 0 0 1 1 1 0	EVX	**evfscmpeq**
evfscmpgt	0 0 0 1 0 0	crfD	/ /	rA	rB	0 1 0 1 0 0 0 1 1 0 0	EVX	**evfscmpgt**
evfscmplt	0 0 0 1 0 0	crfD	/ /	rA	rB	0 1 0 1 0 0 0 1 1 0 1	EVX	**evfscmplt**
evfsctsf	0 0 0 1 0 0	rD		///	rB	0 1 0 1 0 0 1 0 1 1 1	EVX	**evfsctsf**
evfsctsi	0 0 0 1 0 0	rD		///	rB	0 1 0 1 0 0 1 0 1 0 1	EVX	**evfsctsi**
evfsctsiz	0 0 0 1 0 0	rD		///	rB	0 1 0 1 0 0 1 1 0 1 0	EVX	**evfsctsiz**
evfsctuf	0 0 0 1 0 0	rD		///	rB	0 1 0 1 0 0 1 0 1 1 0	EVX	**evfsctuf**
evfsctui	0 0 0 1 0 0	rD		///	rB	0 1 0 1 0 0 1 0 1 0 0	EVX	**evfsctui**
evfsctuiz	0 0 0 1 0 0	rD		///	rB	0 1 0 1 0 0 1 1 0 0 0	EVX	**evfsctuiz**
evfsdiv	0 0 0 1 0 0	rD		rA	rB	0 1 0 1 0 0 0 1 0 0 1	EVX	**evfsdiv**
evfsmul	0 0 0 1 0 0	rD		rA	rB	0 1 0 1 0 0 0 1 0 0 0	EVX	**evfsmul**
evfsnabs	0 0 0 1 0 0	rD		rA	///	0 1 0 1 0 0 0 0 1 0 1	EVX	**evfsnabs**
evfsneg	0 0 0 1 0 0	rD		rA	///	0 1 0 1 0 0 0 0 1 1 0	EVX	**evfsneg**
evfssub	0 0 0 1 0 0	rD		rA	rB	0 1 0 1 0 0 0 0 0 0 1	EVX	**evfssub**
evfststeq	0 0 0 1 0 0	crfD	/ /	rA	rB	0 1 0 1 0 0 1 1 1 1 0	EVX	**evfststeq**
evfststgt	0 0 0 1 0 0	crfD	/ /	rA	rB	0 1 0 1 0 0 1 1 1 0 0	EVX	**evfststgt**
evfststlt	0 0 0 1 0 0	crfD	/ /	rA	rB	0 1 0 1 0 0 1 1 1 0 1	EVX	**evfststlt**
evldd	0 0 0 1 0 0	rD		rA	UIMM[6]	0 1 1 0 0 0 0 0 0 0 1	EVX	**evldd**
evlddx	0 0 0 1 0 0	rD		rA	rB	0 1 1 0 0 0 0 0 0 0 0	EVX	**evlddx**
evldh	0 0 0 1 0 0	rD		rA	UIMM [6]	0 1 1 0 0 0 0 0 1 0 1	EVX	**evldh**
evldhx	0 0 0 1 0 0	rD		rA	rB	0 1 1 0 0 0 0 0 1 0 0	EVX	**evldhx**
evldw	0 0 0 1 0 0	rD		rA	UIMM [6]	0 1 1 0 0 0 0 0 0 1 1	EVX	**evldw**
evldwx	0 0 0 1 0 0	rD		rA	rB	0 1 1 0 0 0 0 0 0 1 0	EVX	**evldwx**
evlhhesplat	0 0 0 1 0 0	rD		rA	UIMM[7]	0 1 1 0 0 0 0 1 0 0 1	EVX	**evlhhesplat**
evlhhesplatx	0 0 0 1 0 0	rD		rA	rB	0 1 1 0 0 0 0 1 0 0 0	EVX	**evlhhesplatx**
evlhhossplat	0 0 0 1 0 0	rD		rA	UIMM [7]	0 1 1 0 0 0 0 1 1 1 1	EVX	**evlhhossplat**
evlhhossplatx	0 0 0 1 0 0	rD		rA	rB	0 1 1 0 0 0 0 1 1 1 0	EVX	**evlhhossplatx**
evlhhousplat	0 0 0 1 0 0	rD		rA	UIMM [7]	0 1 1 0 0 0 0 1 1 0 1	EVX	**evlhhousplat**
evlhhousplatx	0 0 0 1 0 0	rD		rA	rB	0 1 1 0 0 0 0 1 1 0 0	EVX	**evlhhousplatx**
evlwhe	0 0 0 1 0 0	rD		rA	UIMM[8]	0 1 1 0 0 0 1 0 0 0 1	EVX	**evlwhe**

Table D-1. Instructions (Binary) by Mnemonic

Mnemonic	0	1	2	3	4	5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evlwhex	0	0	0	1	0	0	rD	rA	rB	0	1	1	0	0	0	1	0	0	0	0	EVX	**evlwhex**
evlwhos	0	0	0	1	0	0	rD	rA	UIMM [8]	0	1	1	0	0	0	1	0	1	1	1	EVX	**evlwhos**
evlwhosx	0	0	0	1	0	0	rD	rA	rB	0	1	1	0	0	0	1	0	1	1	0	EVX	**evlwhosx**
evlwhou	0	0	0	1	0	0	rD	rA	UIMM [8]	0	1	1	0	0	0	1	0	1	0	1	EVX	**evlwhou**
evlwhoux	0	0	0	1	0	0	rD	rA	rB	0	1	1	0	0	0	1	0	1	0	0	EVX	**evlwhoux**
evlwhsplat	0	0	0	1	0	0	rD	rA	UIMM [8]	0	1	1	0	0	0	1	1	1	0	1	EVX	**evlwhsplat**
evlwhsplatx	0	0	0	1	0	0	rD	rA	rB	0	1	1	0	0	0	1	1	1	0	0	EVX	**evlwhsplatx**
evlwwsplat	0	0	0	1	0	0	rD	rA	UIMM [8]	0	1	1	0	0	0	1	1	0	0	1	EVX	**evlwwsplat**
evlwwsplatx	0	0	0	1	0	0	rD	rA	rB	0	1	1	0	0	0	1	1	0	0	0	EVX	**evlwwsplatx**
evmergehi	0	0	0	1	0	0	rD	rA	rB	0	1	0	0	0	1	0	1	1	0	0	EVX	**evmergehi**
evmergehilo	0	0	0	1	0	0	rD	rA	rB	0	1	0	0	0	1	0	1	1	1	0	EVX	**evmergehilo**
evmergelo	0	0	0	1	0	0	rD	rA	rB	0	1	0	0	0	1	0	1	1	0	1	EVX	**evmergelo**
evmergelohi	0	0	0	1	0	0	rD	rA	rB	0	1	0	0	0	1	0	1	1	1	1	EVX	**evmergelohi**
evmhegsmfaa	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	0	1	0	1	0	1	1	EVX	**evmhegsmfaa**
evmhegsmfan	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	1	0	1	0	1	1	EVX	**evmhegsmfan**
evmhegsmiaa	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	0	1	0	1	0	0	1	EVX	**evmhegsmiaa**
evmhegsmian	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	1	0	1	0	0	1	EVX	**evmhegsmian**
evmhegumiaa	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	0	1	0	1	0	0	0	EVX	**evmhegumiaa**
evmhegumian	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	1	0	1	0	0	0	EVX	**evmhegumian**
evmhesmf	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	0	0	1	0	1	1	EVX	**evmhesmf**
evmhesmfa	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	1	0	1	0	1	1	EVX	**evmhesmfa**
evmhesmfaaw	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	0	0	0	1	0	1	1	EVX	**evmhesmfaaw**
evmhesmfanw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	0	0	1	0	1	1	EVX	**evmhesmfanw**
evmhesmi	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	0	0	1	0	0	1	EVX	**evmhesmi**
evmhesmia	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	1	0	1	0	0	1	EVX	**evmhesmia**
evmhesmiaaw	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	0	0	0	1	0	0	1	EVX	**evmhesmiaaw**
evmhesmianw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	0	0	1	0	0	1	EVX	**evmhesmianw**
evmhessf	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	0	0	0	0	1	1	EVX	**evmhessf**
evmhessfa	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	1	0	0	0	1	1	EVX	**evmhessfa**
evmhessfaaw	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	0	0	0	0	0	1	1	EVX	**evmhessfaaw**
evmhessfanw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	0	0	0	0	1	1	EVX	**evmhessfanw**
evmhessiaaw	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	0	0	0	0	0	0	1	EVX	**evmhessiaaw**
evmhessianw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	0	0	0	0	0	1	EVX	**evmhessianw**

Table D-1. Instructions (Binary) by Mnemonic

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30 31	Form	Mnemonic
evmheumi	0 0 0 1 0 0	rD	rA	rB	1 0 0 0 0 0 0 1 0 0 0	EVX	**evmheumi**
evmheumia	0 0 0 1 0 0	rD	rA	rB	1 0 0 0 0 1 0 1 0 0 0	EVX	**evmheumia**
evmheumiaaw	0 0 0 1 0 0	rD	rA	rB	1 0 1 0 0 0 0 1 0 0 0	EVX	**evmheumiaaw**
evmheumianw	0 0 0 1 0 0	rD	rA	rB	1 0 1 1 0 0 0 1 0 0 0	EVX	**evmheumianw**
evmheusiaaw	0 0 0 1 0 0	rD	rA	rB	1 0 1 0 0 0 0 0 0 0 0	EVX	**evmheusiaaw**
evmheusianw	0 0 0 1 0 0	rD	rA	rB	1 0 1 1 0 0 0 0 0 0 0	EVX	**evmheusianw**
evmhogsmfaa	0 0 0 1 0 0	rD	rA	rB	1 0 1 0 0 1 0 1 1 1 1	EVX	**evmhogsmfaa**
evmhogsmfan	0 0 0 1 0 0	rD	rA	rB	1 0 1 1 0 1 0 1 1 1 1	EVX	**evmhogsmfan**
evmhogsmiaa	0 0 0 1 0 0	rD	rA	rB	1 0 1 0 0 1 0 1 1 0 1	EVX	**evmhogsmiaa**
evmhogsmian	0 0 0 1 0 0	rD	rA	rB	1 0 1 1 0 1 0 1 1 0 1	EVX	**evmhogsmian**
evmhogumiaa	0 0 0 1 0 0	rD	rA	rB	1 0 1 0 0 1 0 1 1 0 0	EVX	**evmhogumiaa**
evmhogumian	0 0 0 1 0 0	rD	rA	rB	1 0 1 1 0 1 0 1 1 0 0	EVX	**evmhogumian**
evmhosmf	0 0 0 1 0 0	rD	rA	rB	1 0 0 0 0 0 0 1 1 1 1	EVX	**evmhosmf**
evmhosmfa	0 0 0 1 0 0	rD	rA	rB	1 0 0 0 0 1 0 1 1 1 1	EVX	**evmhosmfa**
evmhosmfaaw	0 0 0 1 0 0	rD	rA	rB	1 0 1 0 0 0 0 1 1 1 1	EVX	**evmhosmfaaw**
evmhosmfanw	0 0 0 1 0 0	rD	rA	rB	1 0 1 1 0 0 0 1 1 1 1	EVX	**evmhosmfanw**
evmhosmi	0 0 0 1 0 0	rD	rA	rB	1 0 0 0 0 0 0 1 1 0 1	EVX	**evmhosmi**
evmhosmia	0 0 0 1 0 0	rD	rA	rB	1 0 0 0 0 1 0 1 1 0 1	EVX	**evmhosmia**
evmhosmiaaw	0 0 0 1 0 0	rD	rA	rB	1 0 1 0 0 0 0 1 1 0 1	EVX	**evmhosmiaaw**
evmhosmianw	0 0 0 1 0 0	rD	rA	rB	1 0 1 1 0 0 0 1 1 0 1	EVX	**evmhosmianw**
evmhossf	0 0 0 1 0 0	rD	rA	rB	1 0 0 0 0 0 0 0 1 1 1	EVX	**evmhossf**
evmhossfa	0 0 0 1 0 0	rD	rA	rB	1 0 0 0 0 1 0 0 1 1 1	EVX	**evmhossfa**
evmhossfaaw	0 0 0 1 0 0	rD	rA	rB	1 0 1 0 0 0 0 0 1 1 1	EVX	**evmhossfaaw**
evmhossfanw	0 0 0 1 0 0	rD	rA	rB	1 0 1 1 0 0 0 0 1 1 1	EVX	**evmhossfanw**
evmhossiaaw	0 0 0 1 0 0	rD	rA	rB	1 0 1 0 0 0 0 0 1 0 1	EVX	**evmhossiaaw**
evmhossianw	0 0 0 1 0 0	rD	rA	rB	1 0 1 1 0 0 0 0 1 0 1	EVX	**evmhossianw**
evmhoumi	0 0 0 1 0 0	rD	rA	rB	1 0 0 0 0 0 0 1 1 0 0	EVX	**evmhoumi**
evmhoumia	0 0 0 1 0 0	rD	rA	rB	1 0 0 0 0 1 0 1 1 0 0	EVX	**evmhoumia**
evmhoumiaaw	0 0 0 1 0 0	rD	rA	rB	1 0 1 0 0 0 0 1 1 0 0	EVX	**evmhoumiaaw**
evmhoumianw	0 0 0 1 0 0	rD	rA	rB	1 0 1 1 0 0 0 1 1 0 0	EVX	**evmhoumianw**
evmhousiaaw	0 0 0 1 0 0	rD	rA	rB	1 0 1 0 0 0 0 0 1 0 0	EVX	**evmhousiaaw**
evmhousianw	0 0 0 1 0 0	rD	rA	rB	1 0 1 1 0 0 0 0 1 0 0	EVX	**evmhousianw**
evmr	**evmr r**D,**r**A		equivalent to	**evor r**D,**r**A,**r**A			**evmr**

Table D-1. Instructions (Binary) by Mnemonic

Mnemonic	0	1	2	3	4	5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evmra	0	0	0	1	0	0	rD	rA	///	1	0	0	1	1	0	0	0	1	0	0	EVX	**evmra**
evmwhsmf	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	0	1	1	1	1	EVX	**evmwhsmf**
evmwhsmfa	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	0	1	1	1	1	EVX	**evmwhsmfa**
evmwhsmi	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	0	1	1	0	1	EVX	**evmwhsmi**
evmwhsmia	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	0	1	1	0	1	EVX	**evmwhsmia**
evmwhssf	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	0	0	1	1	1	EVX	**evmwhssf**
evmwhssfa	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	0	0	1	1	1	EVX	**evmwhssfa**
evmwhumi	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	0	1	1	0	0	EVX	**evmwhumi**
evmwhumia	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	0	1	1	0	0	EVX	**evmwhumia**
evmwhusiaaw	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	1	0	0	0	1	0	0	EVX	**evmwhusiaaw**
evmwhusianw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	1	0	0	0	1	0	0	EVX	**evmwhusianw**
evmwlumi	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	0	1	0	0	0	EVX	**evmwlumi**
evmwlumia	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	0	1	0	0	0	EVX	**evmwlumia**
evmwlumiaaw	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	1	0	0	1	0	0	0	EVX	**evmwlumiaaw**
evmwlumianw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	1	0	0	1	0	0	0	EVX	**evmwlumianw**
evmwlusiaaw	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	1	0	0	0	0	0	0	EVX	**evmwlusiaaw**
evmwlusianw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	1	0	0	0	0	0	0	EVX	**evmwlusianw**
evmwsmf	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	1	1	0	1	1	EVX	**evmwsmf**
evmwsmfa	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	1	1	0	1	1	EVX	**evmwsmfa**
evmwsmfaa	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	1	0	1	1	0	1	1	EVX	**evmwsmfaa**
evmwsmfan	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	1	0	1	1	0	1	1	EVX	**evmwsmfan**
evmwsmi	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	1	1	0	0	1	EVX	**evmwsmi**
evmwsmia	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	1	1	0	0	1	EVX	**evmwsmia**
evmwsmiaa	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	1	0	1	1	0	0	1	EVX	**evmwsmiaa**
evmwsmian	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	1	0	1	1	0	0	1	EVX	**evmwsmian**
evmwssf	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	1	0	0	1	1	EVX	**evmwssf**
evmwssfa	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	1	0	0	1	1	EVX	**evmwssfa**
evmwssfaa	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	1	0	1	0	0	1	1	EVX	**evmwssfaa**
evmwssfan	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	1	0	1	0	0	1	1	EVX	**evmwssfan**
evmwumi	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	1	1	0	0	0	EVX	**evmwumi**
evmwumia	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	1	1	0	0	0	EVX	**evmwumia**
evmwumiaa	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	1	0	1	1	0	0	0	EVX	**evmwumiaa**
evmwumian	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	1	0	1	1	0	0	0	EVX	**evmwumian**

Table D-1. Instructions (Binary) by Mnemonic

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30 31	Form	Mnemonic
evnand	0 0 0 1 0 0	rD	rA	rB	0 1 0 0 0 0 1 1 1 1 0	EVX	**evnand**
evneg	0 0 0 1 0 0	rD	rA	///	0 1 0 0 0 0 0 1 0 0 1	EVX	**evneg**
evnor	0 0 0 1 0 0	rD	rA	rB	0 1 0 0 0 0 1 1 0 0 0	EVX	**evnor**
evnot	**evnot r**D,**r**A		equivalent to	**evnor r**D,**r**A,**r**A			**evnot**
evor	0 0 0 1 0 0	rD	rA	rB	0 1 0 0 0 0 1 0 1 1 1	EVX	**evor**
evorc	0 0 0 1 0 0	rD	rA	rB	0 1 0 0 0 0 1 1 0 1 1	EVX	**evorc**
evrlw	0 0 0 1 0 0	rD	rA	rB	0 1 0 0 0 1 0 1 0 0 0	EVX	**evrlw**
evrlwi	0 0 0 1 0 0	rD	rA	UIMM	0 1 0 0 0 1 0 1 0 1 0	EVX	**evrlwi**
evrndw	0 0 0 1 0 0	rD	rA	UIMM	0 1 0 0 0 0 0 1 1 0 0	EVX	**evrndw**
evsel	0 0 0 1 0 0	rD	rA	rB	0 1 0 0 1 1 1 1 | crfS	EVX	**evsel**
evslw	0 0 0 1 0 0	rD	rA	rB	0 1 0 0 0 1 0 0 1 0 0	EVX	**evslw**
evslwi	0 0 0 1 0 0	rD	rA	UIMM	0 1 0 0 0 1 0 0 1 1 0	EVX	**evslwi**
evsplatfi	0 0 0 1 0 0	rD	SIMM	///	0 1 0 0 0 1 0 1 0 1 1	EVX	**evsplatfi**
evsplati	0 0 0 1 0 0	rD	SIMM	///	0 1 0 0 0 1 0 1 0 0 1	EVX	**evsplati**
evsrwis	0 0 0 1 0 0	rD	rA	UIMM	0 1 0 0 0 1 0 0 0 1 1	EVX	**evsrwis**
evsrwiu	0 0 0 1 0 0	rD	rA	UIMM	0 1 0 0 0 1 0 0 0 1 0	EVX	**evsrwiu**
evsrws	0 0 0 1 0 0	rD	rA	rB	0 1 0 0 0 1 0 0 0 0 1	EVX	**evsrws**
evsrwu	0 0 0 1 0 0	rD	rA	rB	0 1 0 0 0 1 0 0 0 0 0	EVX	**evsrwu**
evstdd	0 0 0 1 0 0	rD	rA	UIMM [6]	0 1 1 0 0 1 0 0 0 0 1	EVX	**evstdd**
evstddx	0 0 0 1 0 0	rS	rA	rB	0 1 1 0 0 1 0 0 0 0 0	EVX	**evstddx**
evstdh	0 0 0 1 0 0	rS	rA	UIMM [6]	0 1 1 0 0 1 0 0 1 0 1	EVX	**evstdh**
evstdhx	0 0 0 1 0 0	rS	rA	rB	0 1 1 0 0 1 0 0 1 0 0	EVX	**evstdhx**
evstdw	0 0 0 1 0 0	rS	rA	UIMM [6]	0 1 1 0 0 1 0 0 0 1 1	EVX	**evstdw**
evstdwx	0 0 0 1 0 0	rS	rA	rB	0 1 1 0 0 1 0 0 0 1 0	EVX	**evstdwx**
evstwhe	0 0 0 1 0 0	rS	rA	UIMM [8]	0 1 1 0 0 1 1 0 0 0 1	EVX	**evstwhe**
evstwhex	0 0 0 1 0 0	rS	rA	rB	0 1 1 0 0 1 1 0 0 0 0	EVX	**evstwhex**
evstwho	0 0 0 1 0 0	rS	rA	UIMM [8]	0 1 1 0 0 1 1 0 1 0 1	EVX	**evstwho**
evstwhox	0 0 0 1 0 0	rS	rA	rB	0 1 1 0 0 1 1 0 1 0 0	EVX	**evstwhox**
evstwwe	0 0 0 1 0 0	rS	rA	UIMM [8]	0 1 1 0 0 1 1 1 0 0 1	EVX	**evstwwe**
evstwwex	0 0 0 1 0 0	rS	rA	rB	0 1 1 0 0 1 1 1 0 0 0	EVX	**evstwwex**
evstwwo	0 0 0 1 0 0	rS	rA	UIMM [8]	0 1 1 0 0 1 1 1 1 0 1	EVX	**evstwwo**
evstwwox	0 0 0 1 0 0	rS	rA	rB	0 1 1 0 0 1 1 1 1 0 0	EVX	**evstwwox**
evsubfsmiaaw	0 0 0 1 0 0	rD	rA	///	1 0 0 1 1 0 0 1 0 1 1	EVX	**evsubfsmiaaw**

Table D-1. Instructions (Binary) by Mnemonic

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30 31	Form	Mnemonic
evsubfssiaaw	0 0 0 1 0 0	rD	rA	///	1 0 0 1 1 0 0 0 0 1 1	EVX	**evsubfssiaaw**
evsubfumiaaw	0 0 0 1 0 0	rD	rA	///	1 0 0 1 1 0 0 1 0 1 0	EVX	**evsubfumiaaw**
evsubfusiaaw	0 0 0 1 0 0	rD	rA	///	1 0 0 1 1 0 0 0 0 1 0	EVX	**evsubfusiaaw**
evsubfw	0 0 0 1 0 0	rD	rA	rB	0 1 0 0 0 0 0 0 1 0 0	EVX	**evsubfw**
evsubifw	0 0 0 1 0 0	rD	UIMM	rB	0 1 0 0 0 0 0 0 1 1 0	EVX	**evsubifw**
evsubiw	**evsubiw r**D,**r**B,**UIMM**		equivalent to	**evsubifw r**D,UIMM,**r**B			**evsubiw**
evsubw	**evsubw r**D,**r**B,**r**A		equivalent to	**evsubfw r**D,**r**A,**r**B			**evsubw**
evxor	0 0 0 1 0 0	rD	rA	rB	0 1 0 0 0 0 1 0 1 1 0	EVX	**evxor**
extlwi	**extlwi r**A,**r**S,*n*,*b* (n > 0)		equivalent to	**rlwinm r**A,**r**S,*b*,**0**,*n* – 1			**extlwi**
extrwi	**extrwi r**A,**r**S,*n*,*b* (*n* > 0)		equivalent to	**rlwinm r**A,**r**S,*b* + *n*, 32 – *n*,**31**			**extrwi**
extsb	0 1 1 1 1 1	rS	rA	///	1 1 1 0 1 1 1 0 1 0 0	X	**extsb**
extsb.	0 1 1 1 1 1	rS	rA	///	1 1 1 0 1 1 1 0 1 0 1	X	**extsb.**
extsh	0 1 1 1 1 1	rS	rA	///	1 1 1 0 0 1 1 0 1 0 0	X	**extsh**
extsh.	0 1 1 1 1 1	rS	rA	///	1 1 1 0 0 1 1 0 1 0 1	X	**extsh.**
icbi	0 1 1 1 1 1	///	rA	rB	1 1 1 1 0 1 0 1 1 0 /	X	**icbi**
icblc	0 1 1 1 1 1	CT	rA	rB	0 0 1 1 1 0 0 1 1 0 0	X	**icblc**
icbt	0 1 1 1 1 1	CT	rA	rB	0 0 0 0 0 1 0 1 1 0 /	X	**icbt**
icbtls	0 1 1 1 1 1	CT	rA	rB	0 1 1 1 1 0 0 1 1 0 0	X	**icbtls**
inslwi	**inslwi r**A,**r**S,*n*,*b* **(***n* **>** 0)		equivalent to	**rlwimi r**A,**r**S,32 – *b*,*b*,(*b* + *n*) – 1			**inslwi**
insrwi	**insrwi r**A,**r**S,*n*,*b* (*n* **>** 0)		equivalent to	**rlwimi r**A,**r**S,32 – (*b* + *n*),*b*,(*b* + *n*) – 1			**insrwi**
isel	0 1 1 1 1 1	rD	rA	rB	crb 0 1 1 1 1 0	X	**isel**
iseleq	**iseleq r**D,**r**A,**r**B		equivalent to	**isel r**D,**r**A,**r**B,**2**			**iseleq**
iselgt	**iselgt r**D,**r**A,**r**B		equivalent to	**isel r**D,**r**A,**r**B,**1**			**iselgt**
isellt	**isellt r**D,**r**A,**r**B		equivalent to	**isel r**D,**r**A,**r**B,**0**			**isellt**
isync	0 1 0 0 1 1	///			0 0 1 0 0 1 0 1 1 0 /	XL	**isync**
la	**la r**D,**d(r**A)		equivalent to	**addi r**D,**r**A,d			**la**
lbz	1 0 0 0 1 0	rD	rA	D		D	**lbz**
lbzu	1 0 0 0 1 1	rD	rA	D		D	**lbzu**
lbzux	0 1 1 1 1 1	rD	rA	rB	0 0 0 1 1 1 0 1 1 1 /	X	**lbzux**
lbzx	0 1 1 1 1 1	rD	rA	rB	0 0 0 1 0 1 0 1 1 1 /	X	**lbzx**
lha	1 0 1 0 1 0	rD	rA	D		D	**lha**
lhau	1 0 1 0 1 1	rD	rA	D		D	**lhau**
lhaux	0 1 1 1 1 1	rD	rA	rB	0 1 0 1 1 1 0 1 1 1 /	X	**lhaux**

Table D-1. Instructions (Binary) by Mnemonic

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
lhax	0	1	1	1	1	1	rD					rA					rB					0	1	0	1	0	1	0	1	1	1	/	X	**lhax**
lhbrx	0	1	1	1	1	1	rD					rA					rB					1	1	0	0	0	1	0	1	1	0	/	X	**lhbrx**
lhz	1	0	1	0	0	0	rD					rA					D																D	**lhz**
lhzu	1	0	1	0	0	1	rD					rA					D																D	**lhzu**
lhzux	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	1	1	0	1	1	1	/	X	**lhzux**
lhzx	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	1	0	1	1	1	/	X	**lhzx**
li	**li r**D**,value**											equivalent to							**addi r**D**,0,**value															**li**
lis	**lis r**D**,value**											equivalent to							**addis r**D**,0,**value															**lis**
lmw	1	0	1	1	1	0	rD					rA					D																D	**lmw**
lwarx	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	0	1	0	1	0	0	/	X	**lwarx**
lwbrx	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	1	0	1	1	0	/	X	**lwbrx**
lwz	1	0	0	0	0	0	rD					rA					D																D	**lwz**
lwzu	1	0	0	0	0	1	rD					rA					D																D	**lwzu**
lwzux	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	1	1	0	1	1	1	/	X	**lwzux**
lwzx	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	0	1	0	1	1	1	/	X	**lwzx**
mbar	0	1	1	1	1	1	MO					///										1	1	0	1	0	1	0	1	1	0	/	X	**mbar**
mcrf	0	1	0	0	1	1	crfD			//		crfS			///							0	0	0	0	0	0	0	0	0	0	/	XL	**mcrf**
mcrxr	0	1	1	1	1	1	crfD			///												1	0	0	0	0	0	0	0	0	0	/	X	**mcrxr**
mfcr	0	1	1	1	1	1	rD					///										0	0	0	0	0	1	0	0	1	1	/	X	**mfcr**
mfcr	**mtcr r**S											equivalent to							**mtcrf 0xFF,r**S															**mfcr**
mfmsr	0	1	1	1	1	1	rD					///										0	0	0	1	0	1	0	0	1	1	/	X	**mfmsr**
mfpmr	0	1	1	1	1	1	rD					PMRN5–9					PMRN0–4					0	1	0	1	0	0	1	1	1	0	0	XFX	**mfpmr**
mf*regname*	**mf***regname* **r**D											equivalent to							**mfspr r**D**,SPR*n***															**mfasr**
mfspr	0	1	1	1	1	1	rD					SPRN5–9					SPRN0–4					0	1	0	1	0	1	0	0	1	1	/	XFX	**mfspr**
mr	**mr r**A**,r**S											equivalent to							**or r**A**,r**S**,r**S															**mr**
msync	0	1	1	1	1	1	///															1	0	0	1	0	1	0	1	1	0	/	X	**msync**
mtcr	**mtcr r**S											equivalent to							**mtcrf** 0xFF**,r**S															**mtcr**
mtcrf	0	1	1	1	1	1	rS					/	CRM								/	0	0	1	0	0	1	0	0	0	0	/	XFX	**mtcrf**
mtmsr	0	1	1	1	1	1	rS					///										0	0	1	0	0	1	0	0	1	0	/	X	**mtmsr**
mtpmr	0	1	1	1	1	1	rS					PMRN5–9					PMRN0–4					0	1	1	1	0	0	1	1	1	0	0	XFX	**mtpmr**
mt*regname*	**mt***regname* **r**S											equivalent to							**mtspr SPR***n* **r**S															**mt***regname*
mtspr	0	1	1	1	1	1	rS					SPRN5–9					SPRN0–4					0	1	1	1	0	1	0	0	1	1	/	XFX	**mtspr**
mulhw	0	1	1	1	1	1	rD					rA					rB					/	0	0	1	0	0	1	0	1	1	0	X	**mulhw**

Table D-1. Instructions (Binary) by Mnemonic

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29 30	31	Form	Mnemonic
mulhw.	0 1 1 1 1 1	rD	rA	rB	/ 0 0 1 0	0 1 0 1 1	1	X	**mulhw.**
mulhwu	0 1 1 1 1 1	rD	rA	rB	/ 0 0 0 0	0 1 0 1 1	0	X	**mulhwu**
mulhwu.	0 1 1 1 1 1	rD	rA	rB	/ 0 0 0 0	0 1 0 1 1	1	X	**mulhwu.**
mulli	0 0 0 1 1 1	rD	rA	SIMM				D	**mulli**
mullw	0 1 1 1 1 1	rD	rA	rB	0 0 1 1 1	0 1 0 1 1	0	X	**mullw**
mullw.	0 1 1 1 1 1	rD	rA	rB	0 0 1 1 1	0 1 0 1 1	1	X	**mullw.**
mullwo	0 1 1 1 1 1	rD	rA	rB	1 0 1 1 1	0 1 0 1 1	0	X	**mullwo**
mullwo.	0 1 1 1 1 1	rD	rA	rB	1 0 1 1 1	0 1 0 1 1	1	X	**mullwo.**
nand	0 1 1 1 1 1	rS	rA	rB	0 1 1 1 0	1 1 1 0 0	0	X	**nand**
nand.	0 1 1 1 1 1	rS	rA	rB	0 1 1 1 0	1 1 1 0 0	1	X	**nand.**
neg	0 1 1 1 1 1	rD	rA	///	0 0 0 1 1	0 1 0 0 0	0	X	**neg**
neg.	0 1 1 1 1 1	rD	rA	///	0 0 0 1 1	0 1 0 0 0	1	X	**neg.**
nego	0 1 1 1 1 1	rD	rA	///	1 0 0 1 1	0 1 0 0 0	0	X	**nego**
nego.	0 1 1 1 1 1	rD	rA	///	1 0 0 1 1	0 1 0 0 0	1	X	**nego.**
nop	**nop**		equivalent to	**ori 0,0,0**					**nop**
nor	0 1 1 1 1 1	rS	rA	rB	0 0 0 1 1	1 1 1 0 0	0	X	**nor**
nor.	0 1 1 1 1 1	rS	rA	rB	0 0 0 1 1	1 1 1 0 0	1	X	**nor.**
not	**not r**A,**r**S		equivalent to	**nor r**A,**r**S,**r**S					**not**
or	0 1 1 1 1 1	rS	rA	rB	0 1 1 0 1	1 1 1 0 0	0	X	**or**
or.	0 1 1 1 1 1	rS	rA	rB	0 1 1 0 1	1 1 1 0 0	1	X	**or.**
orc	0 1 1 1 1 1	rS	rA	rB	0 1 1 0 0	1 1 1 0 0	0	X	**orc**
orc.	0 1 1 1 1 1	rS	rA	rB	0 1 1 0 0	1 1 1 0 0	1	X	**orc.**
ori	0 1 1 0 0 0	rS	rA	UIMM				D	**ori**
oris	0 1 1 0 0 1	rS	rA	UIMM				D	**oris**
rfci	0 1 0 0 1 1	///			0 0 0 0 1	1 0 0 1 1	/	XL	**rfci**
rfi	0 1 0 0 1 1	///			0 0 0 0 1	1 0 0 1 0	/	XL	**rfi**
rfmci	0 1 0 0 1 1	///			0 0 0 0 1	0 0 1 1 0	/	XL	**rfmci**
rlwimi	0 1 0 1 0 0	rS	rA	SH	MB	ME	Rc	M	**rlwimi**
rlwimi.	0 1 0 1 0 0	rS	rA	SH	MB	ME	Rc	M	**rlwimi.**
rlwinm	0 1 0 1 0 1	rS	rA	SH	MB	ME	0	M	**rlwinm**
rlwinm.	0 1 0 1 0 1	rS	rA	SH	MB	ME	1	M	**rlwinm.**
rlwnm	0 1 0 1 1 1	rS	rA	rB	MB	ME	Rc	M	**rlwnm**
rlwnm.	0 1 0 1 1 1	rS	rA	rB	MB	ME	Rc	M	**rlwnm.**

Table D-1. Instructions (Binary) by Mnemonic

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31	Form	Mnemonic
rotlw	**rotlw r**A,**r**S,**r**B		equivalent to	**rlwnm r**A,**r**S,**r**B,**0**,**31**				**rotlw**
rotlwi	**rotlwi r**A,**r**S,*n*		equivalent to	**rlwinm r**A,**r**S,*n*,**0**,**31**				**rotlwi**
rotrwi	**rotrwi r**A,**r**S,*n*		equivalent to	**rlwinm r**A,**r**S,32 – *n*,**0**,**31**				**rotrwi**
sc	0 1 0 0 0 1	///			1	/	SC	**sc**
slw	0 1 1 1 1 1	rS	rA	rB	0 0 0 0 0 1 1 0 0 0	0	X	**slw**
slw.	0 1 1 1 1 1	rS	rA	rB	0 0 0 0 0 1 1 0 0 0	1	X	**slw.**
slwi	**slwi r**A,**r**S,*n* (n < 32)		equivalent to	**rlwinm r**A,**r**S,*n*,**0**,31 – *n*				**slwi**
sraw	0 1 1 1 1 1	rS	rA	rB	1 1 0 0 0 1 1 0 0 0	0	X	**sraw**
sraw.	0 1 1 1 1 1	rS	rA	rB	1 1 0 0 0 1 1 0 0 0	1	X	**sraw.**
srawi	0 1 1 1 1 1	rS	rA	SH	1 1 0 0 1 1 1 0 0 0	0	X	**srawi**
srawi.	0 1 1 1 1 1	rS	rA	SH	1 1 0 0 1 1 1 0 0 0	1	X	**srawi.**
srw	0 1 1 1 1 1	rS	rA	rB	1 0 0 0 0 1 1 0 0 0	0	X	**srw**
srw.	0 1 1 1 1 1	rS	rA	rB	1 0 0 0 0 1 1 0 0 0	1	X	**srw.**
srwi	**srwi r**A,**r**S,*n* (n < 32)		equivalent to	**rlwinm r**A,**r**S,32 – *n*,*n*,**31**				**srwi**
stb	1 0 0 1 1 0	rS	rA	D			D	**stb**
stbu	1 0 0 1 1 1	rS	rA	D			D	**stbu**
stbux	0 1 1 1 1 1	rS	rA	rB	0 0 1 1 1 1 0 1 1 1	0	X	**stbux**
stbx	0 1 1 1 1 1	rS	rA	rB	0 0 1 1 0 1 0 1 1 1	0	X	**stbx**
sth	1 0 1 1 0 0	rS	rA	D			D	**sth**
sthbrx	0 1 1 1 1 1	rS	rA	rB	1 1 1 0 0 1 0 1 1 0	/	X	**sthbrx**
sthu	1 0 1 1 0 1	rS	rA	D			D	**sthu**
sthux	0 1 1 1 1 1	rS	rA	rB	0 1 1 0 1 1 0 1 1 1	/	X	**sthux**
sthx	0 1 1 1 1 1	rS	rA	rB	0 1 1 0 0 1 0 1 1 1	/	X	**sthx**
stmw	1 0 1 1 1 1	rS	rA	D			D	**stmw**
stw	1 0 0 1 0 0	rS	rA	D			D	**stw**
stwbrx	0 1 1 1 1 1	rS	rA	rB	1 0 1 0 0 1 0 1 1 0	/	X	**stwbrx**
stwcx.	0 1 1 1 1 1	rS	rA	rB	0 0 1 0 0 1 0 1 1 0	1	X	**stwcx.**
stwu	1 0 0 1 0 1	rS	rA	D			D	**stwu**
stwux	0 1 1 1 1 1	rS	rA	rB	0 0 1 0 1 1 0 1 1 1	/	D	**stwux**
stwx	0 1 1 1 1 1	rS	rA	rB	0 0 1 0 0 1 0 1 1 1	/	D	**stwx**
sub	**sub r**D,**r**A,**r**B		equivalent to	**subf r**D,**r**B,**r**A				**sub**
subc	**subc r**D,**r**A,**r**B		equivalent to	**subfc r**D,**r**B,**r**A				**subc**
subf	0 1 1 1 1 1	rD	rA	rB	0 0 0 0 1 0 1 0 0 0	0	X	**subf**

Table D-1. Instructions (Binary) by Mnemonic

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31	Form	Mnemonic
subf.	0 1 1 1 1 1	rD	rA	rB	0 0 0 0 1 0 1 0 0 0	1	X	**subf.**
subfc	0 1 1 1 1 1	rD	rA	rB	0 0 0 0 0 0 1 0 0 0	0	X	**subfc**
subfc.	0 1 1 1 1 1	rD	rA	rB	0 0 0 0 0 0 1 0 0 0	1	X	**subfc.**
subfco	0 1 1 1 1 1	rD	rA	rB	1 0 0 0 0 0 1 0 0 0	0	X	**subfco**
subfco.	0 1 1 1 1 1	rD	rA	rB	1 0 0 0 0 0 1 0 0 0	1	X	**subfco.**
subfe	0 1 1 1 1 1	rD	rA	rB	0 0 1 0 0 0 1 0 0 0	0	X	**subfe**
subfe.	0 1 1 1 1 1	rD	rA	rB	0 0 1 0 0 0 1 0 0 0	1	X	**subfe.**
subfeo	0 1 1 1 1 1	rD	rA	rB	1 0 1 0 0 0 1 0 0 0	0	X	**subfeo**
subfeo.	0 1 1 1 1 1	rD	rA	rB	1 0 1 0 0 0 1 0 0 0	1	X	**subfeo.**
subfic	0 0 1 0 0 0	rD	rA	SIMM			D	**subfic**
subfme	0 1 1 1 1 1	rD	rA	///	0 0 1 1 1 0 1 0 0 0	0	X	**subfme**
subfme.	0 1 1 1 1 1	rD	rA	///	0 0 1 1 1 0 1 0 0 0	1	X	**subfme.**
subfmeo	0 1 1 1 1 1	rD	rA	///	1 0 1 1 1 0 1 0 0 0	0	X	**subfmeo**
subfmeo.	0 1 1 1 1 1	rD	rA	///	1 0 1 1 1 0 1 0 0 0	1	X	**subfmeo.**
subfo	0 1 1 1 1 1	rD	rA	rB	1 0 0 0 1 0 1 0 0 0	0	X	**subfo**
subfo.	0 1 1 1 1 1	rD	rA	rB	1 0 0 0 1 0 1 0 0 0	1	X	**subfo.**
subfze	0 1 1 1 1 1	rD	rA	///	0 0 1 1 0 0 1 0 0 0	0	X	**subfze**
subfze.	0 1 1 1 1 1	rD	rA	///	0 0 1 1 0 0 1 0 0 0	1	X	**subfze.**
subfzeo	0 1 1 1 1 1	rD	rA	///	1 0 1 1 0 0 1 0 0 0	0	X	**subfzeo**
subfzeo.	0 1 1 1 1 1	rD	rA	///	1 0 1 1 0 0 1 0 0 0	1	X	**subfzeo.**
subi	**subi r**D,**r**A,**value**	equivalent to	**addi r**D,**r**A,–value					**subi**
subic	**subic r**D,**r**A,**value**	equivalent to	**addic r**D,**r**A,–value					**subic**
subic.	**subic. r**D,**r**A,**value**	equivalent to	**addic. r**D,**r**A,–value					**subic.**
subis	**subis r**D,**r**A,**value**	equivalent to	**addis r**D,**r**A,–value					**subis**
tlbivax	0 1 1 1 1 1	///	rA	rB	1 1 0 0 0 1 0 0 1 0	/	X	**tlbivax**
tlbre	0 1 1 1 1 1	///[9]			1 1 1 0 1 1 0 0 1 0	/	X	**tlbre**
tlbsx	0 1 1 1 1 1	///	rA	rB	1 1 1 0 0 1 0 0 1 0	/	X	**tlbsx**
tlbsync	0 1 1 1 1 1	///			1 0 0 0 1 1 0 1 1 0	/	X	**tlbsync**
tlbwe	0 1 1 1 1 1	///			1 1 1 1 0 1 0 0 1 0	/	X	**tlbwe**
tw	0 1 1 1 1 1	TO	rA	rB	0 0 0 0 0 0 0 1 0 0	/	X	**tw**
tweq	**tweq r**A,**SIMM**	equivalent to	**tw 4,r**A,SIMM					**tweq**
tweqi	**tweqi r**A,**SIMM**	equivalent to	**twi 4,r**A,SIMM					**tweqi**
twge	**twge r**A,**SIMM**	equivalent to	**tw 12,r**A,SIMM					**twge**

Table D-1. Instructions (Binary) by Mnemonic

Mnemonic	0–5	6–10	11–15	16–20	21–30	31	Form	Mnemonic
twgei	**twgei rA,SIMM**	equivalent to	**twi 12,rA,SIMM**					**twgei**
twgt	**twgt rA,SIMM**	equivalent to	**tw 8,rA,SIMM**					**twgt**
twgti	**twgti rA,SIMM**	equivalent to	**twi 8,rA,SIMM**					**twgti**
twi	0 0 0 0 1 1	TO	rA	SIMM			D	**twi**
twle	**twle rA,SIMM**	equivalent to	**tw 20,rA,SIMM**					**twle**
twlei	**twlei rA,SIMM**	equivalent to	**twi 20,rA,SIMM**					**twlei**
twlge	**twlge rA,SIMM**	equivalent to	**tw 12,rA,SIMM**					**twlge**
twlgei	**twlgei rA,SIMM**	equivalent to	**twi 12,rA,SIMM**					**twlgei**
twlgt	**twlgt rA,SIMM**	equivalent to	**tw 1,rA,SIMM**					**twlgt**
twlgti	**twlgti rA,SIMM**	equivalent to	**twi 1,rA,SIMM**					**twlgti**
twlle	**twlle rA,SIMM**	equivalent to	**tw 6,rA,SIMM**					**twlle**
twllei	**twllei rA,SIMM**	equivalent to	**twi 6,rA,SIMM**					**twllei**
twllt	**twllt rA,SIMM**	equivalent to	**tw 2,rA,SIMM**					**twllt**
twllti	**twllti rA,SIMM**	equivalent to	**twi 2,rA,SIMM**					**twllti**
twlng	**twlng rA,SIMM**	equivalent to	**tw 6,rA,SIMM**					**twlng**
twlngi	**twlngi rA,SIMM**	equivalent to	**twi 6,rA,SIMM**					**twlngi**
twlnl	**twlnl rA,SIMM**	equivalent to	**tw 5,rA,SIMM**					**twlnl**
twlnli	**twlnli rA,SIMM**	equivalent to	**twi 5,rA,SIMM**					**twlnli**
twlt	**twlt rA,SIMM**	equivalent to	**tw 16,rA,SIMM**					**twlt**
twlti	**twlti rA,SIMM**	equivalent to	**twi 16,rA,SIMM**					**twlti**
twne	**twne rA,SIMM**	equivalent to	**tw 24,rA,SIMM**					**twne**
twnei	**twnei rA,SIMM**	equivalent to	**twi 24,rA,SIMM**					**twnei**
twng	**twng rA,SIMM**	equivalent to	**tw 20,rA,SIMM**					**twng**
twngi	**twngi rA,SIMM**	equivalent to	**twi 20,rA,SIMM**					**twngi**
twnl	**twnl rA,SIMM**	equivalent to	**tw 12,rA,SIMM**					**twnl**
twnli	**twnli rA,SIMM**	equivalent to	**twi 12,rA,SIMM**					**twnli**
wrtee	0 1 1 1 1 1	rS	///		0 0 1 0 0 0 0 0 1 1	/	X	**wrtee**
wrteei	0 1 1 1 1 1	///		E ///	0 0 1 0 1 0 0 0 1 1	/	X	**wrteei**
xor	0 1 1 1 1 1	rS	rA	rB	0 1 0 0 1 1 1 1 0 0	0	X	**xor**
xor.	0 1 1 1 1 1	rS	rA	rB	0 1 0 0 1 1 1 1 0 0	1	X	**xor.**
xori	0 1 1 0 1 0	rS	rA	UIMM			D	**xori**
xoris	0 1 1 0 1 1	rS	rA	UIMM			D	**xoris**

1 Simplified mnemonics for branch instructions that do not test a CR bit should not specify one; a programming error may occur.

[2] The value in the BI operand selects CR*n*[2], the EQ bit.
[3] The value in the BI operand selects CR*n*[0], the LT bit.
[4] The value in the BI operand selects CR*n*[1], the GT bit.
[5] The value in the BI operand selects CR*n*[3], the SO bit.
[6] d = UIMM * 8
[7] d = UIMM * 2
[8] d = UIMM * 4
[9] This field is defined as allocated by the Book E architecture, for possible use in an implementation. These bits are not implemented in the e500.

D.2 Instructions (Decimal and Hexadecimal) by Opcode

Table D-2 lists e500 instructions by opcode.

Table D-2. Instructions (Decimal and Hexadecimal) by Opcode

Mnemonic	0 1 2 3 4 5	6 7 8	9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30 31	Form	Mnemonic
twi	03	TO		rA	SIMM		D	**twi**
brinc	04	rD		rA	rB	0 1 0 0 0 0 0 1 1 1 1	EVX	**brinc**
efsabs	04	rD		rA	///	0 1 0 1 1 0 0 0 1 0 0	EFX	**efsabs**
efsadd	04	rD		rA	rB	0 1 0 1 1 0 0 0 0 0 0	EFX	**efsadd**
efscfsf	04	rD		///	rB	0 1 0 1 1 0 1 0 0 1 1	EFX	**efscfsf**
efscfsi	04	rD		///	rB	0 1 0 1 1 0 1 0 0 0 1	EFX	**efscfsi**
efscfuf	04	rD		///	rB	0 1 0 1 1 0 1 0 0 1 0	EFX	**efscfuf**
efscfui	04	rD		///	rB	0 1 0 1 1 0 1 0 0 0 0	EFX	**efscfui**
efscmpeq	04	crfD	/ /	rA	rB	0 1 0 1 1 0 0 1 1 1 0	EFX	**efscmpeq**
efscmpgt	04	crfD	/ /	rA	rB	0 1 0 1 1 0 0 1 1 0 0	EFX	**efscmpgt**
efscmplt	04	crfD	/ /	rA	rB	0 1 0 1 1 0 0 1 1 0 1	EFX	**efscmplt**
efsctsf	04	rD		///	rB	0 1 0 1 1 0 1 0 1 1 1	EFX	**efsctsf**
efsctsi	04	rD		///	rB	0 1 0 1 1 0 1 0 1 0 1	EFX	**efsctsi**
efsctsiz	04	rD		///	rB	0 1 0 1 1 0 1 1 0 1 0	EFX	**efsctsiz**
efsctuf	04	rD		///	rB	0 1 0 1 1 0 1 0 1 1 0	EFX	**efsctuf**
efsctui	04	rD		///	rB	0 1 0 1 1 0 1 0 1 0 0	EFX	**efsctui**
efsctuiz	04	rD		///	rB	0 1 0 1 1 0 1 1 0 0 0	EFX	**efsctuiz**
efsdiv	04	rD		rA	rB	0 1 0 1 1 0 0 1 0 0 1	EFX	**efsdiv**
efsmul	04	rD		rA	rB	0 1 0 1 1 0 0 1 0 0 0	EFX	**efsmul**
efsnabs	04	rD		rA	///	0 1 0 1 1 0 0 0 1 0 1	EFX	**efsnabs**
efsneg	04	rD		rA	///	0 1 0 1 1 0 0 0 1 1 0	EFX	**efsneg**
efssub	04	rD		rA	rB	0 1 0 1 1 0 0 0 0 0 1	EFX	**efssub**
efststeq	04	crfD	/ /	rA	rB	0 1 0 1 1 0 1 1 1 1 0	EFX	**efststeq**
efststgt	04	crfD	/ /	rA	rB	0 1 0 1 1 0 1 1 1 0 0	EFX	**efststgt**

Table D-2. Instructions (Decimal and Hexadecimal) by Opcode

Mnemonic	0–5	6–10	11–15	16–20	21–31	Form	Mnemonic
efststlt	04	crfD / /	rA	rB	0 1 0 1 1 0 1 1 1 0 1	EFX	**efststlt**
evabs	04	rD	rA	///	0 1 0 0 0 0 0 1 0 0 0	EVX	**evabs**
evaddiw	04	rD	UIMM	rB	0 1 0 0 0 0 0 0 0 1 0	EVX	**evaddiw**
evaddsmiaaw	04	rD	rA	///	1 0 0 1 1 0 0 1 0 0 1	EVX	**evaddsmiaaw**
evaddssiaaw	04	rD	rA	///	1 0 0 1 1 0 0 0 0 0 1	EVX	**evaddssiaaw**
evaddumiaaw	04	rD	rA	///	1 0 0 1 1 0 0 1 0 0 0	EVX	**evaddumiaaw**
evaddusiaaw	04	rD	rA	///	1 0 0 1 1 0 0 0 0 0 0	EVX	**evaddusiaaw**
evaddw	04	rD	rA	rB	0 1 0 0 0 0 0 0 0 0 0	EVX	**evaddw**
evand	04	rD	rA	rB	0 1 0 0 0 0 1 0 0 0 1	EVX	**evand**
evandc	04	rD	rA	rB	0 1 0 0 0 0 1 0 0 1 0	EVX	**evandc**
evcmpeq	04	crfD / /	rA	rB	0 1 0 0 0 1 1 0 1 0 0	EVX	**evcmpeq**
evcmpgts	04	crfD / /	rA	rB	0 1 0 0 0 1 1 0 0 0 1	EVX	**evcmpgts**
evcmpgtu	04	crfD / /	rA	rB	0 1 0 0 0 1 1 0 0 0 0	EVX	**evcmpgtu**
evcmplts	04	crfD / /	rA	rB	0 1 0 0 0 1 1 0 0 1 1	EVX	**evcmplts**
evcmpltu	04	crfD / /	rA	rB	0 1 0 0 0 1 1 0 0 1 0	EVX	**evcmpltu**
evcntlsw	04	rD	rA	///	0 1 0 0 0 0 0 1 1 1 0	EVX	**evcntlsw**
evcntlzw	04	rD	rA	///	0 1 0 0 0 0 0 1 1 0 1	EVX	**evcntlzw**
evdivws	04	rD	rA	rB	1 0 0 1 1 0 0 0 1 1 0	EVX	**evdivws**
evdivwu	04	rD	rA	rB	1 0 0 1 1 0 0 0 1 1 1	EVX	**evdivwu**
eveqv	04	rD	rA	rB	0 1 0 0 0 0 1 1 0 0 1	EVX	**eveqv**
evextsb	04	rD	rA	///	0 1 0 0 0 0 0 1 0 1 0	EVX	**evextsb**
evextsh	04	rD	rA	///	0 1 0 0 0 0 0 1 0 1 1	EVX	**evextsh**
evfsabs	04	rD	rA	///	0 1 0 1 0 0 0 0 1 0 0	EVX	**evfsabs**
evfsadd	04	rD	rA	rB	0 1 0 1 0 0 0 0 0 0 0	EVX	**evfsadd**
evfscfsf	04	rD	///	rB	0 1 0 1 0 0 1 0 0 1 1	EVX	**evfscfsf**
evfscfsi	04	rD	///	rB	0 1 0 1 0 0 1 0 0 0 1	EVX	**evfscfsi**
evfscfuf	04	rD	///	rB	0 1 0 1 0 0 1 0 0 1 0	EVX	**evfscfuf**
evfscfui	04	rD	///	rB	0 1 0 1 0 0 1 0 0 0 0	EVX	**evfscfui**
evfscmpeq	04	crfD / /	rA	rB	0 1 0 1 0 0 0 1 1 1 0	EVX	**evfscmpeq**
evfscmpgt	04	crfD / /	rA	rB	0 1 0 1 0 0 0 1 1 0 0	EVX	**evfscmpgt**
evfscmplt	04	crfD / /	rA	rB	0 1 0 1 0 0 0 1 1 0 1	EVX	**evfscmplt**
evfsctsf	04	rD	///	rB	0 1 0 1 0 0 1 0 1 1 1	EVX	**evfsctsf**
evfsctsi	04	rD	///	rB	0 1 0 1 0 0 1 0 1 0 1	EVX	**evfsctsi**

Table D-2. Instructions (Decimal and Hexadecimal) by Opcode

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30 31	Form	Mnemonic
evfsctsiz	04	rD	///	rB	0 1 0 1 0 0 1 1 0 1 0	EVX	**evfsctsiz**
evfsctuf	04	rD	///	rB	0 1 0 1 0 0 1 0 1 1 0	EVX	**evfsctuf**
evfsctui	04	rD	///	rB	0 1 0 1 0 0 1 0 1 0 0	EVX	**evfsctui**
evfsctuiz	04	rD	///	rB	0 1 0 1 0 0 1 1 0 0 0	EVX	**evfsctuiz**
evfsdiv	04	rD	rA	rB	0 1 0 1 0 0 0 1 0 0 1	EVX	**evfsdiv**
evfsmul	04	rD	rA	rB	0 1 0 1 0 0 0 1 0 0 0	EVX	**evfsmul**
evfsnabs	04	rD	rA	///	0 1 0 1 0 0 0 0 1 0 1	EVX	**evfsnabs**
evfsneg	04	rD	rA	///	0 1 0 1 0 0 0 0 1 1 0	EVX	**evfsneg**
evfssub	04	rD	rA	rB	0 1 0 1 0 0 0 0 0 0 1	EVX	**evfssub**
evfststeq	04	crfD / /	rA	rB	0 1 0 1 0 0 1 1 1 1 0	EVX	**evfststeq**
evfststgt	04	crfD / /	rA	rB	0 1 0 1 0 0 1 1 1 0 0	EVX	**evfststgt**
evfststlt	04	crfD / /	rA	rB	0 1 0 1 0 0 1 1 1 0 1	EVX	**evfststlt**
efscfd	04	rD	0 0 0 0 0	rB	0 1 0 1 1 0 0 1 1 1 1	EFX	**efscfd**
efdcfs	04	rD	0 0 0 0 0	rB	0 1 0 1 1 1 0 1 1 1 1	EFX	**efdcfs**
evldd	04	rD	rA	UIMM[1]	0 1 1 0 0 0 0 0 0 0 1	EVX	**evldd**
evlddx	04	rD	rA	rB	0 1 1 0 0 0 0 0 0 0 0	EVX	**evlddx**
evldh	04	rD	rA	UIMM [1]	0 1 1 0 0 0 0 0 1 0 1	EVX	**evldh**
evldhx	04	rD	rA	rB	0 1 1 0 0 0 0 0 1 0 0	EVX	**evldhx**
evldw	04	rD	rA	UIMM [1]	0 1 1 0 0 0 0 0 0 1 1	EVX	**evldw**
evldwx	04	rD	rA	rB	0 1 1 0 0 0 0 0 0 1 0	EVX	**evldwx**
evlhhesplat	04	rD	rA	UIMM[2]	0 1 1 0 0 0 0 1 0 0 1	EVX	**evlhhesplat**
evlhhesplatx	04	rD	rA	rB	0 1 1 0 0 0 0 1 0 0 0	EVX	**evlhhesplatx**
evlhhossplat	04	rD	rA	UIMM [2]	0 1 1 0 0 0 0 1 1 1 1	EVX	**evlhhossplat**
evlhhossplatx	04	rD	rA	rB	0 1 1 0 0 0 0 1 1 1 0	EVX	**evlhhossplatx**
evlhhousplat	04	rD	rA	UIMM [2]	0 1 1 0 0 0 0 1 1 0 1	EVX	**evlhhousplat**
evlhhousplatx	04	rD	rA	rB	0 1 1 0 0 0 0 1 1 0 0	EVX	**evlhhousplatx**
evlwhe	04	rD	rA	UIMM[3]	0 1 1 0 0 0 1 0 0 0 1	EVX	**evlwhe**
evlwhex	04	rD	rA	rB	0 1 1 0 0 0 1 0 0 0 0	EVX	**evlwhex**
evlwhos	04	rD	rA	UIMM [3]	0 1 1 0 0 0 1 0 1 1 1	EVX	**evlwhos**
evlwhosx	04	rD	rA	rB	0 1 1 0 0 0 1 0 1 1 0	EVX	**evlwhosx**
evlwhou	04	rD	rA	UIMM [3]	0 1 1 0 0 0 1 0 1 0 1	EVX	**evlwhou**
evlwhoux	04	rD	rA	rB	0 1 1 0 0 0 1 0 1 0 0	EVX	**evlwhoux**
evlwhsplat	04	rD	rA	UIMM [3]	0 1 1 0 0 0 1 1 1 0 1	EVX	**evlwhsplat**

Table D-2. Instructions (Decimal and Hexadecimal) by Opcode

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evlwhsplatx	04	rD	rA	rB	0	1	1	0	0	0	1	1	1	0	0	EVX	**evlwhsplatx**
evlwwsplat	04	rD	rA	UIMM [3]	0	1	1	0	0	0	1	1	0	0	1	EVX	**evlwwsplat**
evlwwsplatx	04	rD	rA	rB	0	1	1	0	0	0	1	1	0	0	0	EVX	**evlwwsplatx**
evmergehi	04	rD	rA	rB	0	1	0	0	0	1	0	1	1	0	0	EVX	**evmergehi**
evmergehilo	04	rD	rA	rB	0	1	0	0	0	1	0	1	1	1	0	EVX	**evmergehilo**
evmergelo	04	rD	rA	rB	0	1	0	0	0	1	0	1	1	0	1	EVX	**evmergelo**
evmergelohi	04	rD	rA	rB	0	1	0	0	0	1	0	1	1	1	1	EVX	**evmergelohi**
evmhegsmfaa	04	rD	rA	rB	1	0	1	0	0	1	0	1	0	1	1	EVX	**evmhegsmfaa**
evmhegsmfan	04	rD	rA	rB	1	0	1	1	0	1	0	1	0	1	1	EVX	**evmhegsmfan**
evmhegsmiaa	04	rD	rA	rB	1	0	1	0	0	1	0	1	0	0	1	EVX	**evmhegsmiaa**
evmhegsmian	04	rD	rA	rB	1	0	1	1	0	1	0	1	0	0	1	EVX	**evmhegsmian**
evmhegumiaa	04	rD	rA	rB	1	0	1	0	0	1	0	1	0	0	0	EVX	**evmhegumiaa**
evmhegumian	04	rD	rA	rB	1	0	1	1	0	1	0	1	0	0	0	EVX	**evmhegumian**
evmhesmf	04	rD	rA	rB	1	0	0	0	0	0	0	1	0	1	1	EVX	**evmhesmf**
evmhesmfa	04	rD	rA	rB	1	0	0	0	0	1	0	1	0	1	1	EVX	**evmhesmfa**
evmhesmfaaw	04	rD	rA	rB	1	0	1	0	0	0	0	1	0	1	1	EVX	**evmhesmfaaw**
evmhesmfanw	04	rD	rA	rB	1	0	1	1	0	0	0	1	0	1	1	EVX	**evmhesmfanw**
evmhesmi	04	rD	rA	rB	1	0	0	0	0	0	0	1	0	0	1	EVX	**evmhesmi**
evmhesmia	04	rD	rA	rB	1	0	0	0	0	1	0	1	0	0	1	EVX	**evmhesmia**
evmhesmiaaw	04	rD	rA	rB	1	0	1	0	0	0	0	1	0	0	1	EVX	**evmhesmiaaw**
evmhesmianw	04	rD	rA	rB	1	0	1	1	0	0	0	1	0	0	1	EVX	**evmhesmianw**
evmhessf	04	rD	rA	rB	1	0	0	0	0	0	0	0	0	1	1	EVX	**evmhessf**
evmhessfa	04	rD	rA	rB	1	0	0	0	0	1	0	0	0	1	1	EVX	**evmhessfa**
evmhessfaaw	04	rD	rA	rB	1	0	1	0	0	0	0	0	0	1	1	EVX	**evmhessfaaw**
evmhessfanw	04	rD	rA	rB	1	0	1	1	0	0	0	0	0	1	1	EVX	**evmhessfanw**
evmhessiaaw	04	rD	rA	rB	1	0	1	0	0	0	0	0	0	0	1	EVX	**evmhessiaaw**
evmhessianw	04	rD	rA	rB	1	0	1	1	0	0	0	0	0	0	1	EVX	**evmhessianw**
evmheumi	04	rD	rA	rB	1	0	0	0	0	0	0	1	0	0	0	EVX	**evmheumi**
evmheumia	04	rD	rA	rB	1	0	0	0	0	1	0	1	0	0	0	EVX	**evmheumia**
evmheumiaaw	04	rD	rA	rB	1	0	1	0	0	0	0	1	0	0	0	EVX	**evmheumiaaw**
evmheumianw	04	rD	rA	rB	1	0	1	1	0	0	0	1	0	0	0	EVX	**evmheumianw**
evmheusiaaw	04	rD	rA	rB	1	0	1	0	0	0	0	0	0	0	0	EVX	**evmheusiaaw**
evmheusianw	04	rD	rA	rB	1	0	1	1	0	0	0	0	0	0	0	EVX	**evmheusianw**

Table D-2. Instructions (Decimal and Hexadecimal) by Opcode

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evmhogsmfaa	04	rD	rA	rB	1	0	1	0	0	1	0	1	1	1	1	EVX	**evmhogsmfaa**
evmhogsmfan	04	rD	rA	rB	1	0	1	1	0	1	0	1	1	1	1	EVX	**evmhogsmfan**
evmhogsmiaa	04	rD	rA	rB	1	0	1	0	0	1	0	1	1	0	1	EVX	**evmhogsmiaa**
evmhogsmian	04	rD	rA	rB	1	0	1	1	0	1	0	1	1	0	1	EVX	**evmhogsmian**
evmhogumiaa	04	rD	rA	rB	1	0	1	0	0	1	0	1	1	0	0	EVX	**evmhogumiaa**
evmhogumian	04	rD	rA	rB	1	0	1	1	0	1	0	1	1	0	0	EVX	**evmhogumian**
evmhosmf	04	rD	rA	rB	1	0	0	0	0	0	0	1	1	1	1	EVX	**evmhosmf**
evmhosmfa	04	rD	rA	rB	1	0	0	0	0	1	0	1	1	1	1	EVX	**evmhosmfa**
evmhosmfaaw	04	rD	rA	rB	1	0	1	0	0	0	0	1	1	1	1	EVX	**evmhosmfaaw**
evmhosmfanw	04	rD	rA	rB	1	0	1	1	0	0	0	1	1	1	1	EVX	**evmhosmfanw**
evmhosmi	04	rD	rA	rB	1	0	0	0	0	0	0	1	1	0	1	EVX	**evmhosmi**
evmhosmia	04	rD	rA	rB	1	0	0	0	0	1	0	1	1	0	1	EVX	**evmhosmia**
evmhosmiaaw	04	rD	rA	rB	1	0	1	0	0	0	0	1	1	0	1	EVX	**evmhosmiaaw**
evmhosmianw	04	rD	rA	rB	1	0	1	1	0	0	0	1	1	0	1	EVX	**evmhosmianw**
evmhossf	04	rD	rA	rB	1	0	0	0	0	0	0	0	1	1	1	EVX	**evmhossf**
evmhossfa	04	rD	rA	rB	1	0	0	0	0	1	0	0	1	1	1	EVX	**evmhossfa**
evmhossfaaw	04	rD	rA	rB	1	0	1	0	0	0	0	0	1	1	1	EVX	**evmhossfaaw**
evmhossfanw	04	rD	rA	rB	1	0	1	1	0	0	0	0	1	1	1	EVX	**evmhossfanw**
evmhossiaaw	04	rD	rA	rB	1	0	1	0	0	0	0	0	1	0	1	EVX	**evmhossiaaw**
evmhossianw	04	rD	rA	rB	1	0	1	1	0	0	0	0	1	0	1	EVX	**evmhossianw**
evmhoumi	04	rD	rA	rB	1	0	0	0	0	0	0	1	1	0	0	EVX	**evmhoumi**
evmhoumia	04	rD	rA	rB	1	0	0	0	0	1	0	1	1	0	0	EVX	**evmhoumia**
evmhoumiaaw	04	rD	rA	rB	1	0	1	0	0	0	0	1	1	0	0	EVX	**evmhoumiaaw**
evmhoumianw	04	rD	rA	rB	1	0	1	1	0	0	0	1	1	0	0	EVX	**evmhoumianw**
evmhousiaaw	04	rD	rA	rB	1	0	1	0	0	0	0	0	1	0	0	EVX	**evmhousiaaw**
evmhousianw	04	rD	rA	rB	1	0	1	1	0	0	0	0	1	0	0	EVX	**evmhousianw**
evmra	04	rD	rA	///	1	0	0	1	1	0	0	0	1	0	0	EVX	**evmra**
evmwhsmf	04	rD	rA	rB	1	0	0	0	1	0	0	1	1	1	1	EVX	**evmwhsmf**
evmwhsmfa	04	rD	rA	rB	1	0	0	0	1	1	0	1	1	1	1	EVX	**evmwhsmfa**
evmwhsmi	04	rD	rA	rB	1	0	0	0	1	0	0	1	1	0	1	EVX	**evmwhsmi**
evmwhsmia	04	rD	rA	rB	1	0	0	0	1	1	0	1	1	0	1	EVX	**evmwhsmia**
evmwhssf	04	rD	rA	rB	1	0	0	0	1	0	0	0	1	1	1	EVX	**evmwhssf**
evmwhssfa	04	rD	rA	rB	1	0	0	0	1	1	0	0	1	1	1	EVX	**evmwhssfa**

Table D-2. Instructions (Decimal and Hexadecimal) by Opcode

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evmwhumi	04	rD	rA	rB	1	0	0	0	1	0	0	1	1	0	0	EVX	**evmwhumi**
evmwhumia	04	rD	rA	rB	1	0	0	0	1	1	0	1	1	0	0	EVX	**evmwhumia**
evmwhusiaaw	04	rD	rA	rB	1	0	1	0	1	0	0	0	1	0	0	EVX	**evmwhusiaaw**
evmwhusianw	04	rD	rA	rB	1	0	1	1	1	0	0	0	1	0	0	EVX	**evmwhusianw**
evmwlumi	04	rD	rA	rB	1	0	0	0	1	0	0	1	0	0	0	EVX	**evmwlumi**
evmwlumia	04	rD	rA	rB	1	0	0	0	1	1	0	1	0	0	0	EVX	**evmwlumia**
evmwlumiaaw	04	rD	rA	rB	1	0	1	0	1	0	0	1	0	0	0	EVX	**evmwlumiaaw**
evmwlumianw	04	rD	rA	rB	1	0	1	1	1	0	0	1	0	0	0	EVX	**evmwlumianw**
evmwlusiaaw	04	rD	rA	rB	1	0	1	0	1	0	0	0	0	0	0	EVX	**evmwlusiaaw**
evmwlusianw	04	rD	rA	rB	1	0	1	1	1	0	0	0	0	0	0	EVX	**evmwlusianw**
evmwsmf	04	rD	rA	rB	1	0	0	0	1	0	1	1	0	1	1	EVX	**evmwsmf**
evmwsmfa	04	rD	rA	rB	1	0	0	0	1	1	1	1	0	1	1	EVX	**evmwsmfa**
evmwsmfaa	04	rD	rA	rB	1	0	1	0	1	0	1	1	0	1	1	EVX	**evmwsmfaa**
evmwsmfan	04	rD	rA	rB	1	0	1	1	1	0	1	1	0	1	1	EVX	**evmwsmfan**
evmwsmi	04	rD	rA	rB	1	0	0	0	1	0	1	1	0	0	1	EVX	**evmwsmi**
evmwsmia	04	rD	rA	rB	1	0	0	0	1	1	1	1	0	0	1	EVX	**evmwsmia**
evmwsmiaa	04	rD	rA	rB	1	0	1	0	1	0	1	1	0	0	1	EVX	**evmwsmiaa**
evmwsmian	04	rD	rA	rB	1	0	1	1	1	0	1	1	0	0	1	EVX	**evmwsmian**
evmwssf	04	rD	rA	rB	1	0	0	0	1	0	1	0	0	1	1	EVX	**evmwssf**
evmwssfa	04	rD	rA	rB	1	0	0	0	1	1	1	0	0	1	1	EVX	**evmwssfa**
evmwssfaa	04	rD	rA	rB	1	0	1	0	1	0	1	0	0	1	1	EVX	**evmwssfaa**
evmwssfan	04	rD	rA	rB	1	0	1	1	1	0	1	0	0	1	1	EVX	**evmwssfan**
evmwumi	04	rD	rA	rB	1	0	0	0	1	0	1	1	0	0	0	EVX	**evmwumi**
evmwumia	04	rD	rA	rB	1	0	0	0	1	1	1	1	0	0	0	EVX	**evmwumia**
evmwumiaa	04	rD	rA	rB	1	0	1	0	1	0	1	1	0	0	0	EVX	**evmwumiaa**
evmwumian	04	rD	rA	rB	1	0	1	1	1	0	1	1	0	0	0	EVX	**evmwumian**
evnand	04	rD	rA	rB	0	1	0	0	0	0	1	1	1	1	0	EVX	**evnand**
evneg	04	rD	rA	///	0	1	0	0	0	0	0	1	0	0	1	EVX	**evneg**
evnor	04	rD	rA	rB	0	1	0	0	0	0	1	1	0	0	0	EVX	**evnor**
evor	04	rD	rA	rB	0	1	0	0	0	0	1	0	1	1	1	EVX	**evor**
evorc	04	rD	rA	rB	0	1	0	0	0	0	1	1	0	1	1	EVX	**evorc**
evrlw	04	rD	rA	rB	0	1	0	0	0	1	0	1	0	0	0	EVX	**evrlw**
evrlwi	04	rD	rA	UIMM	0	1	0	0	0	1	0	1	0	1	0	EVX	**evrlwi**

Table D-2. Instructions (Decimal and Hexadecimal) by Opcode

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30 31	Form	Mnemonic
evrndw	04	rD	rA	UIMM	0 1 0 0 0 0 0 1 1 0 0	EVX	**evrndw**
evsel	04	rD	rA	rB	0 1 0 0 1 1 1 1 crfS	EVX	**evsel**
evslw	04	rD	rA	rB	0 1 0 0 0 1 0 0 1 0 0	EVX	**evslw**
evslwi	04	rD	rA	UIMM	0 1 0 0 0 1 0 0 1 1 0	EVX	**evslwi**
evsplatfi	04	rD	SIMM	///	0 1 0 0 0 1 0 1 0 1 1	EVX	**evsplatfi**
evsplati	04	rD	SIMM	///	0 1 0 0 0 1 0 1 0 0 1	EVX	**evsplati**
evsrwis	04	rD	rA	UIMM	0 1 0 0 0 1 0 0 0 1 1	EVX	**evsrwis**
evsrwiu	04	rD	rA	UIMM	0 1 0 0 0 1 0 0 0 1 0	EVX	**evsrwiu**
evsrws	04	rD	rA	rB	0 1 0 0 0 1 0 0 0 0 1	EVX	**evsrws**
evsrwu	04	rD	rA	rB	0 1 0 0 0 1 0 0 0 0 0	EVX	**evsrwu**
evstdd	04	rD	rA	UIMM [1]	0 1 1 0 0 1 0 0 0 0 1	EVX	**evstdd**
evstddx	04	rS	rA	rB	0 1 1 0 0 1 0 0 0 0 0	EVX	**evstddx**
evstdh	04	rS	rA	UIMM [1]	0 1 1 0 0 1 0 0 1 0 1	EVX	**evstdh**
evstdhx	04	rS	rA	rB	0 1 1 0 0 1 0 0 1 0 0	EVX	**evstdhx**
evstdw	04	rS	rA	UIMM [1]	0 1 1 0 0 1 0 0 0 1 1	EVX	**evstdw**
evstdwx	04	rS	rA	rB	0 1 1 0 0 1 0 0 0 1 0	EVX	**evstdwx**
evstwhe	04	rS	rA	UIMM [3]	0 1 1 0 0 1 1 0 0 0 1	EVX	**evstwhe**
evstwhex	04	rS	rA	rB	0 1 1 0 0 1 1 0 0 0 0	EVX	**evstwhex**
evstwho	04	rS	rA	UIMM [3]	0 1 1 0 0 1 1 0 1 0 1	EVX	**evstwho**
evstwhox	04	rS	rA	rB	0 1 1 0 0 1 1 0 1 0 0	EVX	**evstwhox**
evstwwe	04	rS	rA	UIMM [3]	0 1 1 0 0 1 1 1 0 0 1	EVX	**evstwwe**
evstwwex	04	rS	rA	rB	0 1 1 0 0 1 1 1 0 0 0	EVX	**evstwwex**
evstwwo	04	rS	rA	UIMM [3]	0 1 1 0 0 1 1 1 1 0 1	EVX	**evstwwo**
evstwwox	04	rS	rA	rB	0 1 1 0 0 1 1 1 1 0 0	EVX	**evstwwox**
evsubfsmiaaw	04	rD	rA	///	1 0 0 1 1 0 0 1 0 1 1	EVX	**evsubfsmiaaw**
evsubfssiaaw	04	rD	rA	///	1 0 0 1 1 0 0 0 0 1 1	EVX	**evsubfssiaaw**
evsubfumiaaw	04	rD	rA	///	1 0 0 1 1 0 0 1 0 1 0	EVX	**evsubfumiaaw**
evsubfusiaaw	04	rD	rA	///	1 0 0 1 1 0 0 0 0 1 0	EVX	**evsubfusiaaw**
evsubfw	04	rD	rA	rB	0 1 0 0 0 0 0 0 1 0 0	EVX	**evsubfw**
evsubifw	04	rD	UIMM	rB	0 1 0 0 0 0 0 0 1 1 0	EVX	**evsubifw**
evxor	04	rD	rA	rB	0 1 0 0 0 0 1 0 1 1 0	EVX	**evxor**
mulli	07	rD	rA	SIMM		D	**mulli**
subfic	08	rD	rA	SIMM		D	**subfic**

Table D-2. Instructions (Decimal and Hexadecimal) by Opcode

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
cmpli	10 (0x0A)						crfD			/	L	rA					UIMM																D	**cmpli**
cmpi	11 (0x0B)						crfD			/	L	rA					SIMM																D	**cmpi**
addic	12 (0x0C)						rD					rA					SIMM																D	**addic**
addic.	13 (0x0D)						rD					rA					SIMM																D	**addic.**
addi	14 (0x0E)						rD					rA					SIMM																D	**addi**
addis	15 (0x0F)						rD					rA					SIMM																D	**addis**
bc	16 (0x10)						BO					BI					BD														0	0	B	**bc**
bca	16 (0x10)						BO					BI					BD														1	0	B	**bca**
bcl	16 (0x10)						BO					BI					BD														0	1	B	**bcl**
bcla	16 (0x10)						BO					BI					BD														1	1	B	**bcla**
sc	17 (0x11)						///																								1	/	SC	**sc**
b	18 (0x12)						LI																								0	0	I	**b**
ba	18 (0x12)						LI																								1	0	I	**ba**
bl	18 (0x12)						LI																								0	1	I	**bl**
bla	18 (0x12)						LI																								1	1	I	**bla**
bcctr	19 (0x13)						BO					BI					///					1	0	0	0	0	1	0	0	0	0	0	XL	**bcctr**
bcctrl	19 (0x13)						BO					BI					///					1	0	0	0	0	1	0	0	0	0	1	XL	**bcctrl**
bclr	19 (0x13)						BO					BI					///					0	0	0	0	0	1	0	0	0	0	0	XL	**bclr**
bclrl	19 (0x13)						BO					BI					///					0	0	0	0	0	1	0	0	0	0	1	XL	**bclrl**
crand	19 (0x13)						crbD					crbA					crbB					0	1	0	0	0	0	0	0	0	1	/	XL	**crand**
crandc	19 (0x13)						crbD					crbA					crbB					0	0	1	0	0	0	0	0	0	1	/	XL	**crandc**
creqv	19 (0x13)						crbD					crbA					crbB					0	1	0	0	1	0	0	0	0	1	/	XL	**creqv**
crnand	19 (0x13)						crbD					crbA					crbB					0	0	1	1	1	0	0	0	0	1	/	XL	**crnand**
crnor	19 (0x13)						crbD					crbA					crbB					0	0	0	0	1	0	0	0	0	1	/	XL	**crnor**
cror	19 (0x13)						crbD					crbA					crbB					0	1	1	1	0	0	0	0	0	1	/	XL	**cror**
crorc	19 (0x13)						crbD					crbA					crbB					0	1	1	0	1	0	0	0	0	1	/	XL	**crorc**
crxor	19 (0x13)						crbD					crbA					crbB					0	0	1	1	0	0	0	0	0	1	/	XL	**crxor**
isync	19 (0x13)						///															0	0	1	0	0	1	0	1	1	0	/	XL	**isync**
mcrf	19 (0x13)						crfD			//		crfS			///							0	0	0	0	0	0	0	0	0	0	/	XL	**mcrf**
rfci	19 (0x13)						///															0	0	0	0	1	1	0	0	1	1	/	XL	**rfci**
rfi	19 (0x13)						///															0	0	0	0	1	1	0	0	1	0	/	XL	**rfi**
rfmci	19 (0x13)						///															0	0	0	0	1	0	0	1	1	0	/	XL	**rfmci**
rlwimi	20 (0x14)						rS					rA					SH					MB					ME					Rc	M	**rlwimi**

Table D-2. Instructions (Decimal and Hexadecimal) by Opcode

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25	26 27 28 29 30	31	Form	Mnemonic
rlwimi.	20 (0x14)	rS	rA	SH	MB	ME	Rc	M	**rlwimi.**
rlwinm	21 (0x15)	rS	rA	SH	MB	ME	0	M	**rlwinm**
rlwinm.	21 (0x15)	rS	rA	SH	MB	ME	1	M	**rlwinm.**
rlwnm	23 (0x17)	rS	rA	rB	MB	ME	Rc	M	**rlwnm**
rlwnm.	23 (0x17)	rS	rA	rB	MB	ME	Rc	M	**rlwnm.**
ori	24 (0x18)	rS	rA	UIMM				D	**ori**
oris	25 (0x19)	rS	rA	UIMM				D	**oris**
xori	26 (0x1A)	rS	rA	UIMM				D	**xori**
xoris	27 (0x1B)	rS	rA	UIMM				D	**xoris**
andi.	28 (0x1C)	rS	rA	UIMM				D	**andi.**
andis.	29 (0x1D)	rS	rA	UIMM				D	**andis.**
add	31 (0x1F)	rD	rA	rB	0 1 0 0 0	0 1 0 1 0	0	X	**add**
add.	31 (0x1F)	rD	rA	rB	0 1 0 0 0	0 1 0 1 0	1	X	**add.**
addc	31 (0x1F)	rD	rA	rB	0 0 0 0 0	0 1 0 1 0	0	X	**addc**
addc.	31 (0x1F)	rD	rA	rB	0 0 0 0 0	0 1 0 1 0	1	X	**addc.**
addco	31 (0x1F)	rD	rA	rB	1 0 0 0 0	0 1 0 1 0	0	X	**addco**
addco.	31 (0x1F)	rD	rA	rB	1 0 0 0 0	0 1 0 1 0	1	X	**addco.**
adde	31 (0x1F)	rD	rA	rB	0 0 1 0 0	0 1 0 1 0	0	X	**adde**
adde.	31 (0x1F)	rD	rA	rB	0 0 1 0 0	0 1 0 1 0	1	X	**adde.**
addeo	31 (0x1F)	rD	rA	rB	1 0 1 0 0	0 1 0 1 0	0	X	**addeo**
addeo.	31 (0x1F)	rD	rA	rB	1 0 1 0 0	0 1 0 1 0	1	X	**addeo.**
addme	31 (0x1F)	rD	rA	///	0 0 1 1 1	0 1 0 1 0	0	X	**addme**
addme.	31 (0x1F)	rD	rA	///	0 0 1 1 1	0 1 0 1 0	1	X	**addme.**
addmeo	31 (0x1F)	rD	rA	///	1 0 1 1 1	0 1 0 1 0	0	X	**addmeo**
addmeo.	31 (0x1F)	rD	rA	///	1 0 1 1 1	0 1 0 1 0	1	X	**addmeo.**
addo	31 (0x1F)	rD	rA	rB	1 1 0 0 0	0 1 0 1 0	0	X	**addo**
addo.	31 (0x1F)	rD	rA	rB	1 1 0 0 0	0 1 0 1 0	1	X	**addo.**
addze	31 (0x1F)	rD	rA	///	0 0 1 1 0	0 1 0 1 0	0	X	**addze**
addze.	31 (0x1F)	rD	rA	///	0 0 1 1 0	0 1 0 1 0	1	X	**addze.**
addzeo	31 (0x1F)	rD	rA	///	1 0 1 1 0	0 1 0 1 0	0	X	**addzeo**
addzeo.	31 (0x1F)	rD	rA	///	1 0 1 1 0	0 1 0 1 0	1	X	**addzeo.**
and	31 (0x1F)	rS	rA	rB	0 0 0 0 0	1 1 1 0 0	0	X	**and**
and.	31 (0x1F)	rS	rA	rB	0 0 0 0 0	1 1 1 0 0	1	X	**and.**

Table D-2. Instructions (Decimal and Hexadecimal) by Opcode

Mnemonic	0–5	6–10	11–15	16–20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
andc	31 (0x1F)	rS	rA	rB	0	0	0	0	1	1	1	1	0	0	0	X	**andc**
andc.	31 (0x1F)	rS	rA	rB	0	0	0	0	1	1	1	1	0	0	1	X	**andc.**
bbelr	31 (0x1F)	///			1	0	0	0	1	0	0	1	1	0	0	X	**bbelr**
bblels	31 (0x1F)	///			1	0	0	0	1	0	0	1	1	0	0	X	**bblels**
cmp	31 (0x1F)	crfD / L	rA	rB	0	0	0	0	0	0	0	0	0	0	/	X	**cmp**
cmpl	31 (0x1F)	crfD / L	rA	rB	0	0	0	0	1	0	0	0	0	0	/	X	**cmpl**
cntlzw	31 (0x1F)	rS	rA	///	0	0	0	0	0	1	1	0	1	0	0	X	**cntlzw**
cntlzw.	31 (0x1F)	rS	rA	///	0	0	0	0	0	1	1	0	1	0	1	X	**cntlzw.**
dcba	31 (0x1F)	///	rA	rB	1	0	1	1	1	1	0	1	1	0	/	X	**dcba**
dcbf	31 (0x1F)	///	rA	rB	0	0	0	1	0	1	0	1	1	0	/	X	**dcbf**
dcbi	31 (0x1F)	///	rA	rB	0	1	1	1	0	1	0	1	1	0	/	X	**dcbi**
dcblc	31 (0x1F)	CT	rA	rB	0	1	1	0	0	0	0	1	1	0	0	X	**dcblc**
dcbst	31 (0x1F)	///	rA	rB	0	0	0	0	1	1	0	1	1	0	/	X	**dcbst**
dcbt	31 (0x1F)	CT	rA	rB	0	1	0	0	0	1	0	1	1	0	/	X	**dcbt**
dcbtls	31 (0x1F)	CT	rA	rB	0	0	1	0	1	0	0	1	1	0	0	X	**dcbtls**
dcbtst	31 (0x1F)	CT	rA	rB	0	0	1	1	1	1	0	1	1	0	/	X	**dcbtst**
dcbtstls	31 (0x1F)	CT	rA	rB	0	0	1	0	0	0	0	1	1	0	0	X	**dcbtstls**
dcbz	31 (0x1F)	///	rA	rB	1	1	1	1	1	1	0	1	1	0	/	X	**dcbz**
divw	31 (0x1F)	rD	rA	rB	0	1	1	1	1	0	1	0	1	1	0	X	**divw**
divw.	31 (0x1F)	rD	rA	rB	0	1	1	1	1	0	1	0	1	1	1	X	**divw.**
divwo	31 (0x1F)	rD	rA	rB	1	1	1	1	1	0	1	0	1	1	0	X	**divwo**
divwo.	31 (0x1F)	rD	rA	rB	1	1	1	1	1	0	1	0	1	1	1	X	**divwo.**
divwu	31 (0x1F)	rD	rA	rB	0	1	1	1	0	0	1	0	1	1	0	X	**divwu**
divwu.	31 (0x1F)	rD	rA	rB	0	1	1	1	0	0	1	0	1	1	1	X	**divwu.**
divwuo	31 (0x1F)	rD	rA	rB	1	1	1	1	0	0	1	0	1	1	0	X	**divwuo**
divwuo.	31 (0x1F)	rD	rA	rB	1	1	1	1	0	0	1	0	1	1	1	X	**divwuo.**
eqv	31 (0x1F)	rD	rA	rB	0	1	0	0	0	1	1	1	0	0	0	X	**eqv**
eqv.	31 (0x1F)	rD	rA	rB	0	1	0	0	0	1	1	1	0	0	1	X	**eqv.**
extsb	31 (0x1F)	rS	rA	///	1	1	1	0	1	1	1	0	1	0	0	X	**extsb**
extsb.	31 (0x1F)	rS	rA	///	1	1	1	0	1	1	1	0	1	0	1	X	**extsb.**
extsh	31 (0x1F)	rS	rA	///	1	1	1	0	0	1	1	0	1	0	0	X	**extsh**
extsh.	31 (0x1F)	rS	rA	///	1	1	1	0	0	1	1	0	1	0	1	X	**extsh.**
icbi	31 (0x1F)	///	rA	rB	1	1	1	1	0	1	0	1	1	0	/	X	**icbi**

Table D-2. Instructions (Decimal and Hexadecimal) by Opcode

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30 31	Form	Mnemonic
icblc	31 (0x1F)	CT	rA	rB	0 0 1 1 1 0 0 1 1 0 0	X	icblc
icbt	31 (0x1F)	CT	rA	rB	0 0 0 0 0 1 0 1 1 0 /	X	icbt
icbtls	31 (0x1F)	CT	rA	rB	0 1 1 1 1 0 0 1 1 0 0	X	icbtls
isel	31 (0x1F)	rD	rA	rB	crb | 0 1 1 1 1 0	X	isel
lbzux	31 (0x1F)	rD	rA	rB	0 0 0 1 1 1 0 1 1 1 /	X	lbzux
lbzx	31 (0x1F)	rD	rA	rB	0 0 0 1 0 1 0 1 1 1 /	X	lbzx
lhaux	31 (0x1F)	rD	rA	rB	0 1 0 1 1 1 0 1 1 1 /	X	lhaux
lhax	31 (0x1F)	rD	rA	rB	0 1 0 1 0 1 0 1 1 1 /	X	lhax
lhbrx	31 (0x1F)	rD	rA	rB	1 1 0 0 0 1 0 1 1 0 /	X	lhbrx
lhzux	31 (0x1F)	rD	rA	rB	0 1 0 0 1 1 0 1 1 1 /	X	lhzux
lhzx	31 (0x1F)	rD	rA	rB	0 1 0 0 0 1 0 1 1 1 /	X	lhzx
lwarx	31 (0x1F)	rD	rA	rB	0 0 0 0 0 1 0 1 0 0 /	X	lwarx
lwbrx	31 (0x1F)	rD	rA	rB	1 0 0 0 0 1 0 1 1 0 /	X	lwbrx
lwzux	31 (0x1F)	rD	rA	rB	0 0 0 0 1 1 0 1 1 1 /	X	lwzux
lwzx	31 (0x1F)	rD	rA	rB	0 0 0 0 0 1 0 1 1 1 /	X	lwzx
mbar	31 (0x1F)	MO	///		1 1 0 1 0 1 0 1 1 0 /	X	mbar
mcrxr	31 (0x1F)	crfD	///		1 0 0 0 0 0 0 0 0 0 /	X	mcrxr
mfcr	31 (0x1F)	rD	///		0 0 0 0 0 1 0 0 1 1 /	X	mfcr
mfmsr	31 (0x1F)	rD	///		0 0 0 1 0 1 0 0 1 1 /	X	mfmsr
mfpmr	31 (0x1F)	rD	PMRN5–9	PMRN0–4	0 1 0 1 0 0 1 1 1 0 0	XFX	mfpmr
mfspr	31 (0x1F)	rD	SPRN5–9	SPRN0–4	0 1 0 1 0 1 0 0 1 1 /	XFX	mfspr
msync	31 (0x1F)	///			1 0 0 1 0 1 0 1 1 0 /	X	msync
mtcrf	31 (0x1F)	rS	/ CRM	/	0 0 1 0 0 1 0 0 0 0 /	XFX	mtcrf
mtmsr	31 (0x1F)	rS	///		0 0 1 0 0 1 0 0 1 0 /	X	mtmsr
mtpmr	31 (0x1F)	rS	PMRN5–9	PMRN0–4	0 1 1 1 0 0 1 1 1 0 0	XFX	mtpmr
mtspr	31 (0x1F)	rS	SPRN5–9	SPRN0–4	0 1 1 1 0 1 0 0 1 1 /	XFX	mtspr
mulhw	31 (0x1F)	rD	rA	rB	/ 0 0 1 0 0 1 0 1 1 0	X	mulhw
mulhw.	31 (0x1F)	rD	rA	rB	/ 0 0 1 0 0 1 0 1 1 1	X	mulhw.
mulhwu	31 (0x1F)	rD	rA	rB	/ 0 0 0 0 0 1 0 1 1 0	X	mulhwu
mulhwu.	31 (0x1F)	rD	rA	rB	/ 0 0 0 0 0 1 0 1 1 1	X	mulhwu.
mullw	31 (0x1F)	rD	rA	rB	0 0 1 1 1 0 1 0 1 1 0	X	mullw
mullw.	31 (0x1F)	rD	rA	rB	0 0 1 1 1 0 1 0 1 1 1	X	mullw.
mullwo	31 (0x1F)	rD	rA	rB	1 0 1 1 1 0 1 0 1 1 0	X	mullwo

Table D-2. Instructions (Decimal and Hexadecimal) by Opcode

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
mullwo.	31 (0x1F)	rD	rA	rB	1	0	1	1	1	0	1	0	1	1	1	X	mullwo.
nand	31 (0x1F)	rS	rA	rB	0	1	1	1	0	1	1	1	0	0	0	X	nand
nand.	31 (0x1F)	rS	rA	rB	0	1	1	1	0	1	1	1	0	0	1	X	nand.
neg	31 (0x1F)	rD	rA	///	0	0	0	1	1	0	1	0	0	0	0	X	neg
neg.	31 (0x1F)	rD	rA	///	0	0	0	1	1	0	1	0	0	0	1	X	neg.
nego	31 (0x1F)	rD	rA	///	1	0	0	1	1	0	1	0	0	0	0	X	nego
nego.	31 (0x1F)	rD	rA	///	1	0	0	1	1	0	1	0	0	0	1	X	nego.
nor	31 (0x1F)	rS	rA	rB	0	0	0	1	1	1	1	1	0	0	0	X	nor
nor.	31 (0x1F)	rS	rA	rB	0	0	0	1	1	1	1	1	0	0	1	X	nor.
or	31 (0x1F)	rS	rA	rB	0	1	1	0	1	1	1	1	0	0	0	X	or
or.	31 (0x1F)	rS	rA	rB	0	1	1	0	1	1	1	1	0	0	1	X	or.
orc	31 (0x1F)	rS	rA	rB	0	1	1	0	0	1	1	1	0	0	0	X	orc
orc.	31 (0x1F)	rS	rA	rB	0	1	1	0	0	1	1	1	0	0	1	X	orc.
slw	31 (0x1F)	rS	rA	rB	0	0	0	0	0	1	1	0	0	0	0	X	slw
slw.	31 (0x1F)	rS	rA	rB	0	0	0	0	0	1	1	0	0	0	1	X	slw.
sraw	31 (0x1F)	rS	rA	rB	1	1	0	0	0	1	1	0	0	0	0	X	sraw
sraw.	31 (0x1F)	rS	rA	rB	1	1	0	0	0	1	1	0	0	0	1	X	sraw.
srawi	31 (0x1F)	rS	rA	SH	1	1	0	0	1	1	1	0	0	0	0	X	srawi
srawi.	31 (0x1F)	rS	rA	SH	1	1	0	0	1	1	1	0	0	0	1	X	srawi.
srw	31 (0x1F)	rS	rA	rB	1	0	0	0	0	1	1	0	0	0	0	X	srw
srw.	31 (0x1F)	rS	rA	rB	1	0	0	0	0	1	1	0	0	0	1	X	srw.
stbux	31 (0x1F)	rS	rA	rB	0	0	1	1	1	1	0	1	1	1	0	X	stbux
stbx	31 (0x1F)	rS	rA	rB	0	0	1	1	0	1	0	1	1	1	0	X	stbx
sthbrx	31 (0x1F)	rS	rA	rB	1	1	1	0	0	1	0	1	1	0	/	X	sthbrx
sthux	31 (0x1F)	rS	rA	rB	0	1	1	0	1	1	0	1	1	1	/	X	sthux
sthx	31 (0x1F)	rS	rA	rB	0	1	1	0	0	1	0	1	1	1	/	X	sthx
stwbrx	31 (0x1F)	rS	rA	rB	1	0	1	0	0	1	0	1	1	0	/	X	stwbrx
stwcx.	31 (0x1F)	rS	rA	rB	0	0	1	0	0	1	0	1	1	0	1	X	stwcx.
stwux	31 (0x1F)	rS	rA	rB	0	0	1	0	1	1	0	1	1	1	/	D	stwux
stwx	31 (0x1F)	rS	rA	rB	0	0	1	0	0	1	0	1	1	1	/	D	stwx
subf	31 (0x1F)	rD	rA	rB	0	0	0	0	1	0	1	0	0	0	0	X	subf
subf.	31 (0x1F)	rD	rA	rB	0	0	0	0	1	0	1	0	0	0	1	X	subf.
subfc	31 (0x1F)	rD	rA	rB	0	0	0	0	0	0	1	0	0	0	0	X	subfc

Table D-2. Instructions (Decimal and Hexadecimal) by Opcode

Mnemonic	0–5	6–10	11–15	16–20	21–30	31	Form	Mnemonic
subfc.	31 (0x1F)	rD	rA	rB	0 0 0 0 0 0 1 0 0 0	1	X	**subfc.**
subfco	31 (0x1F)	rD	rA	rB	1 0 0 0 0 0 1 0 0 0	0	X	**subfco**
subfco.	31 (0x1F)	rD	rA	rB	1 0 0 0 0 0 1 0 0 0	1	X	**subfco.**
subfe	31 (0x1F)	rD	rA	rB	0 0 1 0 0 0 1 0 0 0	0	X	**subfe**
subfe.	31 (0x1F)	rD	rA	rB	0 0 1 0 0 0 1 0 0 0	1	X	**subfe.**
subfeo	31 (0x1F)	rD	rA	rB	1 0 1 0 0 0 1 0 0 0	0	X	**subfeo**
subfeo.	31 (0x1F)	rD	rA	rB	1 0 1 0 0 0 1 0 0 0	1	X	**subfeo.**
subfme	31 (0x1F)	rD	rA	///	0 0 1 1 1 0 1 0 0 0	0	X	**subfme**
subfme.	31 (0x1F)	rD	rA	///	0 0 1 1 1 0 1 0 0 0	1	X	**subfme.**
subfmeo	31 (0x1F)	rD	rA	///	1 0 1 1 1 0 1 0 0 0	0	X	**subfmeo**
subfmeo.	31 (0x1F)	rD	rA	///	1 0 1 1 1 0 1 0 0 0	1	X	**subfmeo.**
subfo	31 (0x1F)	rD	rA	rB	1 0 0 0 1 0 1 0 0 0	0	X	**subfo**
subfo.	31 (0x1F)	rD	rA	rB	1 0 0 0 1 0 1 0 0 0	1	X	**subfo.**
subfze	31 (0x1F)	rD	rA	///	0 0 1 1 0 0 1 0 0 0	0	X	**subfze**
subfze.	31 (0x1F)	rD	rA	///	0 0 1 1 0 0 1 0 0 0	1	X	**subfze.**
subfzeo	31 (0x1F)	rD	rA	///	1 0 1 1 0 0 1 0 0 0	0	X	**subfzeo**
subfzeo.	31 (0x1F)	rD	rA	///	1 0 1 1 0 0 1 0 0 0	1	X	**subfzeo.**
tlbivax	31 (0x1F)	///	rA	rB	1 1 0 0 0 1 0 0 1 0	/	X	**tlbivax**
tlbre	31 (0x1F)	///[4]			1 1 1 0 1 1 0 0 1 0	/	X	**tlbre**
tlbsx	31 (0x1F)	/// [4]	rA	rB	1 1 1 0 0 1 0 0 1 0	/[4]	X	**tlbsx**
tlbsync	31 (0x1F)	///			1 0 0 0 1 1 0 1 1 0	/	X	**tlbsync**
tlbwe	31 (0x1F)	/// [4]			1 1 1 1 0 1 0 0 1 0	/	X	**tlbwe**
tw	31 (0x1F)	TO	rA	rB	0 0 0 0 0 0 0 1 0 0	/	X	**tw**
wrtee	31 (0x1F)	rS	///		0 0 1 0 0 0 0 0 1 1	/	X	**wrtee**
wrteei	31 (0x1F)	///		E ///	0 0 1 0 1 0 0 0 1 1	/	X	**wrteei**
xor	31 (0x1F)	rS	rA	rB	0 1 0 0 1 1 1 1 0 0	0	X	**xor**
xor.	31 (0x1F)	rS	rA	rB	0 1 0 0 1 1 1 1 0 0	1	X	**xor.**
lwz	32 (0x20)	rD	rA	D			D	**lwz**
lwzu	33 (0x21)	rD	rA	D			D	**lwzu**
lbz	34 (0x22)	rD	rA	D			D	**lbz**
lbzu	35 (0x23)	rD	rA	D			D	**lbzu**
stw	36 (0x24)	rS	rA	D			D	**stw**
stwu	37 (0x25)	rS	rA	D			D	**stwu**

Table D-2. Instructions (Decimal and Hexadecimal) by Opcode

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31	Form	Mnemonic
stb	38 (0x26)	rS	rA	D	D	**stb**
stbu	39 (0x27)	rS	rA	D	D	**stbu**
lhz	40 (0x28)	rD	rA	D	D	**lhz**
lhzu	41 (0x29)	rD	rA	D	D	**lhzu**
lha	42 (0x2A)	rD	rA	D	D	**lha**
lhau	43 (0x2B)	rD	rA	D	D	**lhau**
sth	44 (0x2C)	rS	rA	D	D	**sth**
sthu	45 (0x2D)	rS	rA	D	D	**sthu**
lmw	46 (0x2E)	rD	rA	D	D	**lmw**
stmw	47 (0x2F)	rS	rA	D	D	**stmw**

1 d = UIMM * 8
2 d = UIMM * 2
3 d = UIMM * 4
4 This field is defined as allocated by the Book E architecture for possible use in an implementation. These bits are not implemented in the e500.

D.3 Instructions by Form

Table D-3 lists e500 instructions by form.

Table D-3. Instructions (Binary) by Form

Mnemonic	0	1	2	3	4	5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
add	0	1	1	1	1	1	rD	rA	rB	0	1	0	0	0	0	1	0	1	0	0	X	**add**
add.	0	1	1	1	1	1	rD	rA	rB	0	1	0	0	0	0	1	0	1	0	1	X	**add.**
addc	0	1	1	1	1	1	rD	rA	rB	0	0	0	0	0	0	1	0	1	0	0	X	**addc**
addc.	0	1	1	1	1	1	rD	rA	rB	0	0	0	0	0	0	1	0	1	0	1	X	**addc.**
addco	0	1	1	1	1	1	rD	rA	rB	1	0	0	0	0	0	1	0	1	0	0	X	**addco**
addco.	0	1	1	1	1	1	rD	rA	rB	1	0	0	0	0	0	1	0	1	0	1	X	**addco.**
adde	0	1	1	1	1	1	rD	rA	rB	0	0	1	0	0	0	1	0	1	0	0	X	**adde**
adde.	0	1	1	1	1	1	rD	rA	rB	0	0	1	0	0	0	1	0	1	0	1	X	**adde.**
addeo	0	1	1	1	1	1	rD	rA	rB	1	0	1	0	0	0	1	0	1	0	0	X	**addeo**
addeo.	0	1	1	1	1	1	rD	rA	rB	1	0	1	0	0	0	1	0	1	0	1	X	**addeo.**
addme	0	1	1	1	1	1	rD	rA	///	0	0	1	1	1	0	1	0	1	0	0	X	**addme**
addme.	0	1	1	1	1	1	rD	rA	///	0	0	1	1	1	0	1	0	1	0	1	X	**addme.**
addmeo	0	1	1	1	1	1	rD	rA	///	1	0	1	1	1	0	1	0	1	0	0	X	**addmeo**
addmeo.	0	1	1	1	1	1	rD	rA	///	1	0	1	1	1	0	1	0	1	0	1	X	**addmeo.**

Table D-3. Instructions (Binary) by Form

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30	31	Form	Mnemonic
addo	0 1 1 1 1 1	rD	rA	rB	1 1 0 0 0 0 1 0 1 0	0	X	**addo**
addo.	0 1 1 1 1 1	rD	rA	rB	1 1 0 0 0 0 1 0 1 0	1	X	**addo.**
addze	0 1 1 1 1 1	rD	rA	///	0 0 1 1 0 0 1 0 1 0	0	X	**addze**
addze.	0 1 1 1 1 1	rD	rA	///	0 0 1 1 0 0 1 0 1 0	1	X	**addze.**
addzeo	0 1 1 1 1 1	rD	rA	///	1 0 1 1 0 0 1 0 1 0	0	X	**addzeo**
addzeo.	0 1 1 1 1 1	rD	rA	///	1 0 1 1 0 0 1 0 1 0	1	X	**addzeo.**
and	0 1 1 1 1 1	rS	rA	rB	0 0 0 0 0 1 1 1 0 0	0	X	**and**
and.	0 1 1 1 1 1	rS	rA	rB	0 0 0 0 0 1 1 1 0 0	1	X	**and.**
andc	0 1 1 1 1 1	rS	rA	rB	0 0 0 0 1 1 1 1 0 0	0	X	**andc**
andc.	0 1 1 1 1 1	rS	rA	rB	0 0 0 0 1 1 1 1 0 0	1	X	**andc.**
bbelr	0 1 1 1 1 1		///		1 0 0 0 1 0 0 1 1 0	0	X	**bbelr**
bblels	0 1 1 1 1 1		///		1 0 0 0 1 0 0 1 1 0	0	X	**bblels**
cmp	0 1 1 1 1 1	crfD / L	rA	rB	0 0 0 0 0 0 0 0 0 0	/	X	**cmp**
cmpl	0 1 1 1 1 1	crfD / L	rA	rB	0 0 0 0 1 0 0 0 0 0	/	X	**cmpl**
cntlzw	0 1 1 1 1 1	rS	rA	///	0 0 0 0 0 1 1 0 1 0	0	X	**cntlzw**
cntlzw.	0 1 1 1 1 1	rS	rA	///	0 0 0 0 0 1 1 0 1 0	1	X	**cntlzw.**
dcba	0 1 1 1 1 1	///	rA	rB	1 0 1 1 1 1 0 1 1 0	/	X	**dcba**
dcbf	0 1 1 1 1 1	///	rA	rB	0 0 0 1 0 1 0 1 1 0	/	X	**dcbf**
dcbi	0 1 1 1 1 1	///	rA	rB	0 1 1 1 0 1 0 1 1 0	/	X	**dcbi**
dcblc	0 1 1 1 1 1	CT	rA	rB	0 1 1 0 0 0 0 1 1 0	0	X	**dcblc**
dcbst	0 1 1 1 1 1	///	rA	rB	0 0 0 0 1 1 0 1 1 0	/	X	**dcbst**
dcbt	0 1 1 1 1 1	CT	rA	rB	0 1 0 0 0 1 0 1 1 0	/	X	**dcbt**
dcbtls	0 1 1 1 1 1	CT	rA	rB	0 0 1 0 1 0 0 1 1 0	0	X	**dcbtls**
dcbtst	0 1 1 1 1 1	CT	rA	rB	0 0 1 1 1 1 0 1 1 0	/	X	**dcbtst**
dcbtstls	0 1 1 1 1 1	CT	rA	rB	0 0 1 0 0 0 0 1 1 0	0	X	**dcbtstls**
dcbz	0 1 1 1 1 1	///	rA	rB	1 1 1 1 1 1 0 1 1 0	/	X	**dcbz**
divw	0 1 1 1 1 1	rD	rA	rB	0 1 1 1 1 0 1 0 1 1	0	X	**divw**
divw.	0 1 1 1 1 1	rD	rA	rB	0 1 1 1 1 0 1 0 1 1	1	X	**divw.**
divwo	0 1 1 1 1 1	rD	rA	rB	1 1 1 1 1 0 1 0 1 1	0	X	**divwo**
divwo.	0 1 1 1 1 1	rD	rA	rB	1 1 1 1 1 0 1 0 1 1	1	X	**divwo.**
divwu	0 1 1 1 1 1	rD	rA	rB	0 1 1 1 0 0 1 0 1 1	0	X	**divwu**
divwu.	0 1 1 1 1 1	rD	rA	rB	0 1 1 1 0 0 1 0 1 1	1	X	**divwu.**
divwuo	0 1 1 1 1 1	rD	rA	rB	1 1 1 1 0 0 1 0 1 1	0	X	**divwuo**

Table D-3. Instructions (Binary) by Form

Mnemonic	0–5	6–10	11–15	16–20	21–31	Form	Mnemonic
divwuo.	0 1 1 1 1 1	rD	rA	rB	1 | 1 1 1 0 0 1 0 1 1 | 1	X	**divwuo.**
eqv	0 1 1 1 1 1	rD	rA	rB	0 1 0 0 0 1 1 1 0 0 0	X	**eqv**
eqv.	0 1 1 1 1 1	rD	rA	rB	0 1 0 0 0 1 1 1 0 0 1	X	**eqv.**
extsb	0 1 1 1 1 1	rS	rA	///	1 1 1 0 1 1 1 0 1 0 0	X	**extsb**
extsb.	0 1 1 1 1 1	rS	rA	///	1 1 1 0 1 1 1 0 1 0 1	X	**extsb.**
extsh	0 1 1 1 1 1	rS	rA	///	1 1 1 0 0 1 1 0 1 0 0	X	**extsh**
extsh.	0 1 1 1 1 1	rS	rA	///	1 1 1 0 0 1 1 0 1 0 1	X	**extsh.**
icbi	0 1 1 1 1 1	///	rA	rB	1 1 1 1 0 1 0 1 1 0 | /	X	**icbi**
icblc	0 1 1 1 1 1	CT	rA	rB	0 0 1 1 1 0 0 1 1 0 0	X	**icblc**
icbt	0 1 1 1 1 1	CT	rA	rB	0 0 0 0 0 1 0 1 1 0 | /	X	**icbt**
icbtls	0 1 1 1 1 1	CT	rA	rB	0 1 1 1 1 0 0 1 1 0 0	X	**icbtls**
isel	0 1 1 1 1 1	rD	rA	rB	crb | 0 1 1 1 1 0	X	**isel**
lbzux	0 1 1 1 1 1	rD	rA	rB	0 0 0 1 1 1 0 1 1 1 | /	X	**lbzux**
lbzx	0 1 1 1 1 1	rD	rA	rB	0 0 0 1 0 1 0 1 1 1 | /	X	**lbzx**
lhaux	0 1 1 1 1 1	rD	rA	rB	0 1 0 1 1 1 0 1 1 1 | /	X	**lhaux**
lhax	0 1 1 1 1 1	rD	rA	rB	0 1 0 1 0 1 0 1 1 1 | /	X	**lhax**
lhbrx	0 1 1 1 1 1	rD	rA	rB	1 1 0 0 0 1 0 1 1 0 | /	X	**lhbrx**
lhzux	0 1 1 1 1 1	rD	rA	rB	0 1 0 0 1 1 0 1 1 1 | /	X	**lhzux**
lhzx	0 1 1 1 1 1	rD	rA	rB	0 1 0 0 0 1 0 1 1 1 | /	X	**lhzx**
lwarx	0 1 1 1 1 1	rD	rA	rB	0 0 0 0 0 1 0 1 0 0 | /	X	**lwarx**
lwbrx	0 1 1 1 1 1	rD	rA	rB	1 0 0 0 0 1 0 1 1 0 | /	X	**lwbrx**
lwzux	0 1 1 1 1 1	rD	rA	rB	0 0 0 0 1 1 0 1 1 1 | /	X	**lwzux**
lwzx	0 1 1 1 1 1	rD	rA	rB	0 0 0 0 0 1 0 1 1 1 | /	X	**lwzx**
mbar	0 1 1 1 1 1	MO	///		1 1 0 1 0 1 0 1 1 0 | /	X	**mbar**
mcrxr	0 1 1 1 1 1	crfD | ///			1 0 0 0 0 0 0 0 0 0 | /	X	**mcrxr**
mfcr	0 1 1 1 1 1	rD	///		0 0 0 0 0 1 0 0 1 1 | /	X	**mfcr**
mfmsr	0 1 1 1 1 1	rD	///		0 0 0 1 0 1 0 0 1 1 | /	X	**mfmsr**
msync	0 1 1 1 1 1	///			1 0 0 1 0 1 0 1 1 0 | /	X	**msync**
mtmsr	0 1 1 1 1 1	rS	///		0 0 1 0 0 1 0 0 1 0 | /	X	**mtmsr**
mulhw	0 1 1 1 1 1	rD	rA	rB	/ | 0 0 1 0 0 1 0 1 1 | 0	X	**mulhw**
mulhw.	0 1 1 1 1 1	rD	rA	rB	/ | 0 0 1 0 0 1 0 1 1 | 1	X	**mulhw.**
mulhwu	0 1 1 1 1 1	rD	rA	rB	/ | 0 0 0 0 0 1 0 1 1 | 0	X	**mulhwu**
mulhwu.	0 1 1 1 1 1	rD	rA	rB	/ | 0 0 0 0 0 1 0 1 1 | 1	X	**mulhwu.**

Table D-3. Instructions (Binary) by Form

Mnemonic	0	1	2	3	4	5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
mullw	0	1	1	1	1	1	rD	rA	rB	0	0	1	1	1	0	1	0	1	1	0	X	mullw
mullw.	0	1	1	1	1	1	rD	rA	rB	0	0	1	1	1	0	1	0	1	1	1	X	mullw.
mullwo	0	1	1	1	1	1	rD	rA	rB	1	0	1	1	1	0	1	0	1	1	0	X	mullwo
mullwo.	0	1	1	1	1	1	rD	rA	rB	1	0	1	1	1	0	1	0	1	1	1	X	mullwo.
nand	0	1	1	1	1	1	rS	rA	rB	0	1	1	1	0	1	1	1	0	0	0	X	nand
nand.	0	1	1	1	1	1	rS	rA	rB	0	1	1	1	0	1	1	1	0	0	1	X	nand.
neg	0	1	1	1	1	1	rD	rA	///	0	0	0	1	1	0	1	0	0	0	0	X	neg
neg.	0	1	1	1	1	1	rD	rA	///	0	0	0	1	1	0	1	0	0	0	1	X	neg.
nego	0	1	1	1	1	1	rD	rA	///	1	0	0	1	1	0	1	0	0	0	0	X	nego
nego.	0	1	1	1	1	1	rD	rA	///	1	0	0	1	1	0	1	0	0	0	1	X	nego.
nor	0	1	1	1	1	1	rS	rA	rB	0	0	0	1	1	1	1	1	0	0	0	X	nor
nor.	0	1	1	1	1	1	rS	rA	rB	0	0	0	1	1	1	1	1	0	0	1	X	nor.
or	0	1	1	1	1	1	rS	rA	rB	0	1	1	0	1	1	1	1	0	0	0	X	or
or.	0	1	1	1	1	1	rS	rA	rB	0	1	1	0	1	1	1	1	0	0	1	X	or.
orc	0	1	1	1	1	1	rS	rA	rB	0	1	1	0	0	1	1	1	0	0	0	X	orc
orc.	0	1	1	1	1	1	rS	rA	rB	0	1	1	0	0	1	1	1	0	0	1	X	orc.
slw	0	1	1	1	1	1	rS	rA	rB	0	0	0	0	0	1	1	0	0	0	0	X	slw
slw.	0	1	1	1	1	1	rS	rA	rB	0	0	0	0	0	1	1	0	0	0	1	X	slw.
sraw	0	1	1	1	1	1	rS	rA	rB	1	1	0	0	0	1	1	0	0	0	0	X	sraw
sraw.	0	1	1	1	1	1	rS	rA	rB	1	1	0	0	0	1	1	0	0	0	1	X	sraw.
srawi	0	1	1	1	1	1	rS	rA	SH	1	1	0	0	1	1	1	0	0	0	0	X	srawi
srawi.	0	1	1	1	1	1	rS	rA	SH	1	1	0	0	1	1	1	0	0	0	1	X	srawi.
srw	0	1	1	1	1	1	rS	rA	rB	1	0	0	0	0	1	1	0	0	0	0	X	srw
srw.	0	1	1	1	1	1	rS	rA	rB	1	0	0	0	0	1	1	0	0	0	1	X	srw.
stbux	0	1	1	1	1	1	rS	rA	rB	0	0	1	1	1	1	0	1	1	1	0	X	stbux
stbx	0	1	1	1	1	1	rS	rA	rB	0	0	1	1	0	1	0	1	1	1	0	X	stbx
sthbrx	0	1	1	1	1	1	rS	rA	rB	1	1	1	0	0	1	0	1	1	0	/	X	sthbrx
sthux	0	1	1	1	1	1	rS	rA	rB	0	1	1	0	1	1	0	1	1	1	/	X	sthux
sthx	0	1	1	1	1	1	rS	rA	rB	0	1	1	0	0	1	0	1	1	1	/	X	sthx
stwbrx	0	1	1	1	1	1	rS	rA	rB	1	0	1	0	0	1	0	1	1	0	/	X	stwbrx
stwcx.	0	1	1	1	1	1	rS	rA	rB	0	0	1	0	0	1	0	1	1	0	1	X	stwcx.
subf	0	1	1	1	1	1	rD	rA	rB	0	0	0	0	1	0	1	0	0	0	0	X	subf
subf.	0	1	1	1	1	1	rD	rA	rB	0	0	0	0	1	0	1	0	0	0	1	X	subf.

Table D-3. Instructions (Binary) by Form

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
subfc	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	0	0	1	0	0	0	0	X	subfc
subfc.	0	1	1	1	1	1	rD					rA					rB					0	0	0	0	0	0	1	0	0	0	1	X	subfc.
subfco	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	1	0	0	0	0	X	subfco
subfco.	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	0	0	1	0	0	0	1	X	subfco.
subfe	0	1	1	1	1	1	rD					rA					rB					0	0	1	0	0	0	1	0	0	0	0	X	subfe
subfe.	0	1	1	1	1	1	rD					rA					rB					0	0	1	0	0	0	1	0	0	0	1	X	subfe.
subfeo	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	1	0	0	0	0	X	subfeo
subfeo.	0	1	1	1	1	1	rD					rA					rB					1	0	1	0	0	0	1	0	0	0	1	X	subfeo.
subfme	0	1	1	1	1	1	rD					rA					///					0	0	1	1	1	0	1	0	0	0	0	X	subfme
subfme.	0	1	1	1	1	1	rD					rA					///					0	0	1	1	1	0	1	0	0	0	1	X	subfme.
subfmeo	0	1	1	1	1	1	rD					rA					///					1	0	1	1	1	0	1	0	0	0	0	X	subfmeo
subfmeo.	0	1	1	1	1	1	rD					rA					///					1	0	1	1	1	0	1	0	0	0	1	X	subfmeo.
subfo	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	0	1	0	0	0	0	X	subfo
subfo.	0	1	1	1	1	1	rD					rA					rB					1	0	0	0	1	0	1	0	0	0	1	X	subfo.
subfze	0	1	1	1	1	1	rD					rA					///					0	0	1	1	0	0	1	0	0	0	0	X	subfze
subfze.	0	1	1	1	1	1	rD					rA					///					0	0	1	1	0	0	1	0	0	0	1	X	subfze.
subfzeo	0	1	1	1	1	1	rD					rA					///					1	0	1	1	0	0	1	0	0	0	0	X	subfzeo
subfzeo.	0	1	1	1	1	1	rD					rA					///					1	0	1	1	0	0	1	0	0	0	1	X	subfzeo.
tlbivax	0	1	1	1	1	1	///					rA					rB					1	1	0	0	0	1	0	0	1	0	/	X	tlbivax
tlbre	0	1	1	1	1	1	///[1]															1	1	1	0	1	1	0	0	1	0	/	X	tlbre
tlbsx	0	1	1	1	1	1	///					rA					rB					1	1	1	0	0	1	0	0	1	0	/	X	tlbsx
tlbsync	0	1	1	1	1	1	///															1	0	0	0	1	1	0	1	1	0	/	X	tlbsync
tlbwe	0	1	1	1	1	1	///															1	1	1	1	0	1	0	0	1	0	/	X	tlbwe
tw	0	1	1	1	1	1	TO					rA					rB					0	0	0	0	0	0	0	1	0	0	/	X	tw
wrtee	0	1	1	1	1	1	rS					///										0	0	1	0	0	0	0	0	1	1	/	X	wrtee
wrteei	0	1	1	1	1	1	///										E	///				0	0	1	0	1	0	0	0	1	1	/	X	wrteei
xor	0	1	1	1	1	1	rS					rA					rB					0	1	0	0	1	1	1	1	0	0	0	X	xor
xor.	0	1	1	1	1	1	rS					rA					rB					0	1	0	0	1	1	1	1	0	0	1	X	xor.
bc	0	1	0	0	0	0	BO					BI					BD														0	0	B	bc
bca	0	1	0	0	0	0	BO					BI					BD														1	0	B	bca
bcl	0	1	0	0	0	0	BO					BI					BD														0	1	B	bcl
bcla	0	1	0	0	0	0	BO					BI					BD														1	1	B	bcla
addi	0	0	1	1	1	0	rD					rA					SIMM																D	addi

Table D-3. Instructions (Binary) by Form

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
addic	0	0	1	1	0	0	rD					rA					SIMM																D	addic
addic.	0	0	1	1	0	1	rD					rA					SIMM																D	addic.
addis	0	0	1	1	1	1	rD					rA					SIMM																D	addis
andi.	0	1	1	1	0	0	rS					rA					UIMM																D	andi.
andis.	0	1	1	1	0	1	rS					rA					UIMM																D	andis.
cmpi	0	0	1	0	1	1	crfD			/	L	rA					SIMM																D	cmpi
cmpli	0	0	1	0	1	0	crfD			/	L	rA					UIMM																D	cmpli
lbz	1	0	0	0	1	0	rD					rA					D																D	lbz
lbzu	1	0	0	0	1	1	rD					rA					D																D	lbzu
lha	1	0	1	0	1	0	rD					rA					D																D	lha
lhau	1	0	1	0	1	1	rD					rA					D																D	lhau
lhz	1	0	1	0	0	0	rD					rA					D																D	lhz
lhzu	1	0	1	0	0	1	rD					rA					D																D	lhzu
lmw	1	0	1	1	1	0	rD					rA					D																D	lmw
lwz	1	0	0	0	0	0	rD					rA					D																D	lwz
lwzu	1	0	0	0	0	1	rD					rA					D																D	lwzu
mulli	0	0	0	1	1	1	rD					rA					SIMM																D	mulli
ori	0	1	1	0	0	0	rS					rA					UIMM																D	ori
oris	0	1	1	0	0	1	rS					rA					UIMM																D	oris
stb	1	0	0	1	1	0	rS					rA					D																D	stb
stbu	1	0	0	1	1	1	rS					rA					D																D	stbu
sth	1	0	1	1	0	0	rS					rA					D																D	sth
sthu	1	0	1	1	0	1	rS					rA					D																D	sthu
stmw	1	0	1	1	1	1	rS					rA					D																D	stmw
stw	1	0	0	1	0	0	rS					rA					D																D	stw
stwu	1	0	0	1	0	1	rS					rA					D																D	stwu
stwux	0	1	1	1	1	1	rS					rA					rB					0	0	1	0	1	1	0	1	1	1	/	D	stwux
stwx	0	1	1	1	1	1	rS					rA					rB					0	0	1	0	0	1	0	1	1	1	/	D	stwx
subfic	0	0	1	0	0	0	rD					rA					SIMM																D	subfic
twi	0	0	0	0	1	1	TO					rA					SIMM																D	twi
xori	0	1	1	0	1	0	rS					rA					UIMM																D	xori
xoris	0	1	1	0	1	1	rS					rA					UIMM																D	xoris
efdabs	0	0	0	1	0	0	rD					rA					///					0	1	0	1	1	1	0	0	1	0	0	EFX	efdabs

Table D-3. Instructions (Binary) by Form

Mnemonic	0 1 2 3 4 5	6 7 8	9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30 31	Form	Mnemonic
efdadd	0 0 0 1 0 0	rD		rA	rB	0 1 0 1 1 1 0 0 0 0 0	EFX	**efdadd**
efdcfs	0 0 0 1 0 0	rD		0 0 0 0 0	rB	0 1 0 1 1 1 0 1 1 1 1	EFX	**efdcfs**
efdcfsf	0 0 0 1 0 0	rD		///	rB	0 1 0 1 1 1 1 0 0 1 1	EFX	**efdcfsf**
efdcfsi	0 0 0 1 0 0	rD		///	rB	0 1 0 1 1 1 1 0 0 0 1	EFX	**efdcfsi**
efdcfuf	0 0 0 1 0 0	rD		///	rB	0 1 0 1 1 1 1 0 0 1 0	EFX	**efdcfuf**
efdcfui	0 0 0 1 0 0	rD		///	rB	0 1 0 1 1 1 1 0 0 0 0	EFX	**efdcfui**
efdcmpeq	0 0 0 1 0 0	crfD	/ /	rA	rB	0 1 0 1 1 1 0 1 1 1 0	EFX	**efdcmpeq**
efdcmpgt	0 0 0 1 0 0	crfD	/ /	rA	rB	0 1 0 1 1 1 0 1 1 0 0	EFX	**efdcmpgt**
efdcmplt	0 0 0 1 0 0	crfD	/ /	rA	rB	0 1 0 1 1 1 0 1 1 0 1	EFX	**efdcmplt**
efdctsf	0 0 0 1 0 0	rD		///	rB	0 1 0 1 1 1 1 0 1 1 1	EFX	**efdctsf**
efdctsi	0 0 0 1 0 0	rD		///	rB	0 1 0 1 1 1 1 0 1 0 1	EFX	**efdctsi**
efdctsiz	0 0 0 1 0 0	rD		///	rB	0 1 0 1 1 1 1 1 0 1 0	EFX	**efdctsiz**
efdctuf	0 0 0 1 0 0	rD		///	rB	0 1 0 1 1 1 1 0 1 1 0	EFX	**efdctuf**
efdctui	0 0 0 1 0 0	rD		///	rB	0 1 0 1 1 1 1 0 1 0 0	EFX	**efdctui**
efdctuiz	0 0 0 1 0 0	rD		///	rB	0 1 0 1 1 1 1 1 0 0 0	EFX	**efdctuiz**
efddiv	0 0 0 1 0 0	rD		rA	rB	0 1 0 1 1 1 0 1 0 0 1	EFX	**efddiv**
efdmul	0 0 0 1 0 0	rD		rA	rB	0 1 0 1 1 1 0 1 0 0 0	EFX	**efdmul**
efdnabs	0 0 0 1 0 0	rD		rA	///	0 1 0 1 1 1 0 0 1 0 1	EFX	**efdnabs**
efdneg	0 0 0 1 0 0	rD		rA	///	0 1 0 1 1 1 0 0 1 1 0	EFX	**efdneg**
efdsub	0 0 0 1 0 0	rD		rA	rB	0 1 0 1 1 1 0 0 0 0 1	EFX	**efdsub**
efdtsteq	0 0 0 1 0 0	crfD	/ /	rA	rB	0 1 0 1 1 1 1 1 1 1 0	EFX	**efdtsteq**
efdtstgt	0 0 0 1 0 0	crfD	/ /	rA	rB	0 1 0 1 1 1 1 1 1 0 0	EFX	**efdtstgt**
efdtstlt	0 0 0 1 0 0	crfD	/ /	rA	rB	0 1 0 1 1 1 1 1 1 0 1	EFX	**efdtstlt**
efsabs	0 0 0 1 0 0	rD		rA	///	0 1 0 1 1 0 0 0 1 0 0	EFX	**efsabs**
efsadd	0 0 0 1 0 0	rD		rA	rB	0 1 0 1 1 0 0 0 0 0 0	EFX	**efsadd**
efscfd	0 0 0 1 0 0	rD		0 0 0 0 0	rB	0 1 0 1 1 0 0 1 1 1 1	EFX	**efscfd**
efscfsf	0 0 0 1 0 0	rD		///	rB	0 1 0 1 1 0 1 0 0 1 1	EFX	**efscfsf**
efscfsi	0 0 0 1 0 0	rD		///	rB	0 1 0 1 1 0 1 0 0 0 1	EFX	**efscfsi**
efscfuf	0 0 0 1 0 0	rD		///	rB	0 1 0 1 1 0 1 0 0 1 0	EFX	**efscfuf**
efscfui	0 0 0 1 0 0	rD		///	rB	0 1 0 1 1 0 1 0 0 0 0	EFX	**efscfui**
efscmpeq	0 0 0 1 0 0	crfD	/ /	rA	rB	0 1 0 1 1 0 0 1 1 1 0	EFX	**efscmpeq**
efscmpgt	0 0 0 1 0 0	crfD	/ /	rA	rB	0 1 0 1 1 0 0 1 1 0 0	EFX	**efscmpgt**
efscmplt	0 0 0 1 0 0	crfD	/ /	rA	rB	0 1 0 1 1 0 0 1 1 0 1	EFX	**efscmplt**

Table D-3. Instructions (Binary) by Form

Mnemonic	0	1	2	3	4	5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
efsctsf	0	0	0	1	0	0	rD	///	rB	0	1	0	1	1	0	1	0	1	1	1	EFX	**efsctsf**
efsctsi	0	0	0	1	0	0	rD	///	rB	0	1	0	1	1	0	1	0	1	0	1	EFX	**efsctsi**
efsctsiz	0	0	0	1	0	0	rD	///	rB	0	1	0	1	1	0	1	1	0	1	0	EFX	**efsctsiz**
efsctuf	0	0	0	1	0	0	rD	///	rB	0	1	0	1	1	0	1	0	1	1	0	EFX	**efsctuf**
efsctui	0	0	0	1	0	0	rD	///	rB	0	1	0	1	1	0	1	0	1	0	0	EFX	**efsctui**
efsctuiz	0	0	0	1	0	0	rD	///	rB	0	1	0	1	1	0	1	1	0	0	0	EFX	**efsctuiz**
efsdiv	0	0	0	1	0	0	rD	rA	rB	0	1	0	1	1	0	0	1	0	0	1	EFX	**efsdiv**
efsmul	0	0	0	1	0	0	rD	rA	rB	0	1	0	1	1	0	0	1	0	0	0	EFX	**efsmul**
efsnabs	0	0	0	1	0	0	rD	rA	///	0	1	0	1	1	0	0	0	1	0	1	EFX	**efsnabs**
efsneg	0	0	0	1	0	0	rD	rA	///	0	1	0	1	1	0	0	0	1	1	0	EFX	**efsneg**
efssub	0	0	0	1	0	0	rD	rA	rB	0	1	0	1	1	0	0	0	0	0	1	EFX	**efssub**
efststeq	0	0	0	1	0	0	crfD / /	rA	rB	0	1	0	1	1	0	1	1	1	1	0	EFX	**efststeq**
efststgt	0	0	0	1	0	0	crfD / /	rA	rB	0	1	0	1	1	0	1	1	1	0	0	EFX	**efststgt**
efststlt	0	0	0	1	0	0	crfD / /	rA	rB	0	1	0	1	1	0	1	1	1	0	1	EFX	**efststlt**
brinc[2]	0	0	0	1	0	0	rD	rA	rB	0	1	0	0	0	0	0	1	1	1	1	EVX	**brinc**
evabs	0	0	0	1	0	0	rD	rA	///	0	1	0	0	0	0	0	1	0	0	0	EVX	**evabs**
evaddiw	0	0	0	1	0	0	rD	UIMM	rB	0	1	0	0	0	0	0	0	0	1	0	EVX	**evaddiw**
evaddsmiaaw	0	0	0	1	0	0	rD	rA	///	1	0	0	1	1	0	0	1	0	0	1	EVX	**evaddsmiaaw**
evaddssiaaw	0	0	0	1	0	0	rD	rA	///	1	0	0	1	1	0	0	0	0	0	1	EVX	**evaddssiaaw**
evaddumiaaw	0	0	0	1	0	0	rD	rA	///	1	0	0	1	1	0	0	1	0	0	0	EVX	**evaddumiaaw**
evaddusiaaw	0	0	0	1	0	0	rD	rA	///	1	0	0	1	1	0	0	0	0	0	0	EVX	**evaddusiaaw**
evaddw	0	0	0	1	0	0	rD	rA	rB	0	1	0	0	0	0	0	0	0	0	0	EVX	**evaddw**
evand	0	0	0	1	0	0	rD	rA	rB	0	1	0	0	0	0	1	0	0	0	1	EVX	**evand**
evandc	0	0	0	1	0	0	rD	rA	rB	0	1	0	0	0	0	1	0	0	1	0	EVX	**evandc**
evcmpeq	0	0	0	1	0	0	crfD / /	rA	rB	0	1	0	0	0	1	1	0	1	0	0	EVX	**evcmpeq**
evcmpgts	0	0	0	1	0	0	crfD / /	rA	rB	0	1	0	0	0	1	1	0	0	0	1	EVX	**evcmpgts**
evcmpgtu	0	0	0	1	0	0	crfD / /	rA	rB	0	1	0	0	0	1	1	0	0	0	0	EVX	**evcmpgtu**
evcmplts	0	0	0	1	0	0	crfD / /	rA	rB	0	1	0	0	0	1	1	0	0	1	1	EVX	**evcmplts**
evcmpltu	0	0	0	1	0	0	crfD / /	rA	rB	0	1	0	0	0	1	1	0	0	1	0	EVX	**evcmpltu**
evcntlsw	0	0	0	1	0	0	rD	rA	///	0	1	0	0	0	0	0	1	1	1	0	EVX	**evcntlsw**
evcntlzw	0	0	0	1	0	0	rD	rA	///	0	1	0	0	0	0	0	1	1	0	1	EVX	**evcntlzw**
evdivws	0	0	0	1	0	0	rD	rA	rB	1	0	0	1	1	0	0	0	1	1	0	EVX	**evdivws**
evdivwu	0	0	0	1	0	0	rD	rA	rB	1	0	0	1	1	0	0	0	1	1	1	EVX	**evdivwu**

Table D-3. Instructions (Binary) by Form

Mnemonic	0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21 22 23 24 25 26 27 28 29 30 31	Form	Mnemonic
eveqv	0 0 0 1 0 0	rD	rA	rB	0 1 0 0 0 0 1 1 0 0 1	EVX	**eveqv**
evextsb	0 0 0 1 0 0	rD	rA	///	0 1 0 0 0 0 0 1 0 1 0	EVX	**evextsb**
evextsh	0 0 0 1 0 0	rD	rA	///	0 1 0 0 0 0 0 1 0 1 1	EVX	**evextsh**
evfsabs	0 0 0 1 0 0	rD	rA	///	0 1 0 1 0 0 0 0 1 0 0	EVX	**evfsabs**
evfsadd	0 0 0 1 0 0	rD	rA	rB	0 1 0 1 0 0 0 0 0 0 0	EVX	**evfsadd**
evfscfsf	0 0 0 1 0 0	rD	///	rB	0 1 0 1 0 0 1 0 0 1 1	EVX	**evfscfsf**
evfscfsi	0 0 0 1 0 0	rD	///	rB	0 1 0 1 0 0 1 0 0 0 1	EVX	**evfscfsi**
evfscfuf	0 0 0 1 0 0	rD	///	rB	0 1 0 1 0 0 1 0 0 1 0	EVX	**evfscfuf**
evfscfui	0 0 0 1 0 0	rD	///	rB	0 1 0 1 0 0 1 0 0 0 0	EVX	**evfscfui**
evfscmpeq	0 0 0 1 0 0	crfD / /	rA	rB	0 1 0 1 0 0 0 1 1 1 0	EVX	**evfscmpeq**
evfscmpgt	0 0 0 1 0 0	crfD / /	rA	rB	0 1 0 1 0 0 0 1 1 0 0	EVX	**evfscmpgt**
evfscmplt	0 0 0 1 0 0	crfD / /	rA	rB	0 1 0 1 0 0 0 1 1 0 1	EVX	**evfscmplt**
evfsctsf	0 0 0 1 0 0	rD	///	rB	0 1 0 1 0 0 1 0 1 1 1	EVX	**evfsctsf**
evfsctsi	0 0 0 1 0 0	rD	///	rB	0 1 0 1 0 0 1 0 1 0 1	EVX	**evfsctsi**
evfsctsiz	0 0 0 1 0 0	rD	///	rB	0 1 0 1 0 0 1 1 0 1 0	EVX	**evfsctsiz**
evfsctuf	0 0 0 1 0 0	rD	///	rB	0 1 0 1 0 0 1 0 1 1 0	EVX	**evfsctuf**
evfsctui	0 0 0 1 0 0	rD	///	rB	0 1 0 1 0 0 1 0 1 0 0	EVX	**evfsctui**
evfsctuiz	0 0 0 1 0 0	rD	///	rB	0 1 0 1 0 0 1 1 0 0 0	EVX	**evfsctuiz**
evfsdiv	0 0 0 1 0 0	rD	rA	rB	0 1 0 1 0 0 0 1 0 0 1	EVX	**evfsdiv**
evfsmul	0 0 0 1 0 0	rD	rA	rB	0 1 0 1 0 0 0 1 0 0 0	EVX	**evfsmul**
evfsnabs	0 0 0 1 0 0	rD	rA	///	0 1 0 1 0 0 0 0 1 0 1	EVX	**evfsnabs**
evfsneg	0 0 0 1 0 0	rD	rA	///	0 1 0 1 0 0 0 0 1 1 0	EVX	**evfsneg**
evfssub	0 0 0 1 0 0	rD	rA	rB	0 1 0 1 0 0 0 0 0 0 1	EVX	**evfssub**
evfststeq	0 0 0 1 0 0	crfD / /	rA	rB	0 1 0 1 0 0 1 1 1 1 0	EVX	**evfststeq**
evfststgt	0 0 0 1 0 0	crfD / /	rA	rB	0 1 0 1 0 0 1 1 1 0 0	EVX	**evfststgt**
evfststlt	0 0 0 1 0 0	crfD / /	rA	rB	0 1 0 1 0 0 1 1 1 0 1	EVX	**evfststlt**
evldd	0 0 0 1 0 0	rD	rA	UIMM[2]	0 1 1 0 0 0 0 0 0 0 1	EVX	**evldd**
evlddx	0 0 0 1 0 0	rD	rA	rB	0 1 1 0 0 0 0 0 0 0 0	EVX	**evlddx**
evldh	0 0 0 1 0 0	rD	rA	UIMM [2]	0 1 1 0 0 0 0 0 1 0 1	EVX	**evldh**
evldhx	0 0 0 1 0 0	rD	rA	rB	0 1 1 0 0 0 0 0 1 0 0	EVX	**evldhx**
evldw	0 0 0 1 0 0	rD	rA	UIMM [2]	0 1 1 0 0 0 0 0 0 1 1	EVX	**evldw**
evldwx	0 0 0 1 0 0	rD	rA	rB	0 1 1 0 0 0 0 0 0 1 0	EVX	**evldwx**
evlhhesplat	0 0 0 1 0 0	rD	rA	UIMM[3]	0 1 1 0 0 0 0 1 0 0 1	EVX	**evlhhesplat**

Table D-3. Instructions (Binary) by Form

Mnemonic	0	1	2	3	4	5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evlhhesplatx	0	0	0	1	0	0	rD	rA	rB	0	1	1	0	0	0	0	1	0	0	0	EVX	**evlhhesplatx**
evlhhossplat	0	0	0	1	0	0	rD	rA	UIMM [3]	0	1	1	0	0	0	0	1	1	1	1	EVX	**evlhhossplat**
evlhhossplatx	0	0	0	1	0	0	rD	rA	rB	0	1	1	0	0	0	0	1	1	1	0	EVX	**evlhhossplatx**
evlhhousplat	0	0	0	1	0	0	rD	rA	UIMM [3]	0	1	1	0	0	0	0	1	1	0	1	EVX	**evlhhousplat**
evlhhousplatx	0	0	0	1	0	0	rD	rA	rB	0	1	1	0	0	0	0	1	1	0	0	EVX	**evlhhousplatx**
evlwhe	0	0	0	1	0	0	rD	rA	UIMM [4]	0	1	1	0	0	0	1	0	0	0	1	EVX	**evlwhe**
evlwhex	0	0	0	1	0	0	rD	rA	rB	0	1	1	0	0	0	1	0	0	0	0	EVX	**evlwhex**
evlwhos	0	0	0	1	0	0	rD	rA	UIMM [4]	0	1	1	0	0	0	1	0	1	1	1	EVX	**evlwhos**
evlwhosx	0	0	0	1	0	0	rD	rA	rB	0	1	1	0	0	0	1	0	1	1	0	EVX	**evlwhosx**
evlwhou	0	0	0	1	0	0	rD	rA	UIMM [4]	0	1	1	0	0	0	1	0	1	0	1	EVX	**evlwhou**
evlwhoux	0	0	0	1	0	0	rD	rA	rB	0	1	1	0	0	0	1	0	1	0	0	EVX	**evlwhoux**
evlwhsplat	0	0	0	1	0	0	rD	rA	UIMM [4]	0	1	1	0	0	0	1	1	1	0	1	EVX	**evlwhsplat**
evlwhsplatx	0	0	0	1	0	0	rD	rA	rB	0	1	1	0	0	0	1	1	1	0	0	EVX	**evlwhsplatx**
evlwwsplat	0	0	0	1	0	0	rD	rA	UIMM [4]	0	1	1	0	0	0	1	1	0	0	1	EVX	**evlwwsplat**
evlwwsplatx	0	0	0	1	0	0	rD	rA	rB	0	1	1	0	0	0	1	1	0	0	0	EVX	**evlwwsplatx**
evmergehi	0	0	0	1	0	0	rD	rA	rB	0	1	0	0	0	1	0	1	1	0	0	EVX	**evmergehi**
evmergehilo	0	0	0	1	0	0	rD	rA	rB	0	1	0	0	0	1	0	1	1	1	0	EVX	**evmergehilo**
evmergelo	0	0	0	1	0	0	rD	rA	rB	0	1	0	0	0	1	0	1	1	0	1	EVX	**evmergelo**
evmergelohi	0	0	0	1	0	0	rD	rA	rB	0	1	0	0	0	1	0	1	1	1	1	EVX	**evmergelohi**
evmhegsmfaa	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	0	1	0	1	0	1	1	EVX	**evmhegsmfaa**
evmhegsmfan	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	1	0	1	0	1	1	EVX	**evmhegsmfan**
evmhegsmiaa	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	0	1	0	1	0	0	1	EVX	**evmhegsmiaa**
evmhegsmian	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	1	0	1	0	0	1	EVX	**evmhegsmian**
evmhegumiaa	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	0	1	0	1	0	0	0	EVX	**evmhegumiaa**
evmhegumian	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	1	0	1	0	0	0	EVX	**evmhegumian**
evmhesmf	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	0	0	1	0	1	1	EVX	**evmhesmf**
evmhesmfa	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	1	0	1	0	1	1	EVX	**evmhesmfa**
evmhesmfaaw	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	0	0	0	1	0	1	1	EVX	**evmhesmfaaw**
evmhesmfanw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	0	0	1	0	1	1	EVX	**evmhesmfanw**
evmhesmi	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	0	0	1	0	0	1	EVX	**evmhesmi**
evmhesmia	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	1	0	1	0	0	1	EVX	**evmhesmia**
evmhesmiaaw	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	0	0	0	1	0	0	1	EVX	**evmhesmiaaw**
evmhesmianw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	0	0	1	0	0	1	EVX	**evmhesmianw**

Table D-3. Instructions (Binary) by Form

Mnemonic	0	1	2	3	4	5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evmhessf	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	0	0	0	0	1	1	EVX	**evmhessf**
evmhessfa	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	1	0	0	0	1	1	EVX	**evmhessfa**
evmhessfaaw	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	0	0	0	0	0	1	1	EVX	**evmhessfaaw**
evmhessfanw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	0	0	0	0	1	1	EVX	**evmhessfanw**
evmhessiaaw	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	0	0	0	0	0	0	1	EVX	**evmhessiaaw**
evmhessianw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	0	0	0	0	0	1	EVX	**evmhessianw**
evmheumi	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	0	0	1	0	0	0	EVX	**evmheumi**
evmheumia	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	1	0	1	0	0	0	EVX	**evmheumia**
evmheumiaaw	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	0	0	0	1	0	0	0	EVX	**evmheumiaaw**
evmheumianw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	0	0	1	0	0	0	EVX	**evmheumianw**
evmheusiaaw	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	0	0	0	0	0	0	0	EVX	**evmheusiaaw**
evmheusianw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	0	0	0	0	0	0	EVX	**evmheusianw**
evmhogsmfaa	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	0	1	0	1	1	1	1	EVX	**evmhogsmfaa**
evmhogsmfan	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	1	0	1	1	1	1	EVX	**evmhogsmfan**
evmhogsmiaa	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	0	1	0	1	1	0	1	EVX	**evmhogsmiaa**
evmhogsmian	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	1	0	1	1	0	1	EVX	**evmhogsmian**
evmhogumiaa	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	0	1	0	1	1	0	0	EVX	**evmhogumiaa**
evmhogumian	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	1	0	1	1	0	0	EVX	**evmhogumian**
evmhosmf	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	0	0	1	1	1	1	EVX	**evmhosmf**
evmhosmfa	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	1	0	1	1	1	1	EVX	**evmhosmfa**
evmhosmfaaw	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	0	0	0	1	1	1	1	EVX	**evmhosmfaaw**
evmhosmfanw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	0	0	1	1	1	1	EVX	**evmhosmfanw**
evmhosmi	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	0	0	1	1	0	1	EVX	**evmhosmi**
evmhosmia	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	1	0	1	1	0	1	EVX	**evmhosmia**
evmhosmiaaw	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	0	0	0	1	1	0	1	EVX	**evmhosmiaaw**
evmhosmianw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	0	0	1	1	0	1	EVX	**evmhosmianw**
evmhossf	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	0	0	0	1	1	1	EVX	**evmhossf**
evmhossfa	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	1	0	0	1	1	1	EVX	**evmhossfa**
evmhossfaaw	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	0	0	0	0	1	1	1	EVX	**evmhossfaaw**
evmhossfanw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	0	0	0	1	1	1	EVX	**evmhossfanw**
evmhossiaaw	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	0	0	0	0	1	0	1	EVX	**evmhossiaaw**
evmhossianw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	0	0	0	1	0	1	EVX	**evmhossianw**
evmhoumi	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	0	0	1	1	0	0	EVX	**evmhoumi**

Table D-3. Instructions (Binary) by Form

Mnemonic	0	1	2	3	4	5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evmhoumia	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	0	1	0	1	1	0	0	EVX	**evmhoumia**
evmhoumiaaw	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	0	0	0	1	1	0	0	EVX	**evmhoumiaaw**
evmhoumianw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	0	0	1	1	0	0	EVX	**evmhoumianw**
evmhousiaaw	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	0	0	0	0	1	0	0	EVX	**evmhousiaaw**
evmhousianw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	0	0	0	0	1	0	0	EVX	**evmhousianw**
evmra	0	0	0	1	0	0	rD	rA	///	1	0	0	1	1	0	0	0	1	0	0	EVX	**evmra**
evmwhsmf	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	0	1	1	1	1	EVX	**evmwhsmf**
evmwhsmfa	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	0	1	1	1	1	EVX	**evmwhsmfa**
evmwhsmi	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	0	1	1	0	1	EVX	**evmwhsmi**
evmwhsmia	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	0	1	1	0	1	EVX	**evmwhsmia**
evmwhssf	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	0	0	1	1	1	EVX	**evmwhssf**
evmwhssfa	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	0	0	1	1	1	EVX	**evmwhssfa**
evmwhumi	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	0	1	1	0	0	EVX	**evmwhumi**
evmwhumia	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	0	1	1	0	0	EVX	**evmwhumia**
evmwhusiaaw	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	1	0	0	0	1	0	0	EVX	**evmwhusiaaw**
evmwhusianw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	1	0	0	0	1	0	0	EVX	**evmwhusianw**
evmwlumi	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	0	1	0	0	0	EVX	**evmwlumi**
evmwlumia	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	0	1	0	0	0	EVX	**evmwlumia**
evmwlumiaaw	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	1	0	0	1	0	0	0	EVX	**evmwlumiaaw**
evmwlumianw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	1	0	0	1	0	0	0	EVX	**evmwlumianw**
evmwlusiaaw	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	1	0	0	0	0	0	0	EVX	**evmwlusiaaw**
evmwlusianw	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	1	0	0	0	0	0	0	EVX	**evmwlusianw**
evmwsmf	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	1	1	0	1	1	EVX	**evmwsmf**
evmwsmfa	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	1	1	0	1	1	EVX	**evmwsmfa**
evmwsmfaa	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	1	0	1	1	0	1	1	EVX	**evmwsmfaa**
evmwsmfan	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	1	0	1	1	0	1	1	EVX	**evmwsmfan**
evmwsmi	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	1	1	0	0	1	EVX	**evmwsmi**
evmwsmia	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	1	1	0	0	1	EVX	**evmwsmia**
evmwsmiaa	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	1	0	1	1	0	0	1	EVX	**evmwsmiaa**
evmwsmian	0	0	0	1	0	0	rD	rA	rB	1	0	1	1	1	0	1	1	0	0	1	EVX	**evmwsmian**
evmwssf	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	0	1	0	0	1	1	EVX	**evmwssf**
evmwssfa	0	0	0	1	0	0	rD	rA	rB	1	0	0	0	1	1	1	0	0	1	1	EVX	**evmwssfa**
evmwssfaa	0	0	0	1	0	0	rD	rA	rB	1	0	1	0	1	0	1	0	0	1	1	EVX	**evmwssfaa**

Table D-3. Instructions (Binary) by Form

Mnemonic	0 1 2 3 4 5	6–10	11–15	16–20	21 22 23 24 25 26 27 28 29 30 31	Form	Mnemonic
evmwssfan	0 0 0 1 0 0	rD	rA	rB	1 0 1 1 1 0 1 0 0 1 1	EVX	**evmwssfan**
evmwumi	0 0 0 1 0 0	rD	rA	rB	1 0 0 0 1 0 1 1 0 0 0	EVX	**evmwumi**
evmwumia	0 0 0 1 0 0	rD	rA	rB	1 0 0 0 1 1 1 1 0 0 0	EVX	**evmwumia**
evmwumiaa	0 0 0 1 0 0	rD	rA	rB	1 0 1 0 1 0 1 1 0 0 0	EVX	**evmwumiaa**
evmwumian	0 0 0 1 0 0	rD	rA	rB	1 0 1 1 1 0 1 1 0 0 0	EVX	**evmwumian**
evnand	0 0 0 1 0 0	rD	rA	rB	0 1 0 0 0 0 1 1 1 1 0	EVX	**evnand**
evneg	0 0 0 1 0 0	rD	rA	///	0 1 0 0 0 0 0 1 0 0 1	EVX	**evneg**
evnor	0 0 0 1 0 0	rD	rA	rB	0 1 0 0 0 0 1 1 0 0 0	EVX	**evnor**
evor	0 0 0 1 0 0	rD	rA	rB	0 1 0 0 0 0 1 0 1 1 1	EVX	**evor**
evorc	0 0 0 1 0 0	rD	rA	rB	0 1 0 0 0 0 1 1 0 1 1	EVX	**evorc**
evrlw	0 0 0 1 0 0	rD	rA	rB	0 1 0 0 0 1 0 1 0 0 0	EVX	**evrlw**
evrlwi	0 0 0 1 0 0	rD	rA	UIMM	0 1 0 0 0 1 0 1 0 1 0	EVX	**evrlwi**
evrndw	0 0 0 1 0 0	rD	rA	UIMM	0 1 0 0 0 0 0 1 1 0 0	EVX	**evrndw**
evsel	0 0 0 1 0 0	rD	rA	rB	0 1 0 0 1 1 1 1 crfS	EVX	**evsel**
evslw	0 0 0 1 0 0	rD	rA	rB	0 1 0 0 0 1 0 0 1 0 0	EVX	**evslw**
evslwi	0 0 0 1 0 0	rD	rA	UIMM	0 1 0 0 0 1 0 0 1 1 0	EVX	**evslwi**
evsplatfi	0 0 0 1 0 0	rD	SIMM	///	0 1 0 0 0 1 0 1 0 1 1	EVX	**evsplatfi**
evsplati	0 0 0 1 0 0	rD	SIMM	///	0 1 0 0 0 1 0 1 0 0 1	EVX	**evsplati**
evsrwis	0 0 0 1 0 0	rD	rA	UIMM	0 1 0 0 0 1 0 0 0 1 1	EVX	**evsrwis**
evsrwiu	0 0 0 1 0 0	rD	rA	UIMM	0 1 0 0 0 1 0 0 0 1 0	EVX	**evsrwiu**
evsrws	0 0 0 1 0 0	rD	rA	rB	0 1 0 0 0 1 0 0 0 0 1	EVX	**evsrws**
evsrwu	0 0 0 1 0 0	rD	rA	rB	0 1 0 0 0 1 0 0 0 0 0	EVX	**evsrwu**
evstdd	0 0 0 1 0 0	rD	rA	UIMM [2]	0 1 1 0 0 1 0 0 0 0 1	EVX	**evstdd**
evstddx	0 0 0 1 0 0	rS	rA	rB	0 1 1 0 0 1 0 0 0 0 0	EVX	**evstddx**
evstdh	0 0 0 1 0 0	rS	rA	UIMM [2]	0 1 1 0 0 1 0 0 1 0 1	EVX	**evstdh**
evstdhx	0 0 0 1 0 0	rS	rA	rB	0 1 1 0 0 1 0 0 1 0 0	EVX	**evstdhx**
evstdw	0 0 0 1 0 0	rS	rA	UIMM [2]	0 1 1 0 0 1 0 0 0 1 1	EVX	**evstdw**
evstdwx	0 0 0 1 0 0	rS	rA	rB	0 1 1 0 0 1 0 0 0 1 0	EVX	**evstdwx**
evstwhe	0 0 0 1 0 0	rS	rA	UIMM [4]	0 1 1 0 0 1 1 0 0 0 1	EVX	**evstwhe**
evstwhex	0 0 0 1 0 0	rS	rA	rB	0 1 1 0 0 1 1 0 0 0 0	EVX	**evstwhex**
evstwho	0 0 0 1 0 0	rS	rA	UIMM [4]	0 1 1 0 0 1 1 0 1 0 1	EVX	**evstwho**
evstwhox	0 0 0 1 0 0	rS	rA	rB	0 1 1 0 0 1 1 0 1 0 0	EVX	**evstwhox**
evstwwe	0 0 0 1 0 0	rS	rA	UIMM [4]	0 1 1 0 0 1 1 1 0 0 1	EVX	**evstwwe**

Table D-3. Instructions (Binary) by Form

Mnemonic	0 1 2 3 4 5	6–10	11–15	16–20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
evstwwex	0 0 0 1 0 0	rS	rA	rB	0	1	1	0	0	1	1	1	0	0	0	EVX	**evstwwex**
evstwwo	0 0 0 1 0 0	rS	rA	UIMM [4]	0	1	1	0	0	1	1	1	1	0	1	EVX	**evstwwo**
evstwwox	0 0 0 1 0 0	rS	rA	rB	0	1	1	0	0	1	1	1	1	0	0	EVX	**evstwwox**
evsubfsmiaaw	0 0 0 1 0 0	rD	rA	///	1	0	0	1	1	0	0	1	0	1	1	EVX	**evsubfsmiaaw**
evsubfssiaaw	0 0 0 1 0 0	rD	rA	///	1	0	0	1	1	0	0	0	0	1	1	EVX	**evsubfssiaaw**
evsubfumiaaw	0 0 0 1 0 0	rD	rA	///	1	0	0	1	1	0	0	1	0	1	0	EVX	**evsubfumiaaw**
evsubfusiaaw	0 0 0 1 0 0	rD	rA	///	1	0	0	1	1	0	0	0	0	1	0	EVX	**evsubfusiaaw**
evsubfw	0 0 0 1 0 0	rD	rA	rB	0	1	0	0	0	0	0	0	1	0	0	EVX	**evsubfw**
evsubifw	0 0 0 1 0 0	rD	UIMM	rB	0	1	0	0	0	0	0	0	1	1	0	EVX	**evsubifw**
evxor	0 0 0 1 0 0	rD	rA	rB	0	1	0	0	0	0	1	0	1	1	0	EVX	**evxor**
b	0 1 0 0 1 0	LI												0	0	I	**b**
ba	0 1 0 0 1 0	LI												1	0	I	**ba**
bl	0 1 0 0 1 0	LI												0	1	I	**bl**
bla	0 1 0 0 1 0	LI												1	1	I	**bla**
rlwimi	0 1 0 1 0 0	rS	rA	SH	MB					ME					Rc	M	**rlwimi**
rlwimi.	0 1 0 1 0 0	rS	rA	SH	MB					ME					Rc	M	**rlwimi.**
rlwinm	0 1 0 1 0 1	rS	rA	SH	MB					ME					0	M	**rlwinm**
rlwinm.	0 1 0 1 0 1	rS	rA	SH	MB					ME					1	M	**rlwinm.**
rlwnm	0 1 0 1 1 1	rS	rA	rB	MB					ME					Rc	M	**rlwnm**
rlwnm.	0 1 0 1 1 1	rS	rA	rB	MB					ME					Rc	M	**rlwnm.**
sc	0 1 0 0 0 1	///												1	/	SC	**sc**
mfpmr	0 1 1 1 1 1	rD	PMRN5–9	PMRN0–4	0	1	0	1	0	0	1	1	1	0	0	XFX	**mfpmr**
mfspr	0 1 1 1 1 1	rD	SPRN5–9	SPRN0–4	0	1	0	1	0	1	0	0	1	1	/	XFX	**mfspr**
mtcrf	0 1 1 1 1 1	rS	/ CRM /		0	0	1	0	0	1	0	0	0	0	/	XFX	**mtcrf**
mtpmr	0 1 1 1 1 1	rS	PMRN5–9	PMRN0–4	0	1	1	1	0	0	1	1	1	0	0	XFX	**mtpmr**
mtspr	0 1 1 1 1 1	rS	SPRN5–9	SPRN0–4	0	1	1	1	0	1	0	0	1	1	/	XFX	**mtspr**
bcctr	0 1 0 0 1 1	BO	BI	///	1	0	0	0	0	1	0	0	0	0	0	XL	**bcctr**
bcctrl	0 1 0 0 1 1	BO	BI	///	1	0	0	0	0	1	0	0	0	0	1	XL	**bcctrl**
bclr	0 1 0 0 1 1	BO	BI	///	0	0	0	0	0	1	0	0	0	0	0	XL	**bclr**
bclrl	0 1 0 0 1 1	BO	BI	///	0	0	0	0	0	1	0	0	0	0	1	XL	**bclrl**
crand	0 1 0 0 1 1	crbD	crbA	crbB	0	1	0	0	0	0	0	0	0	1	/	XL	**crand**
crandc	0 1 0 0 1 1	crbD	crbA	crbB	0	0	1	0	0	0	0	0	0	1	/	XL	**crandc**
creqv	0 1 0 0 1 1	crbD	crbA	crbB	0	1	0	0	1	0	0	0	0	1	/	XL	**creqv**

Table D-3. Instructions (Binary) by Form

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
crnand	0	1	0	0	1	1	crbD					crbA					crbB					0	0	1	1	1	0	0	0	0	1	/	XL	**crnand**
crnor	0	1	0	0	1	1	crbD					crbA					crbB					0	0	0	0	1	0	0	0	0	1	/	XL	**crnor**
cror	0	1	0	0	1	1	crbD					crbA					crbB					0	1	1	1	0	0	0	0	0	1	/	XL	**cror**
crorc	0	1	0	0	1	1	crbD					crbA					crbB					0	1	1	0	1	0	0	0	0	1	/	XL	**crorc**
crxor	0	1	0	0	1	1	crbD					crbA					crbB					0	0	1	1	0	0	0	0	0	1	/	XL	**crxor**
isync	0	1	0	0	1	1	///															0	0	1	0	0	1	0	1	1	0	/	XL	**isync**
mcrf	0	1	0	0	1	1	crfD			//		crfS			///							0	0	0	0	0	0	0	0	0	0	/	XL	**mcrf**
rfci	0	1	0	0	1	1	///															0	0	0	0	1	1	0	0	1	1	/	XL	**rfci**
rfi	0	1	0	0	1	1	///															0	0	0	0	1	1	0	0	1	0	/	XL	**rfi**
rfmci	0	1	0	0	1	1	///															0	0	0	0	1	0	0	1	1	0	/	XL	**rfmci**

1 This field is defined as allocated by the Book E architecture, for possible use in an implementation. These bits are not implemented in the e500.

2 d = UIMM * 8

3 d = UIMM * 2

4 d = UIMM * 4

Appendix E
Revision History

This appendix provides a list of major differences between revisions of the *PowerPC e500 Core Reference Manual*.

NOTE

While previous revisions of this manual covered only the e500v1 core, referring to it simply as the e500 core, this revision includes coverage of both the e500v1 and e500v2 cores. As a result, substantial portions of the manual were altered.

E.1 Major Changes From Revision 0 to Revision 1

Table E-1. Revision History

Chapter or Section	Description
Throughout	Revised manual to include coverage of e500v2 core. See Section 1.3.1, "e500v2 Differences," for a list of key differences between the e500v1 and e500v2 cores.
	The coverage of Book E and Freescale Book E MMU architecture (formerly in Chapter 13, Cache and MMU Background) was removed. See the EREF: A reference for Freescale Book E and the e500 Core for more information on this subject.
Section 1.9.1, "Address Translation"	Replaced Figure 1-9 to reflect corrections to address translation bit compositions made in MMU chapter. Added Figure 1-10 for the e500v2 core.
Chapter 2, "Register Model"	Deleted MCSR bits 48–54. Also removed "Recoverable" column of bit descriptions
	Removed SHAREN/SHAREND references in MAS2 and MAS4.
Section 2.10.2, "Hardware Implementation-Dependent Register 1 (HID1)"	Modified description of HID1[RFXE]
Section 2.12.2, "MMU Control and Status Register 0 (MMUCSR0)	Deleted bits 59–60. They are now reserved.
Section 2.12.5, "MMU Assist Registers (MAS0–MAS4, MAS6–MAS7)"	Modified MAS register descriptions to correspond to those of MMU chapter

Table E-1. Revision History

Chapter or Section	Description
Chapter 3, “Instruction Model”	Scalar and vector embedded floating-point instructions are now considered to be in separate APUs from the SPE APU.
	Added material on double-precision floating-point APU
	Notes have been added discouraging use of SPE and embedded floating-point instructions in PowerQUICC III applications.
Section 3.2.3.1, “Synchronization Requirements for e500-Specific SPRs”	In Table 3-4, all of the **mtspr** to debug register (IAC, DAC, DBCR0, DBCR1, DBSR) instructions must be followed by a context-synchronizing instruction, but no synchronization is required before them. Previously, no post-synchronization was shown.
Section 3.3.1.6.1, “mbar (MO = 1)”	Added section to provide an EIS architectural definition for **mbar** (MO = 1), which is the classic PowerPC architecture definition of **eieio.**
Section 3.3.1.8.1, “User-Level Cache Instructions”	In Table 3-26, the **dcbz** instruction does not take an alignment interrupt if the cache is disabled.
Section 3.5, “Using msync and mbar to Order Memory Accesses”	Added section
Section 3.10, “Instruction Listing”	Book E 64-bit instructions were removed from Table 3-44.
Section 5.3, “Interrupt Registers	Deleted MCSR[GL_CI] from Table 5-4. Also removed column “Recoverable” in same table
Section 5.7, “Interrupt Definitions”	Deleted references to ESR[AP], which is not implemented on the e500.
Chapter 9, “Timer Facilities”	Corrected concatenation order of WPEXT || WP and (FPEXT || FP) TCR[WPEXT] and TCR[FPEXT], not specified in Book E, are concatenated with TCR[WP] and TCR[FP]
Chapter 10, “Auxiliary Processing Units (APUs)”	Removed coverage of Freescale Book E–defined APUs. See the EREF for more information.
Chapter 11, “L1 Caches”	Removed references to MEI.
	Deleted the ‘D’ from the acronyms for the L1CSR0 bit fields
Section 11.2.2, “L1 Instruction Cache Organization”	Added note: On the e500v1, it is possible for multiple entries in the L1 instruction cache to contain data for the same physical memory location. This error can occur when two different effective addresses (EA) map to the same physical address and accesses to these two EAs occur within the same context and relatively close together in time. This is avoided by not fetching instructions from one physical address through two or more different EAs within any given context.
Section 11.2.3, “L1 Cache Parity”	Added section. .
Section 11.2.4, “Cache Parity Error Injection”	Changed name of L1CSR0[PEIE] to CPI and L1CSR1[IPEIE] to ICPI. Added requirement to have cache parity checking enabled if cache parity injection is enabled.

Table E-1. Revision History

Chapter or Section	Description
Section 11.3.4.5, "Speculative Accesses to Guarded Memory"	Added caution about cacheable and guarded loads for e500v1
Section 11.3.5.2, "Sequential Consistency of Memory Accesses"	Replaced "Newer caching-allowed loads can bypass older caching-allowed loads only if the two loads are to different 32-byte address granules" with "Newer non-guarded, caching-allowed loads can bypass older non-guarded, caching-allowed loads."
Chapter 12, "Memory Management Units"	Removed references to SHAREN, SHAREND, MEI
Section 12.2, "Effective-to-Real Address Translation"	Corrected bit number compositions in effective-to-real address translation figures, Figure 12-1 and Figure 12-2
Chapter 13, "Core Complex Bus (CCB)"	Added chapter

Index

Numerics

Numerics

A

B

C

F

G

H

I

J

L

M

N

O

P

R

S

T

U

W

X