e200z760n3 Power Architecture® Core Reference Manual

Supports
e200z760n3

e200z760RM
Rev. 2
06/2012

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or
+1-480-768-2130
www.freescale.com/support

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)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku
Tokyo 153-0064
Japan
0120 191014 or
+81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 010 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor
Literature Distribution Center
+1-800 441-2447 or
+1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor
@hibbertgroup.com

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters which may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals” must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Freescale, the Freescale logo,and PowerQUICCare trademarks of Freescale Semiconductor, Inc. Reg. U.S. Pat. & Tm. Off. All other product or service names are the property of their respective owners.
© 2010-2012 Freescale Semiconductor, Inc.

Document Number: e200z760RM
Rev. 2, 06/2012

Contents

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

About This Book

Chapter 1
e200z7 Core Complex Overview

Chapter 2
Register Model

Contents

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

Chapter 3
Instruction Model

Contents

Paragraph Number | Title | Page Number

Chapter 4
Instruction Pipeline and Execution Timing

Contents

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

Chapter 5
Embedded Floating-Point Unit

Chapter 6
Signal Processing Extension (SPE)

Contents

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

Contents

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

Chapter 7
Interrupts and Exceptions

Contents

Chapter 8
Performance Monitor

Chapter 9
L1 Cache

Contents

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

Contents

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

Contents

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

Chapter 10
Memory Management Unit

Chapter 11
External Core Complex Interfaces

Contents

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

Contents

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

Contents

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

Contents

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

Chapter 12
Power Management

Chapter 13
Debug Support

Contents

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

Chapter 14
Nexus 3 Module

Contents

Paragraph Number	Title	Page Number

Contents

Paragraph Number	Title	Page Number

Contents

Paragraph Number	Title	Page Number

Appendix A
Register Summary

Appendix B
Revision History

Tables

Table Number | **Title** | **Page Number**

Tables

Tables

Tables

Table Number **Title** **Page Number**

Tables

Tables

Tables

Figures

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

Figures

Figures

Figures

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

Figures

Figures

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

Figures

Figures

About This Book

The primary objective of this manual is to describe the functionality of the e200z760 embedded microprocessor core for software and hardware developers. This book is intended as a companion to the *EREF: A Programmer's Reference Manual for Freescale Embedded Processors* (hereafter referred to as the *EREF*).

Users of prior implementations of the e200 core family, such as the e200z6, may notice new terminology employed throughout this manual. In 2004, most of Freescale’s Embedded Implementation Standards (EIS) were contributed to help launch Power.org whose mission was to develop, enable, and promote technology originally conceived as the PowerPC architecture. References to “PowerPC” are replaced with “Power ISA (Instruction Set Architecture) embedded category.” The term “Auxiliary Processing Unit (APU)” is used to describe a collection of functionality within the EIS. These APUs were either absorbed into various parts of the new Power ISA or retained their identity and became known as individual, and sometimes optional, “categories” or “subcategories” of the Power ISA.

This document includes three levels of architectural and implementation definition, as follows:

- Power ISA embedded category—defines a set of user-level instructions and registers that are a part of the Power ISA.
- e200 implementation details—In some cases, the Power ISA definition provides a general framework, leaving specific details up to the implementation. Some of these details are common to all members of the e200 core family and may be indicated as such.
- e200z7 implementation details—The next level of architectural specificity describes those features that are shared across the cores in the e200z7 sub-family but that may be in the other members of the e200 product line.
- e200z760n3 implementation details—The e200z7 subfamily includes one or more specific cores with unique combinations of functionality. Each processor core in the e200z7 product line defines instructions, registers, register fields, and other aspects that are more detailed than the architectural layers described above. When features are implemented differently between the varieties of e200z7 cores, they are specifically noted as such.

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 parts of this reference manual:

- Chapter 1, "e200z7 Core Complex Overview," provides a general description of e200z760 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, "Instruction Pipeline and Execution Timing," describes how instructions are fetched, decoded, issued, 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 e200z7.
- Chapter 5, "Embedded Floating-Point Unit," describes the instruction set architecture of the Embedded Floating-point (EFPU) implemented on the e200z7. This unit implements scalar and vector single-precision floating-point instructions to accelerate signal processing and other algorithms. The e200z760n3 implements version 2 of the embedded floating-point unit (EFPU2).
- Chapter 6, "Signal Processing Extension (SPE) describes the instruction set architecture of the SPE and implements instructions to accelerate signal processing and other algorithms.
- Chapter 7, "Interrupts and Exceptions," describes how the e200z7 implements the interrupt model as it is defined by the Book E architecture.
- Chapter 9, "L1 Cache ," describes the organization of the on-chip L1 Caches, cache control instructions, and various cache operations.
- Chapter 10, "Memory Management Unit," provides specific hardware and software details regarding the e200z7 MMU implementation.
- Chapter 11, "External Core Complex Interfaces," describes those aspects of the CCB that are configurable or that provide status information through the programming interface. It provides a glossary of signals mentioned throughout the book to offer a clearer understanding of how the core is integrated as part of a larger device.
- Chapter 12, "Power Management," describes the power management facilities as they are defined and implemented in the e200z7 core.
- Chapter 13, "Debug Support," describes the internal debug facilities as they are implemented in the e200z760 core.
- Chapter 14, "Nexus 3 Module," describes the Nexus3 module, which provides real-time development capabilities for e200z760 processors in compliance with the *IEEE-ISTO Nexus 5001-2008* standard.
- Appendix A, "Register Summary," compiles the register figures for this manual.
- Appendix B, "Revision History," contains a revision history for this manual.

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 provides useful information about Power Architecture® technology and computer architecture in general:

- *Power ISA™ Version 2.05*, by Power.org™, 2007, available at the Power.org website.
- *PowerPC Architecture Book*, by Brad Frey, IBM, 2005, available at the IBM website.
- *Computer Architecture: A Quantitative Approach*, Fourth Edition, by John L. Hennessy and David A. Patterson, Morgan Kaufmann Publishers, 2006.
- *Computer Organization and Design: The Hardware/Software Interface*, Third Edition, by David A. Patterson and John L. Hennessy, Morgan Kaufmann Publishers, 2007.

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

- *EREF: A Programmer's Reference Manual for Freescale Embedded Processors* (EREFRM). Describes the programming, memory management, cache, and interrupt models defined by the Power ISA™ for embedded environment processors.
- *Power ISA™*. The latest version of the Power instruction set architecture can be downloaded from the website www.power.org.
- Category-specific programming environments manuals. These books describe the three major extensions to the Power ISA embedded environment of the Power ISA. These include the following:
  - *AltiVec™ Technology Programming Environments Manual* (ALTIVECPEM)
  - *Signal Processing Engine (SPE) Programming Environments Manual: A Supplement to the EREF* (SPEPEM)
  - *Variable-Length Encoding (VLE) Programming Environments Manual: A Supplement to the EREF* (VLEPEM)
- Core reference manuals—These books describe the features and behavior of individual microprocessor cores and provide specific information about how functionality described in the EREF is implemented by a particular core. They also describe implementation-specific features and microarchitectural details, such as instruction timing and cache hardware details, that lie outside the architecture specification.
- Integrated device reference manuals—These manuals describe the features and behavior of integrated devices that implement and utilize a Power ISA processor core.
- Addenda/errata to reference manuals—When processors have follow-on parts, often 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.

- Technical summaries—Each device has a technical summary that provides an overview of its features. This document is roughly the equivalent to the overview (Chapter 1) of an implementation's 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.

Acronyms and Abbreviations

Table i contains acronyms and abbreviations that are used in this document. Note that the meanings for some acronyms (such as XER) are historical, and the words for which an acronym stands may not be intuitively obvious.

Table i. Acronyms and Abbreviated Terms

Term	Meaning
CR	Condition register
CTR	Count register
DCR	Data control register
DTLB	Data translation lookaside buffer
EA	Effective address
ECC	Error checking and correction
FPR	Floating-point register
GPR	General-purpose register
IEEE	Institute of Electrical and Electronics Engineers
LR	Link register
LRU	Least recently used
LSB	Least-significant byte
lsb	Least-significant bit
MMU	Memory management unit
MSB	Most-significant byte
msb	Most-significant bit
MSR	Machine state register
NaN	Not a number
No-op	No operation
PTE	Page table entry
PVR	Processor version register
RISC	Reduced instruction set computing

Table i. Acronyms and Abbreviated Terms (continued)

Term	Meaning
RTL	Register transfer language
SIMM	Signed immediate value
SPR	Special-purpose register
SRR0	Machine status save/restore register 0
SRR1	Machine status save/restore register 1
TB	Time base facility
TBL	Time base lower register
TBU	Time base upper register
TLB	Translation lookaside buffer
UIMM	Unsigned immediate value
UISA	User instruction set architecture
VA	Virtual address
VLE	Variable-length encoding
XER	Register used for indicating conditions such as carries and overflows for integer operations

Terminology Conventions

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

Table ii. Terminology Conventions

The Architecture Specification	This Manual
Extended mnemonics	Simplified mnemonics
Fixed-point unit (FXU)	Integer unit (IU)
Privileged mode (or privileged state)	Supervisor-level privilege
Problem mode (or problem state)	User-level privilege
Real address	Physical address
Relocation	Translation
Storage (locations)	Memory
Storage (the act of)	Access
Store in	Write back
Store through	Write through

Table iii describes instruction field notation conventions used in this manual.

Table iii. Instruction Field Conventions

The Architecture Specification	Equivalent to:
BA, BB, BT	**crb**A, **crb**B, **crb**D (respectively)
BF, BFA	**crf**D, **crf**S (respectively)
D	d
DS	ds
/, //, ///	0...0 (shaded)
RA, RB, RT, RS	**r**A, **r**B, **r**D, **r**S (respectively)
SI	SIMM
U	IMM
UI	UIMM

Chapter 1
e200z7 Core Complex Overview

This chapter provides an overview of the e200z7 microprocessor core built on Power Architecture® technology for embedded processors. It includes the following:

- A summary of the feature set for this core
- An overview of the register set
- An overview of the instruction set
- An overview of interrupts and exception handling
- A summary of instruction pipeline and flow
- A description of the memory management architecture
- High-level details of the core memory and coherency model

1.1 e200z7 Overview

The e200z7 processor core is a low-cost implementation of Power Architecture technology for embedded processors. It is a dual-issue, 32-bit, Power ISA-compliant design with 64-bit, general-purpose registers (GPRs).

In addition to the base Power ISA embedded category instruction set, the e200z7 also implements the variable-length encoding (VLE) category, providing improved code density. See the *EREF* and supplementary *VLE PEM* for more information about the VLE extension.

Instructions of the signal processing extension (SPE) category, as well as of the embedded vector and scalar floating-point categories, are provided to support real-time integer and single-precision embedded floating-point operations using the GPRs. The e200z7 does not support Power ISA floating-point instructions in hardware, but traps them so they can be emulated by software.

All arithmetic instructions that execute in the core operate on data in the GPRs, which have been extended to 64 bits to support vector instructions defined by the SPE and embedded vector floating-point categories. These instructions operate on a vector pair of 16- or 32-bit data types and deliver vector and scalar results.

The e200z7 contains a 16-KB instruction cache, a 16-KB data cache, as well as a memory management unit. A Nexus Class 3+ module is also integrated.

Figure 1-1 shows a high-level block diagram of the e200z7 core.

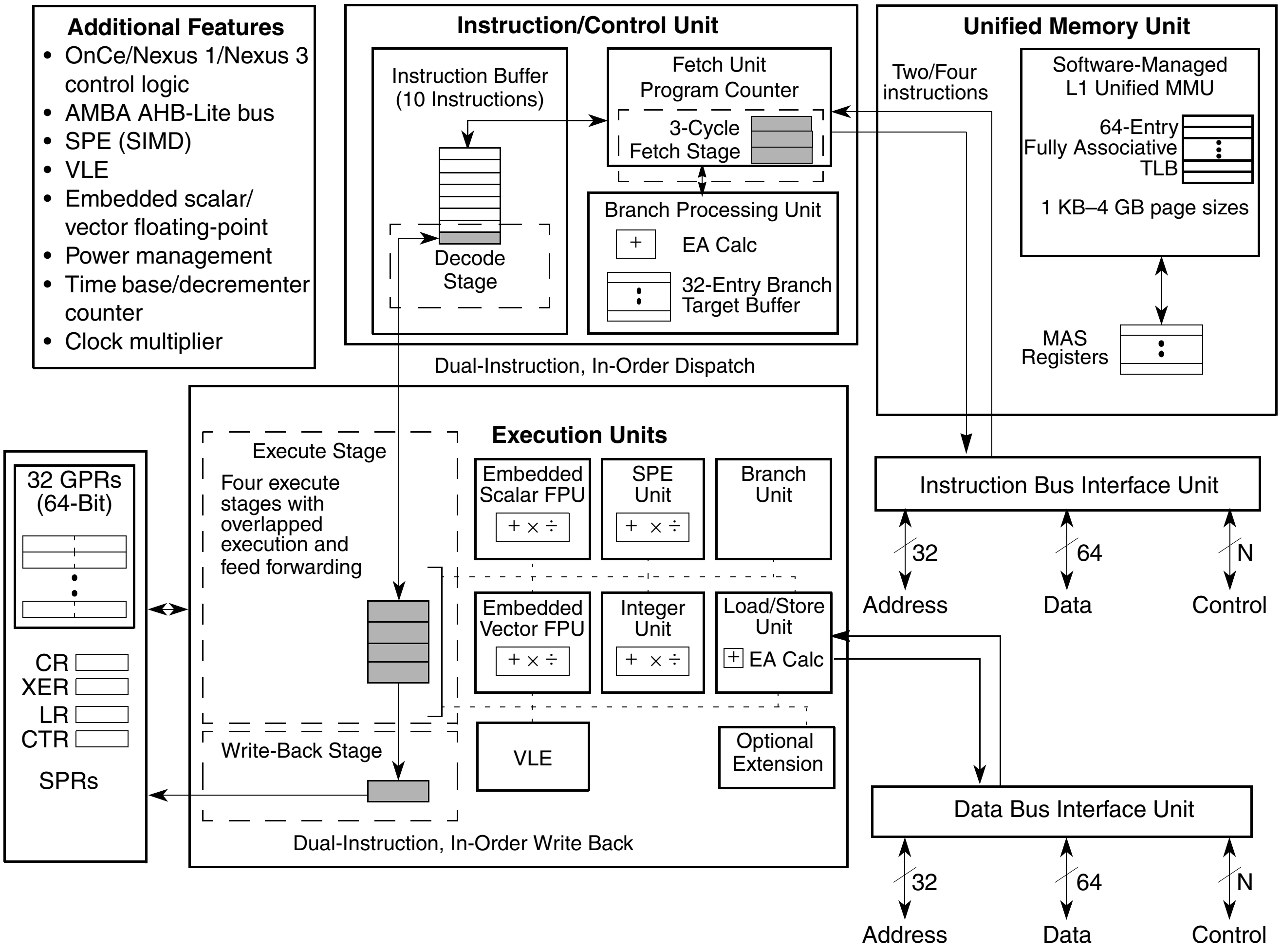


Figure 1-1. e200z7 Block Diagram

1.1.1 Features

Key features of the e200z7 are summarized as follows:

- Dual-issue, 32-bit Power ISA–compliant core
- Implementation of the VLE category for reduced code footprint
- In-order execution and retirement
- Precise exception handling
- Branch processing unit (BPU)
  - — Dedicated branch address calculation adder
  - — Branch target prefetching using a branch target buffer (BTB)
  - — Return address stack
- Load/store unit (LSU)
  - — Three-cycle load latency

  - Fully pipelined
  - Big- and little-endian support
  - Misaligned access support
  - AMBA (advanced microcontroller bus architecture) AHB-Lite (advanced high-performance bus) 64-bit system bus
- Memory management unit (MMU) with 64-entry, fully associative TLB and multiple page-size support
- 16-KB, 4 way set-associative Harvard instruction and data caches
- SPE unit supporting SIMD fixed-point and single-precision floating-point operations, using the 64-bit GPR file.
- Embedded floating-point unit (EFPU) supporting scalar single-precision floating-point operations.
- Performance management unit (PMU) supporting execution profiling
- Nexus Class 3+ real-time development unit
- Power management
  - Low-power design—extensive clock gating
  - Power-saving modes: doze, nap, sleep, and wait
  - Dynamic power management of execution units, caches, and MMUs
- e200z7-specific debug interrupt.
- Testability
  - Synthesizable, full MuxD scan design
  - Built-in parallel signature unit

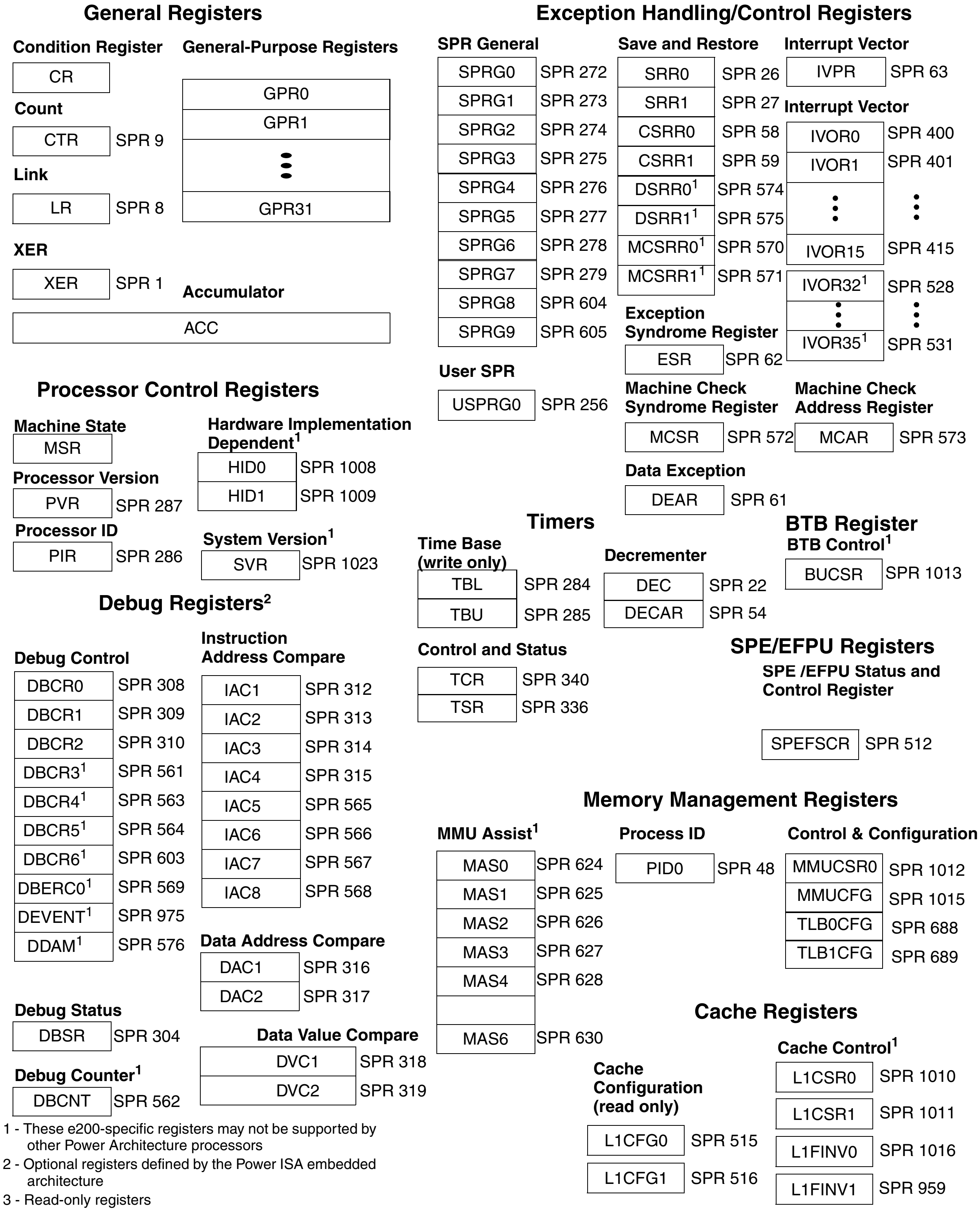
1.2 Programming Model

This section describes the register model, instruction model, and the interrupt model as they are defined by the Power ISA, Freescale EIS, and the e200z7 implementation.

1.2.1 Register Set

Figure 1-2–Figure 1-5 show the complete e200z7 register set divided into supervisor and user-level registers and grouped into general-purpose registers (GPRs), special-purpose registers (SPRs), device control registers (DCRs), and any performance monitor registers (PMRs) that may implemented in a particular variation of the e200z7 core family The number to the right of the special-purpose registers (SPRs) is the decimal number used in the instruction syntax to access the register. For example, the integer exception register (XER) is SPR 1.

Figure 1-2 shows the supervisor mode programmer’s model.



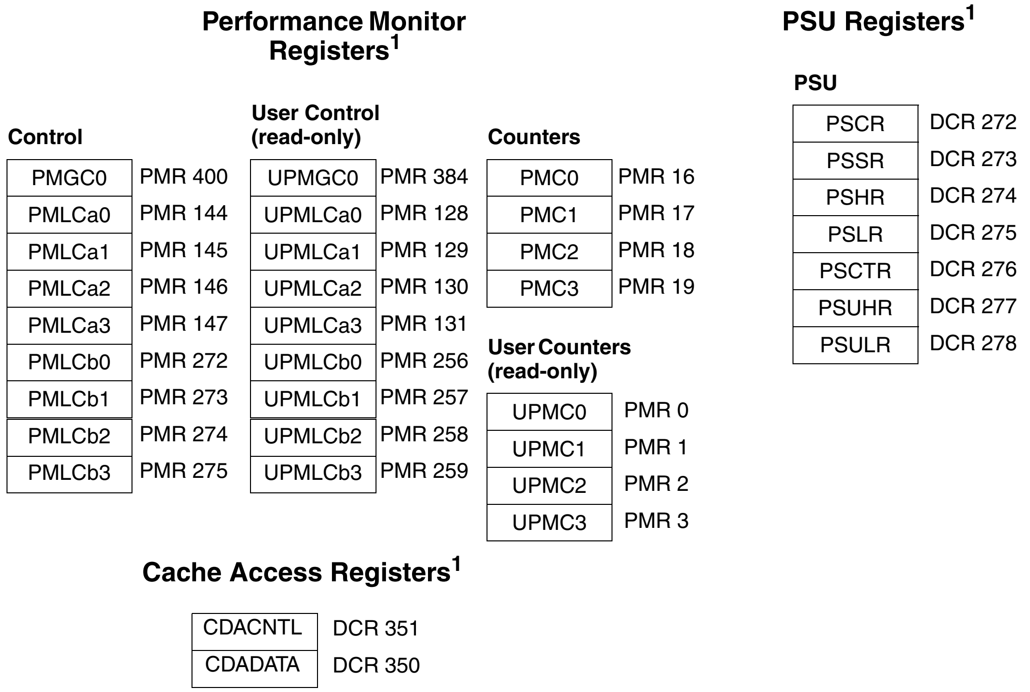
1 - These e200-specific registers may not be supported by other Power Architecture processors

2 - Optional registers defined by the Power ISA embedded architecture

3 - Read-only registers

Figure 1-2. e200z760 Supervisor Mode Programmer’s Model

Figure 1-3 shows the supervisor mode programmer models's DCRs and PMRs.



Note:

1) These e200-specific registers may not be supported by other Power ISA embedded category processors

Figure 1-3. e200z760 Supervisor Mode Programmer's Model DCRs and PMRs

Figure 1-4 shows the user mode programmer's model.

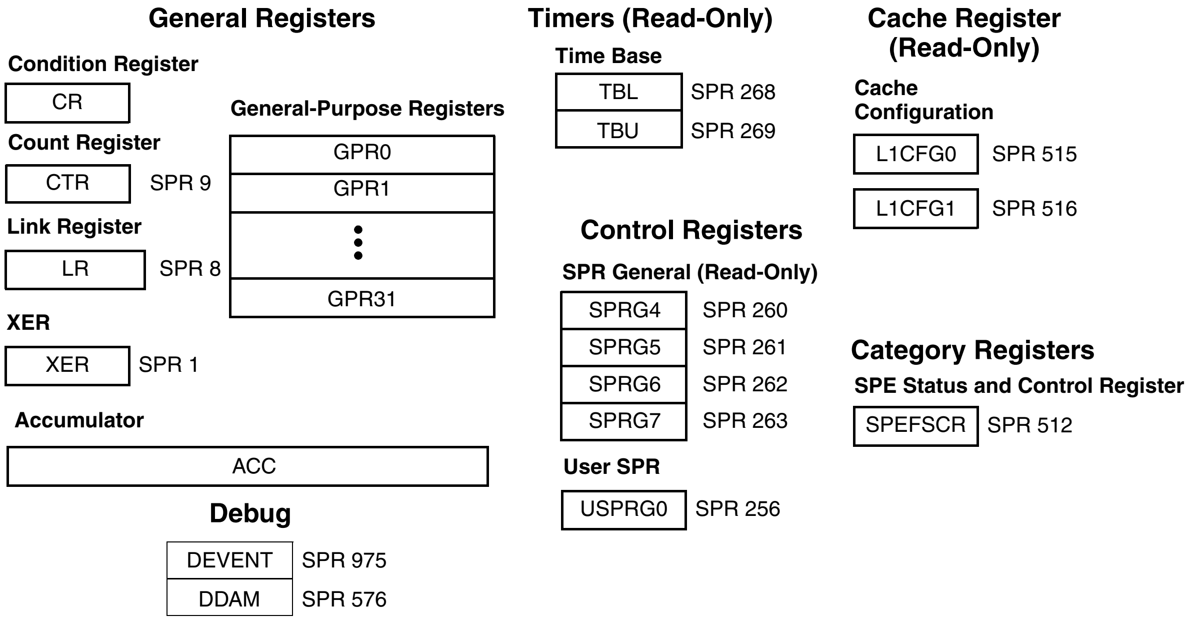
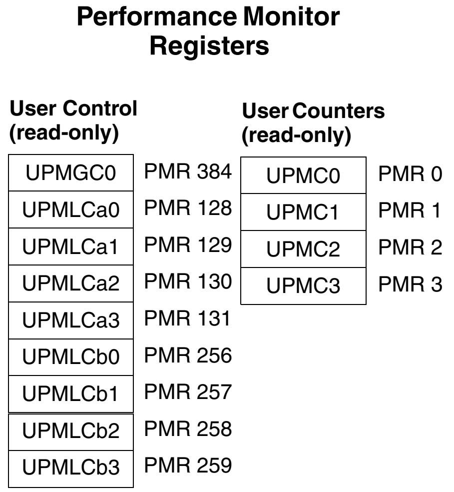


Figure 1-4. e200z7 User Mode Programmer's Model

Figure 1-5 shows the user mode programmer’s model PMRs.



Performance Monitor Registers

User Control (read-only)

Register	PMR
UPMGC0	PMR 384
UPMLCa0	PMR 128
UPMLCa1	PMR 129
UPMLCa2	PMR 130
UPMLCa3	PMR 131
UPMLCb0	PMR 256
UPMLCb1	PMR 257
UPMLCb2	PMR 258
UPMLCb3	PMR 259

User Counters (read-only)

Register	PMR
UPMC0	PMR 0
UPMC1	PMR 1
UPMC2	PMR 2
UPMC3	PMR 3

Note:

These e200-specific registers may not be supported by other Power ISA embedded category processors.

Figure 1-5. e200 User Mode Programmer’s Model PMRs

The GPRs are accessed through instruction operands. Access to other registers can be explicit (by using instructions for that purpose such as the Move To Special Purpose Register (**mtspr**) and Move From Special Purpose Register (**mfspr**) instructions) or implicit as part of the execution of an instruction. Some registers are accessed both explicitly and implicitly.

For more information about the registers, see Chapter 2, “Register Model.”

1.2.2 Instruction Set

The e200z7 supports the following architectural extensions: VLE, ISEL, debug, machine check, wait, SPE, cache line locking, and enhanced reservations.

The e200z7 implements the following instructions:

- The Power ISA instruction set for 32-bit embedded implementations. This is composed primarily of the user-level instructions defined by the user instruction set architecture (UISA). The e200z7 does not include the Power ISA floating-point, load string, or store string instructions.
- The e200z7 supports the following EIS-defined instructions:
  — Integer select category. This category 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, takes fewer clock cycles than the equivalent coding, and reduces the code footprint.

- — Cache line lock and unlock category. The cache block lock and unlock category consists of the instructions described in Table 1-1, which defines a set of instructions for locking and clearing cache lines.

Table 1-1. Cache Block Lock and Unlock Instructions

Name	Mnemonic	Syntax
Data Cache Block Lock Clear	**dcblc**	CT,rA,rB
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

- — Debug category. This category defines the Return from Debug Interrupt instruction (**rfdi**), which defines a separate set of interrupt save and restore registers to provide greater responsiveness for debug interrupts.
- — SPE vector category. New vector instructions are defined that view the 64-bit GPRs as being composed of a vector of two 32-bit elements (some of the instructions also read or write 16-bit elements). Some scalar instructions are defined for DSP that produce a 64-bit scalar result.
- — The embedded floating-point categories provide single-precision scalar and vector floating-point instructions. Scalar floating-point instructions use only the lower 32 bits of the GPRs for single-precision floating-point calculations. Table 1-2 lists embedded floating-point instructions.
- — Wait category. This category consists of the **wait** instruction that allows software to cease all synchronous activity and wait for an asynchronous interrupt to occur.
- — Machine check category. This feature set adds two new instructions (**rfmci**, **se_rfmci**) and four new registers (MCSRRO, MCSRR1, MCSR, MCAR)
- — Volatile Context Save/Restore category supports the capability to quickly save and restore volatile register context on entry into an interrupt handler.

Table 1-2. Scalar and Vector Embedded Floating-Point Instructions

Instruction	Mnemonic		Syntax
	Scalar	Vector	
Convert Floating-Point from Signed Fraction	**efscfsf**	**evfscfsf**	rD,rB
Convert Floating-Point from Signed Integer	**efscfsi**	**evfscfsi**	rD,rB
Convert Floating-Point from Unsigned Fraction	**efscfuf**	**evfscfuf**	rD,rB
Convert Floating-Point from Unsigned Integer	**efscfui**	**evfscfui**	rD,rB
Convert Floating-Point Single-Precision from Half-Precision	**efscfh**	**evfscfh**	rD,rB
Convert Floating-Point Single-Precision to Half-Precision	**efscth**	**evfscth**	rD,rB
Convert Floating-Point to Signed Fraction	**efsctsf**	**evfsctsf**	rD,rB
Convert Floating-Point to Signed Integer	**efsctsi**	**evfsctsi**	rD,rB

Table 1-2. Scalar and Vector Embedded Floating-Point Instructions (continued)

Instruction	Mnemonic		Syntax
	Scalar	Vector	
Convert Floating-Point to Signed Integer with Round Toward Zero	**efsctsiz**	**evfsctsiz**	rD,rB
Convert Floating-Point to Unsigned Fraction	**efsctuf**	**evfsctuf**	rD,rB
Convert Floating-Point to Unsigned Integer	**efsctui**	**evfsctui**	rD,rB
Convert Floating-Point to Unsigned Integer with Round Toward Zero	**efsctuiz**	**evfsctuiz**	rD,rB
Floating-Point Absolute Value	**efsabs**	**evfsabs**	rD,rA
Floating-Point Add	**efsadd**	**evfsadd**	rD,rA,rB
Floating-Point Compare Equal	**efscmpeq**	**evfscmpeq**	crD,rA,rB
Floating-Point Compare Greater Than	**efscmpgt**	**evfscmpgt**	crD,rA,rB
Floating-Point Compare Less Than	**efscmplt**	**evfscmplt**	crD,rA,rB
Floating-Point Divide	**efsdiv**	**evfsdiv**	rD,rA,rB
Floating-Point Multiply	**efsmul**	**evfsmul**	rD,rA,rB
Floating-Point Negate	**efsneg**	**evfsneg**	rD,rA
Floating-Point Negative Absolute Value	**efsnabs**	**evfsnabs**	rD,rA
Floating-Point Subtract	**efssub**	**evfssub**	rD,rA,rB
Floating-Point Test Equal	**efststeq**	**evfststeq**	crD,rA,rB
Floating-Point Test Greater Than	**efststgt**	**evfststgt**	crD,rA,rB
Floating-Point Test Less Than	**efststlt**	**evfststlt**	crD,rA,rB
Floating-Point Single-Precision Maximum	**efsmax**	**evfsmax**	rD,rA,rB
Floating-Point Single-Precision Minimum	**efsmin**	**evfsmin**	rD,rA,rB
Floating-Point Single-Precision Multiply-Add	**efsmadd**	**evfsmadd**	rD,rA,rB
Floating-Point Single-Precision Negative Multiply-Add	**efsnmadd**	**evfsnmadd**	rD,rA,rB
Floating-Point Single-Precision Multiply-Subtract	**efsmsub**	**evfsmsub**	rD,rA,rB
Floating-Point Single-Precision Negative Multiply-Subtract	**efsnmsub**	**evfsnmsub**	rD,rA,rB
Floating-Point Single-Precision Square Root	**efssqrt**	**evfssqrt**	rD,rA
Vector Floating-Point Single-Precision Add / Subtract	—	**evfsaddsub**	rD,rA,rB
Vector Floating-Point Single-Precision Add / Subtract Exchanged	—	**evfsaddsubx**	rD,rA,rB
Vector Floating-Point Single-Precision Add Exchanged	—	**evfsaddx**	rD,rA,rB
Vector Floating-Point Single-Precision Difference / Sum	—	**evfsdiffsum**	rD,rA,rB
Vector Floating-Point Single-Precision Differences	—	**evfsdiff**	rD,rA,rB
Vector Floating-Point Single-Precision Multiply By Even Element	—	**evfsmule**	rD,rA,rB
Vector Floating-Point Single-Precision Multiply By Odd Element	—	**evfsmulo**	rD,rA,rB

Table 1-2. Scalar and Vector Embedded Floating-Point Instructions (continued)

Instruction	Mnemonic		Syntax
	Scalar	Vector	
Vector Floating-Point Single-Precision Multiply Exchanged	—	**evfsmulx**	rD,rA,rB
Vector Floating-Point Single-Precision Subtract / Add Exchanged	—	**evfssubaddx**	rD,rA,rB
Vector Floating-Point Single-Precision Subtract Exchanged	—	**evfssubx**	rD,rA,rB
Vector Floating-Point Single-Precision Subtract/Add	—	**evfssubadd**	rD,rA,rB
Vector Floating-Point Single-Precision Sum / Difference	—	**evfssumdiff**	rD,rA,rB
Vector Floating-Point Single-Precision Sums	—	**evfssum**	rD,rA,rB

For more information about the instruction set, see Chapter 3, “Instruction Model.”

1.2.2.1 VLE Category

This section describes the extensions to the architecture to support VLE.

- **rfci**, **rfdi**, **rfi** do not mask bit 62 of CSRR0, DSRR0, or SRR0. The destination address is [D,C]SRR0[32–62] || 0b0.
- **bclr**, **bclrl**, **bcctr**, **bcctrl** do not mask bit 62 of the LR or CTR. The destination address is [LR, CTR][32–62] || 0b0.

1.2.3 Interrupts and Exception Handling

The core supports an extended exception handling model, with nested interrupt capability and extensive interrupt vector programmability. The following sections define the interrupt model, including an overview of interrupt handling as implemented on the e200z7 core, a brief description of the interrupt classes, and an overview of the registers involved in the processes.

For more information about interrupts and exception handling, see Chapter 7, “Interrupts and Exceptions.”

1.2.3.1 Interrupt Handling

In general, interrupt processing begins with an exception that occurs due to external conditions, errors, or program execution problems. When an exception occurs, the processor checks whether 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 bits in the exception syndrome register (ESR), the machine check syndrome register (MCSR), or the signal processing and embedded floating-point status and control register (SPEFSCR), depending on the exception type, to verify the specific cause of the exception and take appropriate action.

The core complex supports the interrupts described in Section 1.2.3.4, “Interrupt Registers.”

1.2.3.2 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 are further divided into precise and imprecise types.
  - 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 allow determination of which of the two instructions has been 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.

1.2.3.3 Interrupt Types

The e200z7 core processes all interrupts as either debug, machine check, critical, or noncritical types. Separate control and status register sets are provided for each type of interrupt. Table 1-3 describes the interrupt types.

Table 1-3. Interrupt Types

Category	Description	Programming Resources
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.	SRR0/SRR1 SPRs and **rfi** instruction. Asynchronous noncritical interrupts can be masked by the external interrupt enable bit, MSR[EE].
Critical interrupts	Critical input, watchdog timer, and debug interrupts. These interrupts can be taken during a noncritical interrupt or during regular program flow. The critical input and watchdog timer interrupts are treated as critical interrupts. If the debug feature is not enabled, a debug interrupt is treated as a critical interrupt.	Critical save and restore SPRs (CSRR0/CSRR1) and **rfci**. 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].

Table 1-3. Interrupt Types

Category	Description	Programming Resources
Machine check interrupt	Provides a separate set of resources for the machine check interrupt. See Section 7.6.2, “Machine Check Interrupt (IVOR1).”	Machine check save and restore SPRs (MCSRR0/MCSRR1) and **rfmci**. Maskable with the machine check enable bit, MSR[ME]. Includes the machine check syndrome register (MCSR).
Debug interrupt	Provides a separate set of resources for the debug interrupt. See Section 7.6.16, “Debug Interrupt (IVOR15).”	Debug save and restore SPRs (DSRR0/DSRR1) and **rfdi**. Can be masked by the debug interrupt enable bit, MSR[DE]. Includes the debug status register (DBSR).

Because save/restore register pairs are serially reusable, care must be taken to preserve program state that may be lost when an unordered interrupt is taken.

1.2.3.4 Interrupt Registers

The registers associated with interrupt handling are described in Table 1-4.

Table 1-4. Interrupt Registers

Register	Description
Noncritical Interrupt Registers	
SRR0	Save/restore register 0—Stores 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—Saves machine state on noncritical interrupts and restores machine state when an **rfi** instruction is executed.
Critical Interrupt Registers	
CSRR0	Critical save/restore register 0—On critical interrupts, stores either the address of the instruction causing the exception or the address of the instruction that executes after the **rfci**.
CSRR1	Critical save/restore register 1—Saves machine state on critical interrupts and restores machine state when an **rfci** instruction is executed.
Debug Interrupt Registers	
DSRR0	Debug save/restore register 0—Used to store the address of the instruction that will execute when an **rfdi** instruction is executed.
DSRR1	Debug save/restore register 1—Stores machine state on debug interrupts and restores machine state when an **rfdi** instruction is executed.
Machine Check Interrupts	
MCSRR0	Machine check save/restore register 0—On machine check interrupts, stores either the address of the instruction causing the exception or the address of the instruction that executes after the **rfmci** instruction.
MCSRR1	Machine check save/restore register 1—Saves machine state on machine check interrupts and restores those values when an **rfmci** instruction is executed
Syndrome Registers	
MCSR	Machine check syndrome register—Saves machine check syndrome information on machine check interrupts.

Table 1-4. Interrupt Registers (continued)

Register	Description
ESR	Exception syndrome register—Provides a syndrome to differentiate among the different kinds of exceptions that generate the same interrupt type. Upon generation of a specific exception type, the associated bits are set and all other bits are cleared.
SPE 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.
Other Interrupt Registers	
DEAR	Data exception address register—Contains 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 1-5 and Chapter 7, "Interrupts and Exceptions," for more information.
MSR	Machine state register—Defines the state of the processor. When an interrupt occurs, it is updated to preclude unrecoverable interrupts from occurring during the initial portion of the interrupt handler
DBSR	Debug status register—Contains status on debug events and the most recent processor reset. When debug interrupts are enabled, a set bit in DBSR that is not MRR, VLES, or CNT1TRG causes a debug interrupt to be generated.

Each interrupt has an associated interrupt vector address, obtained by concatenating IVPR[32–47] 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 1-5 lists IVOR registers implemented on the e200z7 core and the associated interrupts.

Table 1-5. Exceptions and Conditions

IVOR*n*	Interrupt Type	IVOR*n*	Interrupt Type
None[1]	System reset (not an interrupt)	10	Decrementer
0[2]	Critical input	11	Fixed-interval timer
1	Machine check	12	Watchdog timer
2	Data storage	13	Data TLB error
3	Instruction storage	14	Instruction TLB error
4[2]	External input	15	Debug
5	Alignment	16–31	Reserved
6	Program	32	SPE unavailable
7	Floating-point unavailable	33	SPE data exception
8	System call	34	SPE round exception
9	APU unavailable (not used by this core)		

[1] Vector to [*p_rstbase[0:29]*] || 0xFFC.

[2] Auto-vectored external and critical input interrupts use this IVOR. Vectored interrupts supply an interrupt vector offset directly.

1.3 Microarchitecture Summary

The e200z7 processor has a ten-stage pipeline with four stages for instruction execution. These stages operate in an overlapped fashion, allowing single clock instruction execution for most instructions.

1. Instruction fetch 0
2. Instruction fetch 1
3. Instruction fetch 2
4. Instruction decode 0
5. Instruction decode 1/register file read/effective address calculation
6. Execute 0/memory access 0
7. Execute 1/memory access 1
8. Execute 2/memory access 2
9. Execute 3
10. Register writeback

The integer execution unit consists of a 32-bit arithmetic unit, a logic unit, a 32-bit barrel shifter, a mask-insertion unit, a condition register manipulation unit, a count-leading-zeros unit, a 32 × 32 hardware multiplier array, result feed-forward hardware, and support hardware for division.

Most arithmetic and logical operations are executed in a single cycle with the exception of multiply, which is implemented with a pipelined hardware array, and the divide instructions. A count-leading-zeros unit operates in a single clock cycle.

The instruction unit contains a program counter incrementer and a dedicated branch address adder to minimize delays during change-of-flow operations. Sequential prefetching is performed to ensure a supply of instructions into the execution pipeline. Branch target prefetching is performed to accelerate taken branches. Prefetched instructions are placed into an instruction buffer.

Branch target addresses are calculated in parallel with branch instruction decode, resulting in execution time of four clocks for correctly predicted branches. Conditional branches which are not taken execute in a single clock. Branches with successful BTB target prefetching have an effective execution time of one clock if correctly predicted.

Memory load and store operations are provided for byte, half-word, word (32-bit), and double-word data with automatic zero or sign extension of byte and half-word load data as well as optional byte reversal of data. These instructions can be pipelined to allow effective single-cycle throughput. Load and store multiple word instructions allow low-overhead context save and restore operations. The load/store unit (LSU) contains a dedicated effective address adder to optimize effective address generation.

The condition register unit supports the condition register (CR) and condition register operations defined by the architecture. The CR consists of eight 4-bit fields that reflect the results of certain operations generated by instructions such as move, integer and floating-point compare, arithmetic, and logical instructions. The CR also provides a mechanism for testing and branching.

Vectored and auto-vectored interrupts are supported by the CPU. Vectored interrupt support is provided to allow multiple interrupt sources to have unique interrupt handlers invoked with no software overhead.

The SPE category supports vector instructions operating on 8, 16- and 32-bit integer and fractional data types. The vector and scalar floating-point instructions operate on 32-bit IEEE Std 754™ single-precision floating-point formats, and support single-precision floating-point operations in a pipelined fashion.

The 64-bit GPRs are used for source and destination operands for all vector instructions, and there is a unified storage model for single-precision floating-point data types of 32 bits and the normal integer type. The following low latency fixed-point and floating-point operations are provided:

- Add
- Subtract
- Mixed Add/subtract
- Sum
- Diff
- Min
- Max
- Multiply
- Multiply-add
- Multiply-sub
- Divide
- Square Root
- Compare
- Conversion

Most operations can be pipelined.

1.3.1 Instruction Unit Features

The e200z7 instruction unit implements the following:

- 64-bit fetch path that supports fetching of two 32-bit or up to four 16-bit VLE instructions per clock
- Instruction buffer holds up to ten 32-bit instructions
- Dedicated PC incrementer supporting instruction prefetches
- Branch processing unit with dedicated branch address adder and branch target buffer (BTB) supporting single-cycle execution of successfully predicted branches

1.3.2 Integer Unit Features

The integer unit supports single-cycle execution of most integer instructions:

- 32-bit AU for arithmetic and comparison operations
- 32-bit LU for logical operations
- 32-bit priority encoder for count-leading-zeros function
- 32-bit single-cycle barrel shifter for static shifts and rotates
- 32-bit mask unit for data masking and insertion

- Divider logic for signed and unsigned divide in 4 to 15 clocks with minimized execution timing (EU1 only)
- Pipelined 32 × 32 hardware multiplier array that supports 32 × 32 → 32 multiply with 3-clock latency, 1-clock throughput

1.3.3 Load/Store Unit (LSU) Features

The e200z7 LSU supports load, store, and load multiple/store multiple instructions:

- 32-bit effective address adder for data memory address calculations
- Pipelined operation supports throughput of one load or store operation per cycle
- Dedicated 64-bit interface to memory supports saving and restoring of up to 2 registers per cycle for load multiple and store multiple word instructions

1.3.4 L1 Cache Features

The features of the cache are as follows:

- Separate 16-KB, 4 way set-associative instruction and data caches
- Copy-back and write-through support
- Eight-entry store buffer
- Push buffer
- Line-fill buffers with critical double-word forwarding for both data loads and instruction fetches
- 32-bit address bus plus attributes and control
- Separate unidirectional 64-bit read and 64-bit write data buses
- Cache line locking
- Data cache locking control instructions-Data Cache Block Touch and Lock Set (**dcbtls**), Data Cache Block Touch for Store and Lock Set (**dcbtstls**), and Data Cache Block Lock Clear (**dcblc**).
- Instruction cache locking control instructions-Instruction Cache Block Touch and Lock Set (**icbtls**) and Instruction Cache Block Lock Clear (**icblc**)
- Way allocation
- Write allocation policies
- Tag and data parity
- Hardware cache coherency support for the data cache
- Supports multibit EDC for the instruction cache
- Correction/auto-invalidation capability for the intstruction and data caches
- e200z7-specific L1 cache flush and invalidate registers (L1FINV0 and L1FINV1) support software-based flush and invalidation control on a set and way basis

1.3.5 Memory Management Unit (MMU) Features

The MMU is an implementation of the embedded MMU category of the Power ISA, with the following feature set:

- 32-bit effective-to-real address translation
- 8-bit process identifier (PID)
- 64-entry, fully associative TLB
- Support for multiple page sizes (1 KB to 4 GB)
- Software managed by **tlbre, tlbwe, tlbsx, tlbsync,** and **tlbivax** instructions
- Entry flush protection

1.3.6 System Bus (Core Complex Interface) Features

The features of the core complex interface are as follows:

- Independent instruction and data buses
- Advanced microcontroller bus architecture (AMBA) and advanced high-performance bus (AHB2.v6)-Lite protocol
- 32-bit address bus, 64-bit data bus, plus attributes and control on each bus
- Instruction interface supports read transfers of 16, 32, and 64 bits.
- Data interface has separate unidirectional 64-bit read data bus and 64-bit write data bus.
- Both the instruction and data interface buses support misaligned transfers, true big- and little-endian operating modes, and operates in a pipelined fashion
- Support for HCLK running at a slower rate than CPU clock

1.3.7 Nexus 3+ Module Features

The Nexus 3+ module provides real-time development capabilities for e200z7 processors in compliance with the IEEE-ISTO 5001™-2008 standard. This module provides development support capabilities without requiring the use of address and data pins for internal visibility. The ‘3+’ suffix indicates that some Nexus Class 4 features are implemented.

A portion of the pin interface (the JTAG port) is shared with the OnCE/Nexus 1 unit. The IEEE-ISTO 5001-2008 standard defines an extensible auxiliary port, which is used in conjunction with the JTAG port in e200z7 processors.

Chapter 2
Register Model

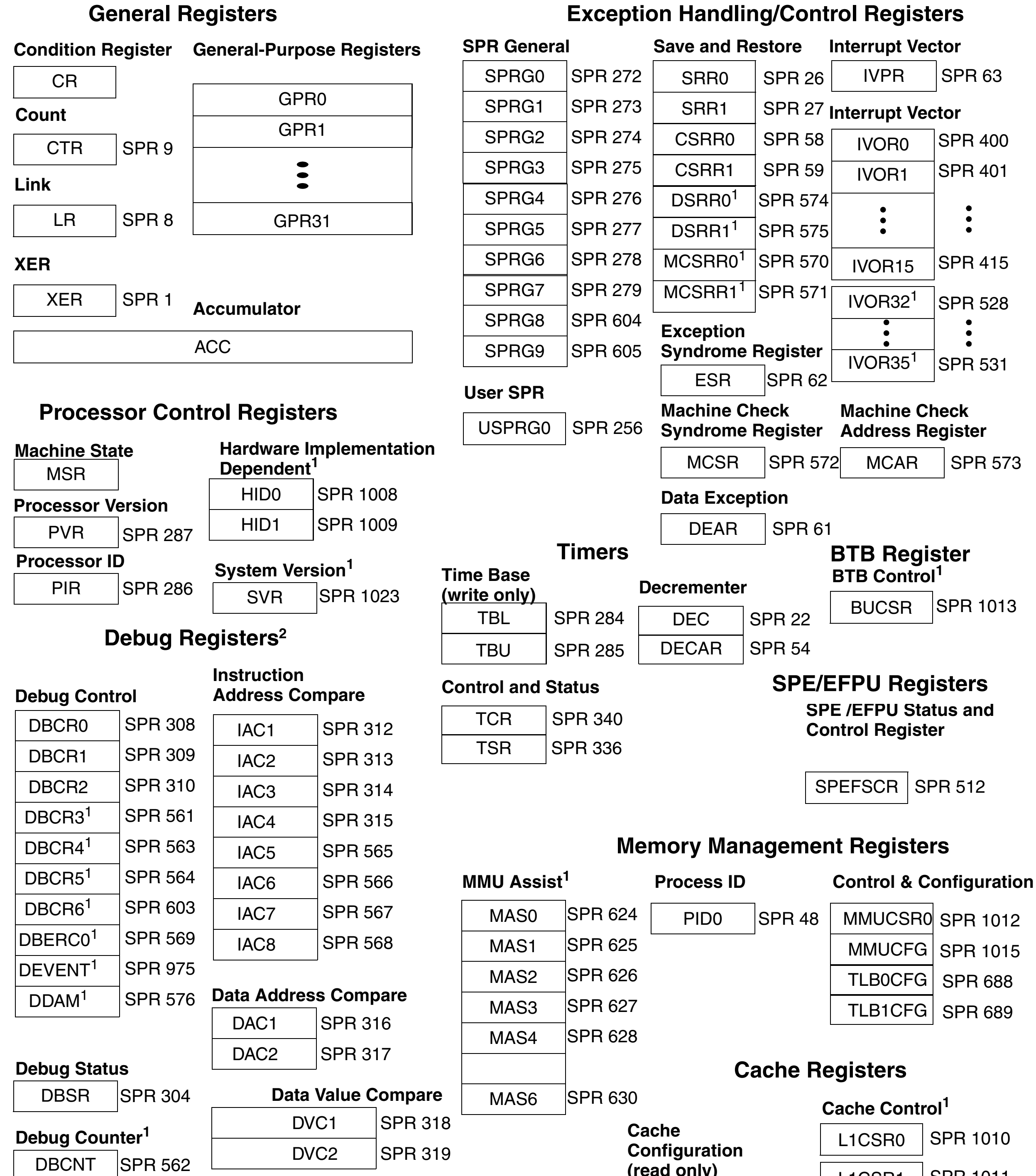
This section describes the registers implemented in the e200z7 core. It includes an overview of registers defined by the Power ISA embedded category architecture and highlights any differences in how these registers are implemented in the e200z7 core. This section also provides detailed descriptions of e200-specific registers. Full descriptions of the architecture-defined register set are provided in the *EREF*.

The Power ISA embedded category architecture defines register-to-register operations for all computational instructions. Source data for these instructions are accessed from the on-chip registers or are provided as immediate values embedded in the opcode. The three-register instruction format allows specification of a target register distinct from the two source registers, thus preserving the original data for use by other instructions. Data is transferred between memory and registers with explicit load and store instructions only.

The e200z7 extends the general-purpose registers (GPRs) to 64 bits for supporting the signal processing engine (SPE) and embedded floating point unit (EFPU) operations. Power ISA embedded category instructions operate on the lower 32 bits of the GPRs only, and the upper 32 bits are unaffected by these instructions. SPE vector instructions operate on the entire 64-bit register. The SPE defines load and store instructions for transferring 64-bit values to/from memory.

Figure 2-1–Figure 2-4 show the complete e200z7 register set divided into supervisor and user-level registers and grouped into general-purpose registers (GPRs), special-purpose registers (SPRs), device control registers (DCRs), and any performance monitor registers (PMRs) that may be implemented in a particular variation of the e200z7 core family The number to the right of the special-purpose registers (SPRs) is the decimal number used in the instruction syntax to access the register. For example, the integer exception register (XER) is SPR 1.

Figure 2-1 shows the supervisor mode programmer’s model.



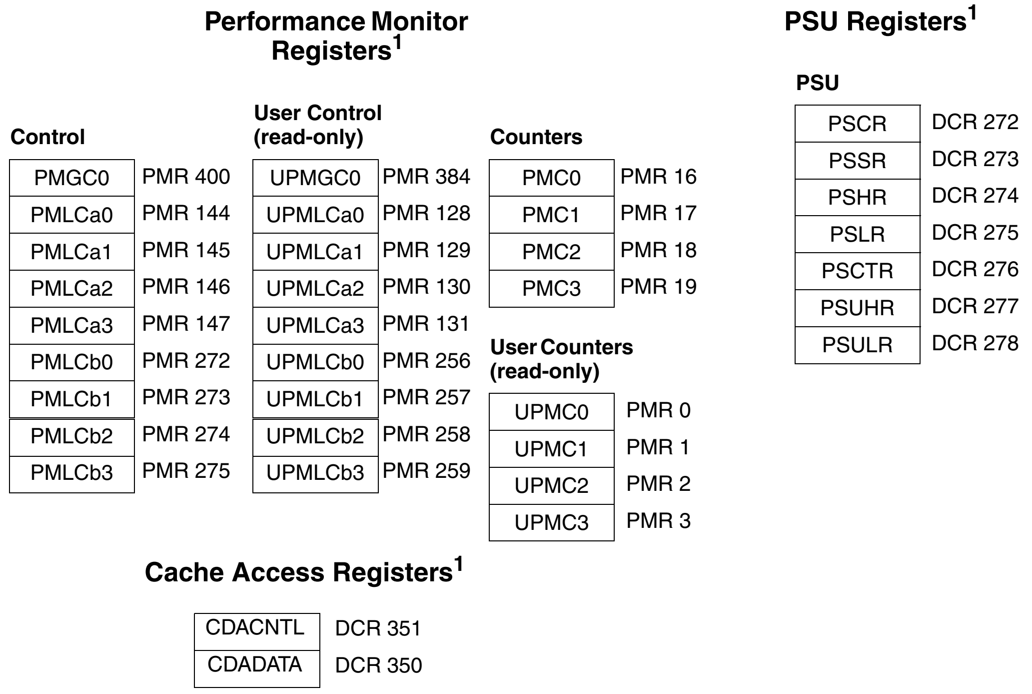
1 - These e200-specific registers may not be supported by other Power Architecture processors

2 - Optional registers defined by the Power ISA embedded architecture

3 - Read-only registers

Figure 2-1. e200z760 Supervisor Mode Programmer’s Model

Figure 2-2 shows the supervisor mode programmer models's DCRs and PMRs.



Note:

[1] These e200-specific registers may not be supported by other Power ISA embedded category processors

Figure 2-2. e200z760 Supervisor Mode Programmer's Model DCRs and PMRs

Figure 2-3 shows the user mode programmer's model.

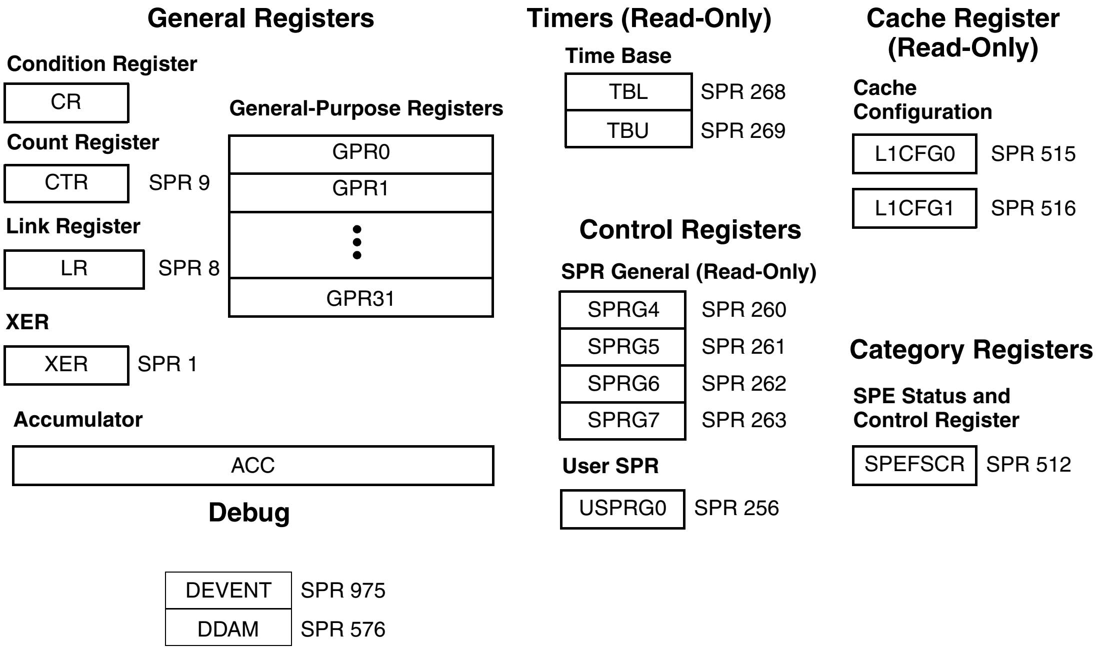
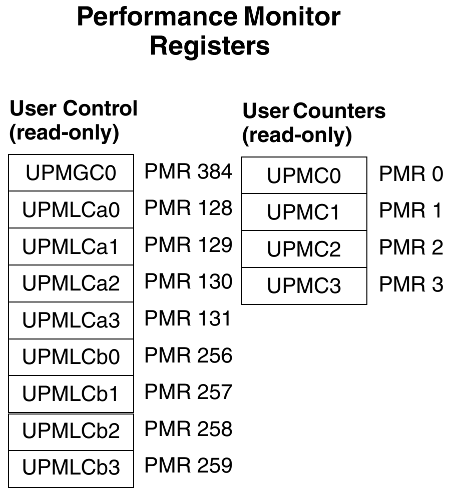


Figure 2-3. e200z7 User Mode Programmer's Model

Figure 2-4 shows the user mode programmer’s model PMRs.



Note:

These e200-specific registers may not be supported by other Power ISA embedded category processors.

Figure 2-4. e200 User Mode Programmer’s Model PMRs

The GPRs are accessed through instruction operands. Access to other registers can be explicit (by using instructions for that purpose such as Move to Special Purpose Register (**mtspr**) and Move from Special Purpose Register (**mfspr**) instructions) or implicit as part of the execution of an instruction. Some registers are accessed both explicitly and implicitly.

2.1 Power ISA Embedded Category Registers

The e200z7 supports most of the registers defined by the Power ISA embedded category architecture. Notable exceptions are the floating-point registers FPR0–FPR31 and FPSCR. The e200z7 does not support the Power ISA floating-point category functionality in hardware. The GPRs have been extended to 64 bits. The *EREF* contains complete descriptions of the Power ISA embedded registers, but there are described briefly as follows:

- User-level registers—The user-level registers can be accessed by all software with either user or supervisor privileges. They include the following:
  - General-purpose registers (GPRs). The thirty-two 64-bit GPRs (GPR0–GPR31) serve as data source or destination registers for integer instructions and provide data for generating addresses. Power ISA embedded category instructions affect only the lower 32 bits of the GPRs. SPE and EFPU instructions are provided which operate on the entire 64-bit register.
  - Condition register (CR). The 32-bit CR consists of eight 4-bit fields, CR0–CR7, that reflect results of certain arithmetic operations and provide a mechanism for testing and branching.

  The remaining user-level registers are SPRs. Note that the Power ISA embedded category architecture provides the **mtspr** and **mfspr** instructions for accessing SPRs.

  - Integer exception register (XER). The XER indicates overflow and carries for integer operations.

  - Link register (LR). The LR provides the branch target address for the branch [conditional] to link register (**bclr**, **bclrl**, **se_blr**, **se_blrl**) instructions, and is used to hold the address of the instruction that follows a branch and link instruction, typically used for linking to subroutines.
  - Count register (CTR). The CTR holds a loop count that can be decremented during execution of appropriately coded branch instructions. The CTR also provides the branch target address for the branch [conditional] to count register (**bcctr**, **bcctrl**, **se_bctr**, **se_bctrl**) instructions.
  - Time base (TB). The TB facility consists of two 32-bit registers—time base upper (TBU) and time base lower (TBL). These two registers are accessible in a read-only fashion to user-level software.
  - SPRG4–SPRG7. The Power ISA embedded category architecture defines software-use special purpose registers (SPRGs). SPRG4–SPRG7 are accessible in a read-only fashion by user-level software. The e200 does not allow user mode access to the SPRG3 register (defined as implementation dependent by Power ISA).
  - USPRG0. The Power ISA embedded category architecture defines user software-use special purpose register (USPRG0). The USPRG0 is accessible in a read-write fashion by user-level software.
- Supervisor-level registers—In addition to the registers accessible in user mode, supervisor-level software has access to additional control and status registers used for configuration, exception handling, and other operating system functions. The Power ISA embedded category architecture defines the following supervisor-level registers:
  - Processor control registers
    - Machine state register (MSR). The MSR defines the state of the processor. The MSR can be modified by the move to machine state register (**mtmsr**), system call (**sc**, **se_sc**), and return from exception (**rfi**, **rfci**, **rfdi**, **rfmci**, **se_rfi**, **se_rfci**, **se_rfdi**, **se_rfmci**) instructions. It can be read by the Move from Machine State Register (**mfmsr)** instruction. When an interrupt occurs, the contents of the MSR are saved to one of the machine state save/restore registers (SRR1, CSRR1, DSRR1, MCSRR1).
    - Processor version register (PVR). This register is a read-only register that identifies the version (model) and revision level of the processor.
    - Processor identification register (PIR). This read/write register is provided to distinguish the processor from other processors in the system.
  - Storage control register
    - Process ID register (PID, also referred to as PID0). This register is provided to indicate the current process or task identifier. It is used by the MMU as an extension to the effective address, and by external Nexus 2/3/4 modules for ownership trace message generation. The Power ISA embedded category architecture allows for multiple PIDs; the e200z7 implements only one.
  - Interrupt registers
    - Data exception address register (DEAR). After most data storage interrupts (DSI), or on an alignment interrupt or data TLB miss interrupt, the DEAR is set to the effective address (EA) generated by the faulting instruction.

  - SPRG0–SPRG7, USPRG0. The SPRG0–SPRG7 and USPRG0 registers are provided for operating system use. The e200 does not allow user mode access to the SPRG3 register (defined as implementation dependent by the Power ISA embedded category architecture).
  - Exception syndrome register (ESR). The ESR register provides a syndrome to differentiate between the different kinds of exceptions which can generate the same interrupt.
  - Interrupt vector prefix register (IVPR) and the interrupt vector offset registers (IVOR0–IVOR15, IVOR32–IVOR35). These registers together provide the address of the interrupt handler for different classes of interrupts.
  - Save/restore register 0 (SRR0). The SRR0 register is used to save machine state on a noncritical interrupt, and contains the address of the instruction at which execution resumes when an **rfi** or **se_rfi** instruction is executed at the end of a noncritical class interrupt handler routine.
  - Critical save/restore register 0 (CSRR0). The CSRR0 register is used to save machine state on a critical interrupt, and contains the address of the instruction at which execution resumes when an **rfci** or **se_rfci** instruction is executed at the end of a critical class interrupt handler routine.
  - Save/restore register 1 (SRR1). The SRR1 register is used to save machine state from the MSR on noncritical interrupts, and to restore machine state when an **rfi** or **se_rfi** executes.
  - Critical save/restore register 1 (CSRR1). The CSRR1 register is used to save machine state from the MSR on critical interrupts, and to restore machine state when **rfci** or **se_rfci** executes.
- Debug facility registers
  - Debug control registers (DBCR0–DBCR2). These registers provide control for enabling and configuring debug events.
  - Debug status register (DBSR). This register contains debug event status.
  - Instruction address compare registers (IAC1–IAC4). These registers contain addresses and/or masks which are used to specify instruction address compare debug events.
  - Data address compare registers (DAC1–DAC2). These registers contain addresses and/or masks which are used to specify data address compare debug events.
  - Data value compare registers (DVC1–DVC2). These registers contain data values which are used to specify Data value compare debug events.
- Timer registers
  - Time base (TB). The TB is a 64-bit structure provided for maintaining the time of day and operating interval timers. The TB consists of two 32-bit registers, Time base upper (TBU) and time base lower (TBL). The time base registers can be written only by supervisor-level software, but can be read by both user and supervisor-level software.
  - Decrementer register (DEC). This register is a 32-bit decrementing counter that provides a mechanism for causing a decrementer exception after a programmable delay.
  - Decrementer auto-reload (DECAR). This register is provided to support the auto-reload feature of the decrementer.

    - Timer control register (TCR). This register controls decrementer, fixed-interval timer, and watchdog timer options.
    - Timer status register (TSR). This register contains status on timer events and the most recent watchdog timer-initiated processor reset.

2.2 e200-Specific Special Purpose Registers

The Power ISA embedded category architecture allows implementation-specific special purpose registers. Those incorporated in the e200 core are as follows:

- User-level registers—The user-level registers can be accessed by all software with either user or supervisor privileges. They include the following:
  - Signal processing extension/embedded floating-point unit status and control register (SPEFSCR). The SPEFSCR contains all fixed-point and floating-point exception signal bits, exception summary bits, exception enable bits, and rounding control bits needed for compliance with IEEE 754. See Section 6.2.3, “SPE Status and Control Register (SPEFSCR).”
  - The L1 cache configuration registers (L1CFG0, L1CGF1). These read-only registers allows software to query the configuration of the L1 Harvard caches.
- Supervisor-level registers—The following supervisor-level registers are defined in the e200 in addition to the Power ISA embedded category registers described above:
  - Configuration registers
    - Hardware implementation dependent register 0 (HID0). This register controls various processor and system functions.
    - Hardware implementation dependent register 1 (HID1). This register controls various processor and system functions.
  - Exception handling and control registers
    - Machine check save/restore register 0 (MCSRR0). The MCSRR0 register is used to save machine state on a machine check interrupt, and contains the address of the instruction at which execution resumes when an **rfmci** or **se_rfmci** instruction is executed.
    - Machine Check save/restore register 1 (MCSRR1). The MCSRR1 register is used to save machine state from the MSR on machine check interrupts, and to restore machine state when an **rfmci** or **se_rfmci** instruction is executed.
    - Machine check syndrome register (MCSR). This register provides a syndrome to differentiate between the different kinds of conditions which can generate a machine check.
    - Machine check address register (MCAR). This register provides an address associated with certain machine checks.
    - Debug save/restore register 0 (DSRR0). When enabled, the DSRR0 register is used to save the address of the instruction at which execution continues when an **rfdi** or **se_rfdi** instruction executes at the end of a debug interrupt handler routine.
    - Debug save/restore register 1 (DSRR1). When enabled, the DSRR1 register is used to save machine status on debug interrupts and to restore machine status when an **rfdi** or **se_rfdi** instruction executes.

  - SPRG8 and SPRG9. The SPRG8 and SPRG9 registers are provided for operating system use for the machine check and debug APUs.
- Debug facility registers
  - Instruction address compare registers (IAC5–IAC8). These registers contain addresses and/or masks which are used to specify instruction address compare debug events.
  - Debug control registers (DBCR3–DBCR6). These registers provides control for debug functions not described in Power ISA embedded category architecture.
  - Debug external resource control register 0 (DBERC0). This register provides control for debug functions not described in Power ISA embedded category architecture.
  - Debug counter register (DBCNT). This register provides counter capability for debug functions.
- Branch unit control and status register (BUCSR) controls operation of the BTB
- Cache registers
  - L1 cache configuration registers (L1CFG0, L1CFG1) is a read-only register that allows software to query the configuration of the L1 caches.
  - L1 cache control and status registers (L1CSR0, L1CSR1) control the operation of the L1 caches such as cache enabling, cache invalidation, cache locking, etc.
  - L1 cache flush and invalidate registers (L1FINV0, L1FINV1) controls software flushing and invalidation of the L1 caches.
- Memory management unit registers
  - MMU configuration register (MMUCFG) is a read-only register that allows software to query the configuration of the MMU.
  - MMU assist (MAS0–MAS4, MAS6) registers. These registers provide the interface to the e200 core from the MMU.
  - MMU control and status register (MMUCSR0) controls invalidation of the MMU.
  - TLB configuration registers (TLB0CFG, TLB1CFG) are read-only registers that allow software to query the configuration of the TLBs.
- System version register (SVR). This register is a read-only register that identifies the version (model) and revision level of the system which includes the e200 processor.

NOTE

It is not guaranteed that the implementation of e200 core-specific registers is consistent among the Power ISA embedded category processors, although other processors may implement similar or identical registers.

All e200 SPR definitions are compliant with the Freescale EIS definitions.

2.3 e200-Specific Device Control Registers

In addition to the SPRs described above, implementations may also choose to implement one or more device control registers (DCRs). The e200z7 implements a set of device control registers to perform a parallel signature capability in the parallel signature unit (PSU). These registers are described in Section 13.9, “Parallel Signature Unit.”

2.4 Special-Purpose Register Descriptions

This section describes the special-purpose registers. Each subsection contains an initial description followed by a register figure and a table of bit field definitions.

2.4.1 Machine State Register (MSR)

A complete description of the machine state register (MSR) is included in the *EREF*. The MSR defines the state of the processor. Chapter 7, “Interrupts and Exceptions,” describes how the MSR is affected when interrupts occur. The e200 MSR is shown in Figure 2-5.

Access: Read/Write

	0 4	5	6	7 12	13	14	15	16	17	18	19	20	21	22	23	24 25	26	27	28	29	30	31
R W	—	UCLE	SPE	—	WE	CE	—	EE	PR	FP	ME	FE0	—	DE	FE1	—	IS	DS	—	PMM	RI	—
Reset	All zeros																					

Figure 2-5. Machine State Register (MSR)

The MSR bits are defined in Table 2-1.

Table 2-1. MSR Field Descriptions

Bits	Name	Description
0–4 (32–36)	—	Reserved
5 (37)	UCLE	User Cache Lock Enable 0 Execution of the cache locking instructions in user mode (MSR[PR] = 1) disabled; DSI exception taken instead, and ILK or DLK set in ESR. 1 Execution of the cache lock instructions in user mode enabled.
6 (38)	SPE	SPE/EFPU Available 0 Execution of SPE and EFPU vector instructions is disabled; SPE/EFPU unavailable exception taken instead, and SPE bit is set in ESR. 1 Execution of SPE and EFPU vector instructions is enabled.
7–12 (39–44)	—	Reserved
13 (45)	WE	Wait State (Power Management) Enable 0 Power management is disabled. 1 Power management is enabled. The processor can enter a power-saving mode when additional conditions are present. The mode chosen is determined by the DOZE, NAP, and SLEEP bits in the HID0 register, described in Section 2.4.11, “Hardware Implementation Dependent Register 0 (HID0).”
14 (46)	CE	Critical Interrupt Enable 0 Critical input and watchdog timer interrupts are disabled. 1 Critical input and watchdog timer interrupts are enabled.
15 (47)	—	Reserved

Table 2-1. MSR Field Descriptions (continued)

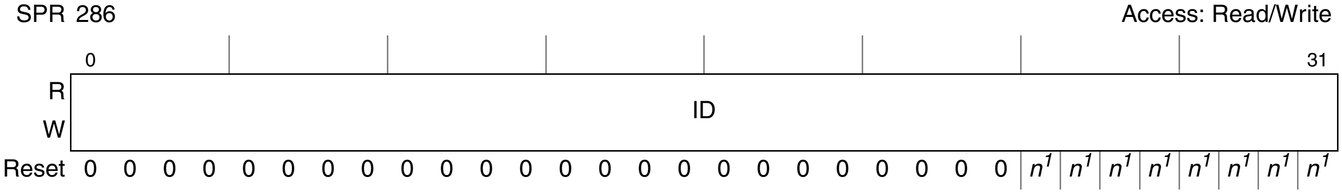
Bits	Name	Description
16 (48)	EE	External Interrupt Enable 0 External input, decrementer, and fixed-interval timer interrupts are disabled. 1 External input, decrementer, and fixed-interval timer interrupts are enabled.
17 (49)	PR	Problem State 0 The processor is in supervisor mode, can execute any instruction, and can access any resource (for example, GPRs, SPRs, MSR, etc.). 1 The processor is in user mode, cannot execute any privileged instruction, and cannot access any privileged resource.
18 (50)	FP	Floating-Point Available 0 Floating-point unit is unavailable. The processor cannot execute floating-point instructions, including floating-point loads, stores, and moves. (An FP unavailable interrupt is generated on attempted execution of floating-point instructions.) 1 Floating-point unit is available. The processor can execute floating-point instructions. Note that for the e200, the floating-point unit is not supported in hardware, and an unimplemented operation exception is generated for attempted execution of Power ISA embedded category floating-point instructions when FP is set.
19 (51)	ME	Machine Check Enable 0 Asynchronous machine check interrupts are disabled. 1 Asynchronous machine check interrupts are enabled.
20 (52)	FE0	Floating-Point Exception Mode 0 (not used by e200)
21 (53)	—	Reserved
22 (54)	DE	Debug Interrupt Enable 0 Debug interrupts are disabled. 1 Debug interrupts are enabled.
23 (55)	FE1	Floating-Point Exception Mode 1 (not used by e200)
24 (56)	—	Reserved
25 (57)	—	Reserved
26 (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).
27 (59)	DS	Data Address Space 0 The processor directs all data storage accesses to address space 0 (TS = 0 in the relevant TLB entry). 1 The processor directs all data storage accesses to address space 1 (TS = 1 in the relevant TLB entry).
28 (60)	—	Reserved

Table 2-1. MSR Field Descriptions (continued)

Bits	Name	Description
29 (61)	PMM	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 time. If this state matches an individual state specified in the performance monitor registers PMLCa n, the state for which monitoring is enabled, counting is enabled.
30 (62)	RI	Recoverable Interrupt. This bit is provided for software use to detect nested exception conditions. This bit is cleared by hardware when a machine check interrupt is taken
31 (63)	—	Reserved

2.4.2 Processor ID Register (PIR)

The processor ID for the CPU core is contained in the processor ID register (PIR), shown in Figure 2-6. The contents of the PIR register are a reflection of hardware input signals to the e200 core following reset. This register may be written by software to modify the default reset value.



[1] Updated to reflect the values on *p_cpuid*[0:7]

Figure 2-6. Processor ID Register (PIR)

The PIR fields are defined in Table 2-2.

Table 2-2. PIR Field Descriptions

Bits	Name	Description
0–23	ID	These bits are reset to 0 and are writable by software.
24–31		These bits are reset to the values provided on the *p_cpuid*[0:7] input signals and are writable by software.

2.4.3 Processor Version Register (PVR)

The processor version register (PVR), shown in Figure 2-7, contains the processor version number for the CPU core.

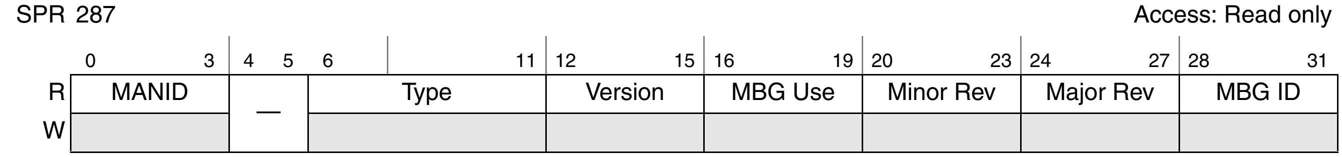


Figure 2-7. Processor Version Register (PVR)

This register contains fields to specify a particular implementation of an e200 family member. This register is read only. Interface signals *p_pvrin*[16:31] provide the contents of a portion of this register.

Table 2-3. PVR Field Descriptions

Bits	Name	Description
0–3	MANID	These bits identify the manufacturer ID. Freescale is 0b1000.
4–5	—	Reserved
6–11	Type	These bits identify the processor type. e200z7 is 0b010110.
12–15	Version	These bits identify the version of the processor and inclusion of optional elements. For e200z760n3, these are tied to 0b0011.
16–19	MBG Use	These bits are allocated for use by Freescale to distinguish different system variants and are provided by the *p_pvrin*[16:19] input signals.
20–23	Minor Rev	These bits distinguish between implementations of the version and are provided by the *p_pvrin*[20:23] input signals.
24–27	Major Rev	These bits distinguish between implementations of the version and are provided by the *p_pvrin*[24:27] input signals.
28–31	MBG ID	These bits identify the Freescale organization responsible for a particular mask set and are provided by the *p_pvrin*[28:31] input signals. MBG value of 0b0000 is reserved.

2.4.4 System Version Register (SVR)

The system version register (SVR), shown in Figure 2-8, contains system version information for an e200-based SoC.

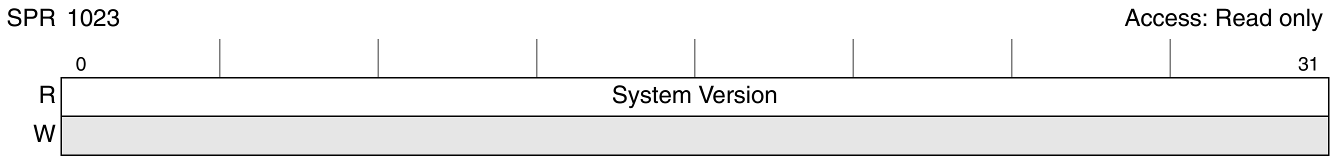


Figure 2-8. System Version Register (SVR)

This register is used to specify a particular implementation of an e200-based system. This register is read only.

Table 2-4. SVR Field Descriptions

Bits	Name	Description
0–31	System Version	These bits are allocated for use by Freescale to distinguish different system variants, and are provided by the *p_sysvers*[0:31] input signals

2.4.5 Integer Exception Register (XER)

A complete description of the integer exception register (XER) can be found in the *EREF*. The XER bit assignments are shown in Figure 2-9.

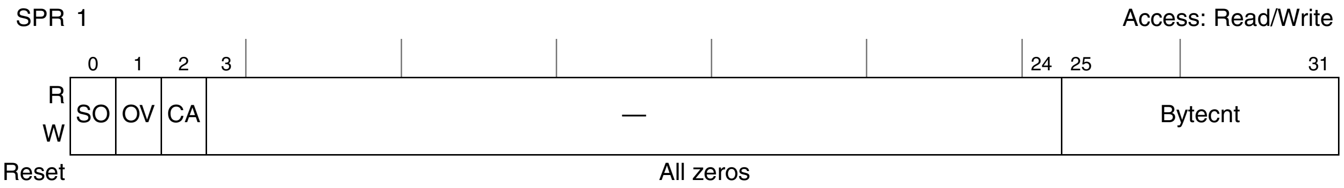


Figure 2-9. Integer Exception Register (XER)

The XER fields are defined in Table 2-5.

Table 2-5. XER Field Descriptions

Bits	Name	Description
0 (32)	SO	Summary Overflow (per Power ISA embedded category)
1 (33)	OV	Overflow (per Power ISA embedded category)
2 (34)	CA	Carry (per Power ISA embedded category)
3–24 (35–56)	—	Reserved
25–31 (57–63)	Bytecnt[1]	Reserved for **lswi**, **lswx**, **stswi**, **stswx** string instructions

[1] These bits are implemented to support emulation of the string instructions.

2.4.6 Exception Syndrome Register

A complete description of the exception syndrome register (ESR) can be found in the *EREF*. The exception syndrome register (ESR) provides a syndrome to differentiate between exceptions that can generate the same interrupt type. The e200 adds some implementation-specific bits to this register, as seen in Figure 2-10.

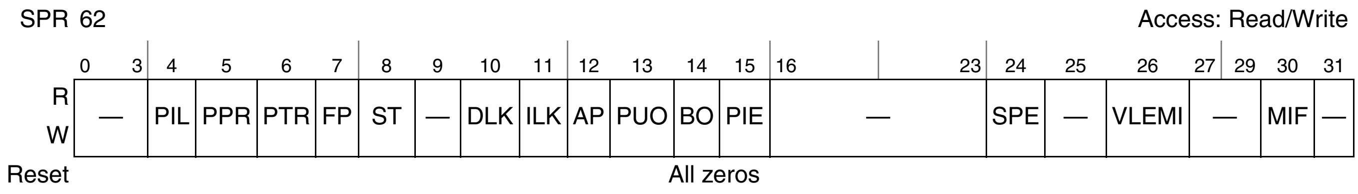


Figure 2-10. Exception Syndrome Register (ESR)

The ESR fields are defined in Table 2-6.

Table 2-6. ESR Field Descriptions

Bit(s)	Name	Description	Associated Interrupt Type
0–3 (32–35)	—	Reserved	—
4 (36)	PIL	Illegal Instruction Exception	Program
5 (37)	PPR	Privileged Instruction Exception	Program
6 (38)	PTR	Trap Exception	Program
7 (39)	FP	Floating-Point Operation	Alignment Data storage Data TLB Program
8 (40)	ST	Store Operation	Alignment Data storage Data TLB
9 (41)	—	Reserved	—
10 (42)	DLK	Data Cache Locking	Data storage
11 (43)	ILK	Instruction Cache Locking	Data storage
12 (44)	AP	Auxiliary Processor Operation (Currently unused in e200)	Alignment Data storage Data TLB Program
13 (45)	PUO	Unimplemented Operation Exception	Program
14 (46)	BO	Byte Ordering Exception Mismatched Instruction Storage Exception	Data storage Instruction storage
15 (47)	PIE	Program Imprecise Exception (Reserved)	Currently unused in e200

Table 2-6. ESR Field Descriptions (continued)

Bit(s)	Name	Description	Associated Interrupt Type
16–23 (48–55)	—	Reserved	—
24 (56)	SPE	SPE/EFPU Operation	SPE/EFPU unavailable EFPU floating-point data exception EFPU floating-point round exception Alignment Data storage Data TLB
25 (57)	—	Reserved	—
26 (58)	VLEMI	VLE Mode Instruction	SPE/EFPU unavailable EFPU floating-point Data exception EFPU floating-point Round exception Data storage Data TLB Instruction storage Alignment Program System call
27–29 (59–61)	—	Reserved	—
30 (62)	MIF	Misaligned Instruction Fetch	Instruction storage Instruction TLB
31 (63)	—	Reserved	—

2.4.6.1 Power ISA VLE Mode Instruction Syndrome

The ESR[VLEMI] is provided to indicate that an interrupt was caused by a Power ISA VLE instruction. This syndrome bit is set on an exception associated with execution or attempted execution of a Power ISA VLE instruction. This bit is updated for the interrupt types indicated in Table 2-6.

2.4.6.2 Misaligned Instruction Fetch Syndrome

The ESR[MIF] is provided to indicate that an instruction storage interrupt was caused by an attempt to fetch an instruction from a Power ISA embedded category page which was not aligned on a word boundary. The fetch may have been caused by execution of a branch class instruction from a VLE page to a non-VLE page, a branch to LR instruction with LR[62] = 1, a branch to CTR instruction with CTR[62] = 1, execution of an **rfi** or **se_rfi** instruction with SRR0[62] = 1, execution of an **rfci** or **se_rfci** instruction with CSRR0[62] = 1, execution of an **rfdi** or **se_rfdi** instruction with DSRR0[62] = 1, or execution of an **rfmci** or **se_rfmci** instruction with MCSRR0[62] = 1, where the destination address corresponds to an instruction page which is not marked as a Power ISA VLE page.

ESR[MIF] is also used to indicate that an instruction TLB interrupt was caused by a TLB miss on the second half of a misaligned 32-bit Power ISA VLE instruction. For this case, SRR0 will be pointing to the first half of the instruction, which will reside on the previous page from the miss at page offset 0xFFE. The ITLB handler may need to realize that the miss corresponds to the next page, although MMU MAS2 contents will correctly reflect the page corresponding to the miss.

2.4.7 Machine Check Syndrome Register (MCSR)

When the core complex takes a machine check interrupt, it updates the machine check syndrome register (MCSR) to differentiate between machine check conditions. Figure 2-11 shows the MCSR.

SPR 572

Access: w1c

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	MCP	IC_ DPERR	CP_ PERR	DC_ DPERR	EXCP _ERR	IC_ TPERR	DC_ TPERR	IC_ LKERR	DC_ LKERR	—		NMI	MAV	MEA	—	IF
W	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c			w1c	w1c	w1c		w 1c
Reset	All zeros															

	16	17	18	19	25	26	27	28	29	30	31
R	LD	ST	G	—		SNPERR	BUS_ IRERR	BUS_ DRERR	BUS_ WRERR	—	
W	w1c	w1c	w1c			w1c	w1c	w1c	w1c		
Reset	All zeros										

Figure 2-11. Machine Check Syndrome Register (MCSR)

Table 2-7 describes MCSR fields. The MCSR indicates the source of a machine check condition. When an async mchk or error report syndrome bit in the MCSR is set, the core complex asserts *p_mcp_out* for system information. Note that the bits in the MCSR are implemented as write one to clear, so software must write ones into those bit positions it wishes to clear, typically by writing back what was originally read. See Section 7.6.2, “Machine Check Interrupt (IVOR1),” for more details of the MCSR settings.

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

Bits	Name	Description	Exception Type[1]	Recoverable
0 (32)	MCP	Machine check input pin	Async Mchk	Maybe
1 (33)	IC_DPERR	Instruction Cache data array parity error	Async Mchk	Precise
2 (34)	CP_PERR	Data Cache push parity error	Async Mchk	Unlikely
3 (35)	DC_DPERR	Data Cache data array parity error	Async Mchk	Maybe
4 (36)	EXCP_ERR	ISI, ITLB, or Bus Error on first instruction fetch for an exception handler	Async Mchk	Precise

Table 2-7. Machine Check Syndrome Register (MCSR) (continued)

Bits	Name	Description	Exception Type[1]	Recoverable
5 (37)	IC_TPERR	Instruction Cache Tag parity error	Async Mchk	Precise
6 (38)	DC_TPERR	Data Cache Tag parity error	Async Mchk	Maybe
7 (39)	IC_LKERR	Instruction Cache Lock error Indicates a cache control operation or invalidation operation invalidated one or more locked lines in the Icache	Status	—
8 (40)	DC_LKERR	Data Cache Lock error Indicates a cache control operation or instruction invalidation operation invalidated one or more locked lines in the Dcache	Status	—
9–10 (41–42)	—	Reserved	—	—
11 (43)	NMI	NMI input pin	NMI	—
12 (44)	MAV	MCAR Address Valid Indicates that the address contained in the MCAR was updated by hardware to correspond to the first detected Async Mchk error condition	Status	—
13 (45)	MEA	MCAR holds Effective Address If MAV = 1, MEA = 1 indicates that the MCAR contains an effective address and MEA = 0 indicates that the MCAR contains a physical address	Status	—
14 (46)	—	Reserved	—	—
15 (47)	IF	Instruction Fetch Error Report An error occurred during the attempt to fetch an instruction. MCSRR0 contains the instruction address.	Error Report	Precise
16 (48)	LD	Load type instruction Error Report An error occurred during the attempt to execute the load type instruction located at the address stored in MCSRR0.	Error Report	Precise
17 (49)	ST	Store type instruction Error Report An error occurred during the attempt to execute the store type instruction located at the address stored in MCSRR0.	Error Report	Precise
18 (50)	G	Guarded instruction Error Report An error occurred during the attempt to execute the load or store type instruction located at the address stored in MCSRR0 and the access was guarded and encountered an error on the external bus.	Error Report	Precise
19–25 (51–57)	—	Reserved	—	—
26 (58)	SNPERR	Snoop Lookup Error An error occurred during certain snoop operations. This is typically due to a data cache tag parity error, in which case DC_TPERR will also be set.	Async Mchk	Unlikely

Table 2-7. Machine Check Syndrome Register (MCSR) (continued)

Bits	Name	Description	Exception Type[1]	Recoverable
27 (59)	BUS_IRERR	Read bus error on Instruction fetch or linefill	Async Mchk	Precise if data used
28 (60)	BUS_DRERR	Read bus error on data load or linefill	Async Mchk	Precise if data used
29 (61)	BUS_WRERR	Write bus error on store or cache line push	Async Mchk	Unlikely
30–31 (62–63)	—	Reserved	—	—

[1] The Exception Type indicates the exception type associated with a given syndrome bit

- “Error Report” indicates that this bit is only set for error report exceptions which cause machine check interrupts. These bits are only updated when the machine check interrupt is actually taken. Error report exceptions are not gated by MSR[ME]. These are synchronous exceptions. These bits remain set until cleared by software writing a 1 to the bit position(s) to be cleared.
- “Status” indicates that this bit is provides additional status information regarding the logging of a machine check exception. These bits remain set until cleared by software writing a 1 to the bit position(s) to be cleared.
- “NMI” indicates that this bit is only set for the non-maskable interrupt type exception which causes a machine check interrupt. This bit is only updated when the machine check interrupt is actually taken. NMI exceptions are not gated by MSR[ME]. This is an asynchronous exception. This bit remains set until cleared by software writing a 1 to the bit position.
- “Async Mchk” indicates that this bit is set for an asynchronous machine check exception. These bits are set immediately upon detection of the error. Once any “Async Mchk” bit is set in the MCSR, a machine check interrupt will occur if MSR[ME] = 1. If MSR[ME] = 0, the machine check exception will remain pending. These bits remain set until cleared by software writing a 1 to the bit position(s) to be cleared.

2.4.8 Timer Control Register (TCR)

The timer control register (TCR) provides control information for the CPU timer facilities. A complete description of the TCR in included in the *EREF*. The TCR[WRC] field functions are defined to be implementation dependent and are described below. In addition, the e200 core implements two fields not specified in the Power ISA embedded category, TCR[WPEXT] and TCR[FPEXT]. Figure 2-12 shows the TCR.

SPR 340 Access: Read/Write

	0–1	2–3	4	5	6–7	8	9	10	11–14	15–18	19–31
R W	WP	WRC	WIE	DIE	FP	FIE	ARE	—	WPEXT	FPEXT	—
Reset	All zeros										

Figure 2-12. Timer Control Register (TCR)

The TCR fields are defined in Table 2-8.

Table 2-8. Timer Control Register Field Descriptions

Bits	Name	Description
0–1 (32–33)	WP	Watchdog Timer Period When concatenated with WPEXT, specifies 1 of 64-bit locations of the time base used to signal a watchdog timer exception on a transition from 0 to 1. TCR[wpext][0–3],TCR[wp][0–1] == 0b000000 selects TBU[0] TCR[wpext[0–3],TCR[wp][0–1] == 0b111111 selects TBL[31]
2–3 (34–35)	WRC	Watchdog Timer Reset Control 00 No watchdog timer reset will occur 01 Assert watchdog reset status output 1 (*p_wrs*[1]) on second timeout of watchdog timer 10 Assert watchdog reset status output 0 (*p_wrs*[0]) on second timeout of watchdog timer 11 Assert watchdog reset status outputs 0 and 1 (*p_wrs*[0], *p_wrs*[1]) on second timeout of watchdog timer TCR[WRC] resets to 0b00. This field may be set by software, but cannot be cleared by software (except by a software-induced reset). Once written to a non-zero value, this field may no longer be altered by software.
4 (36)	WIE	Watchdog Timer Interrupt Enable
5 (37)	DIE	Decrementer Interrupt Enable
6–7 (38–39)	FP	Fixed-Interval Timer Period. When concatenated with FPEXT, specifies 1 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][0–3],TCR[fp][0–1] == 0b000000 selects TBU[0] TCR[fpext][0–3],TCR[fp][0–1] == 0b111111 selects TBL[31]
8 (40)	FIE	Fixed-Interval Timer Interrupt Enable
9 (41)	ARE	Auto-Reload Enable
10 (42)	—	Reserved[1]
11–14 (43–46)	WPEXT	Watchdog Timer Period Extension (see above description for WP). These bits get prepended to the TCR[WP] bits to allow selection of 1 of the 64 time base bits used to signal a watchdog timer exception. tb[0–63] ← TBU[0–31] || TBL[0–31] wp ← TCR[WPEXT] || TCR[WP] tb_wp_bit ← tb[wp]
15–18 (47–50)	FPEXT	Fixed-Interval Timer Period Extension (see above description for FP). These bits get prepended to the TCR[FP] bits to allow selection of 1 of the 64 time base bits used to signal a fixed-interval timer exception. tb[0–63] ← TBU[0–31] || TBL[0–31] fp ← TCR[FPEXT] || TCR[FP] tb_fp_bit ← tb[fp]
19–31 (51–63)	—	Reserved[1]

[1] These bits are not implemented and should be written with zero for future compatibility.

2.4.9 Timer Status Register (TSR)

The timer status register (TSR) provides status information for the CPU timer facilities. A complete description of the TSR can be found in the *EREF*. TSR[WRS] is defined to be implementation dependent and is described below. The TSR is shown in Figure 2-13.

SPR 336 Access: w1c

	0	1	2 3	4	5	6 … 31
R	ENW	WIS	WRS	DIS	FIS	—
W	w1c	w1c	w1c	w1c	w1c	
Reset	All zeros					

Figure 2-13. Timer Status Register (TSR)

The TSR fields are defined in Table 2-9.

Table 2-9. Timer Status Register Field Descriptions

Bits	Name	Description
0 (32)	ENW	Enable Next Watchdog
1 (33)	WIS	Watchdog Timer Interrupt Status
2–3 (34–35)	WRS	Watchdog Timer Reset Status 00 No second time-out of watchdog timer has occurred. 01 Assertion of watchdog reset status output 1 (*p_wrs*[1]) on second timeout of watchdog timer has occurred. 10 Assertion of watchdog reset status output 0 (*p_wrs*[0]) on second timeout of watchdog timer has occurred. 11 Assertion of watchdog reset status outputs 0 and 1 (*p_wrs*[0], *p_wrs*[1]) on second timeout of watchdog timer has occurred.
4 (36)	DIS	Decrementer Interrupt Status
5 (37)	FIS	Fixed-Interval Timer Interrupt Status
6–31 (38–63)	—	Reserved

NOTE

The timer status register can be read using **mfspr** RT,TSR. The timer status register cannot be directly written to. Instead, bits in the timer status register corresponding to 1 bit in GPR[RS] can be cleared using **mtspr** TSR,RS.

2.4.10 Debug Registers

The debug facility registers are described in Chapter 13, “Debug Support.”

2.4.11 Hardware Implementation Dependent Register 0 (HID0)

The HID0 register, shown in Figure 2-14, is an e200-implementation dependent register used for various configuration and control functions.

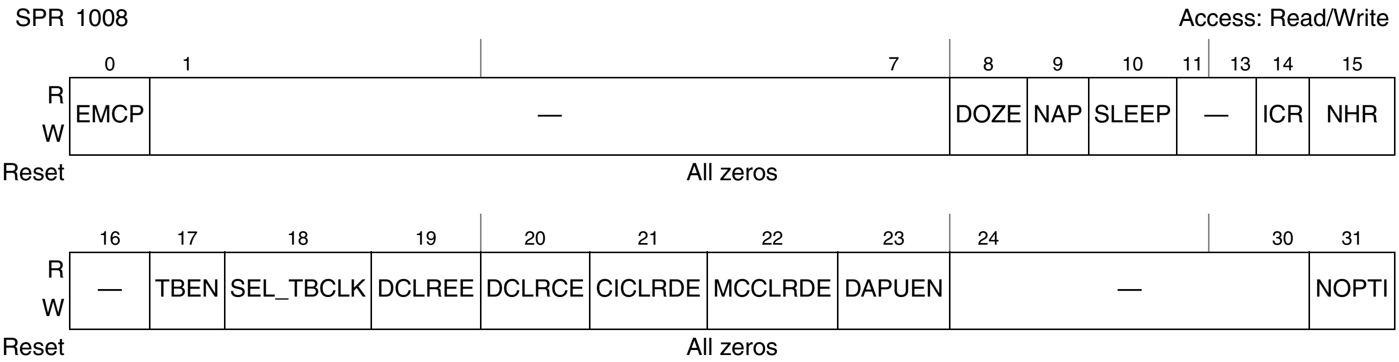


Figure 2-14. Hardware Implementation Dependent Register 0 (HID0)

The HID0 fields are defined in Table 2-10.

Table 2-10. Hardware Implementation Dependent Register 0

Bits	Name	Description
0 [32]	EMCP	Enable Machine Check Pin (*p_mcp_b*) 0 *p_mcp_b* pin is disabled 1 *p_mcp_b* pin is enabled. Asserting *p_mcp_b* causes a machine check interrupt to be reported.
1–7 [33–39]	—	Reserved
8 [40]	DOZE	Configure for Doze Power Management Mode 0 Doze mode is disabled 1 Doze mode is enabled Doze mode is invoked by setting MSR[WE] while this bit is set.
9 [41]	NAP	Configure for Nap Power Management Mode 0 Nap mode is disabled 1 Nap mode is enabled Nap mode is invoked by setting MSR[WE] while this bit is set.
10 [42]	SLEEP	Configure for Sleep Power Management Mode 0 Sleep mode is disabled 1 Sleep mode is enabled Sleep mode is invoked by setting MSR[WE] while this bit is set. Only one of DOZE, NAP, or SLEEP should be set for proper operation.
11–13 [43–45]	—	Reserved
14 [46]	ICR	Interrupt Inputs Clear Reservation 0 External input, critical input, and nonmaskable Interrupts do not affect reservation status 1 External input, critical input, and nonmaskable interrupts clear an outstanding reservation
15 [47]	NHR	Not Hardware Reset 0 indicates to a reset exception handler that a reset occurred if software had previously set this bit. 1 indicates to a reset exception handler that no reset occurred if software had previously set this bit. Provided for software use—set anytime by software, cleared by reset.

Table 2-10. Hardware Implementation Dependent Register 0 (continued)

Bits	Name	Description
16 [48]	—	Reserved
17 [49]	TBEN	Time Base Enable 0 Time base is disabled 1 Time base is enabled
18 [50]	SEL_TBCLK	Select Time Base Clock 0 Time base is based on processor clock 1 Time base is based on *p_tbclk* input This bit controls the clock source for the time base. Altering this bit must be done while the time base is disabled to preclude glitching of the counter. Timer interrupts should be disabled prior to alteration, and the TBL and TBU registers re-initialized following a change of time base clock source.
19 [51]	DCLREE	Debug Interrupt Clears MSR[EE] 0 MSR[EE] unaffected by debug interrupt 1 MSR[EE] cleared by debug interrupt This bit controls whether debug interrupts force external input interrupts to be disabled, or whether they remain unaffected.
20 [52]	DCLRCE	Debug Interrupt Clears MSR[CE] 0 MSR[CE] unaffected by debug interrupt 1 MSR[CE] cleared by debug interrupt This bit controls whether debug interrupts force critical interrupts to be disabled, or whether they remain unaffected.
21 [53]	CICLRDE	Critical Interrupt Clears MSR[DE] 0 MSR[DE] unaffected by critical class interrupt 1 MSR[DE] cleared by critical class interrupt This bit controls whether certain critical interrupts (critical input, watchdog timer) force debug interrupts to be disabled, or whether they remain unaffected. Machine check interrupts have a separate control bit. Note that if critical interrupt debug events are enabled (DBCR0[CIRPT] set, which should only be done when the debug functionality is enabled), and MSR[DE] is set at the time of a (critical input, watchdog timer) critical interrupt, a debug event will be generated after the critical interrupt handler has been fetched, and the debug handler will be executed first. In this case, DSRR0[DE] will have been cleared, such that after returning from the debug handler, the critical interrupt handler will not be run with MSR[DE] enabled.
22 [54]	MCCLRDE	Machine Check Interrupt Clears MSR[DE] 0 MSR[DE] unaffected by machine check interrupt 1 MSR[DE] cleared by machine check interrupt This bit controls whether machine check interrupts force debug interrupts to be disabled, or whether they remain unaffected.

Table 2-10. Hardware Implementation Dependent Register 0 (continued)

Bits	Name	Description
23 [55]	DAPUEN	Debug functionality Enable 0 Debug functionality disabled 1 Debug functionality enabled This bit controls whether the debug functionality is enabled. When enabled, debug interrupts use the DSRR0/DSRR1 registers for saving state, and the **rfdi** instruction is available for returning from a debug interrupt. When disabled, debug Interrupts use the critical interrupt resources CSRR0/CSRR1 for saving state, the **rfci** instruction is used for returning from a debug interrupt, and the **rfdi** instruction is treated as an illegal instruction. When disabled, the settings of the DCLREE, DCLRCE, CICLRDE, and MCCLRDE bits are ignored and are assumed to be ones. Read and write access to DSRR0/DSRR1 via the **mfspr** and **mtspr** instructions is not affected by this bit.
24 [56]	—	Reserved
25–30 [58–62]	—	Reserved
31 [63]	NOPTI	No-Op Touch Instructions 0 **icbt**, **dcbt**, **dcbtst** instructions operate normally 1 **icbt**, **dcbt**, **dcbtst** instructions are no-oped This bit only affects the **icbt**, **dcbt**, and **dcbtst** instructions.

2.4.12 Hardware Implementation Dependent Register 1 (HID1)

The HID1 register is used for bus configuration and system control. HID1 is shown in Figure 2-15.

SPR 1009 Access: Read/Write

	0–15	16–23	24	25–31
R W	—	SYSCTL	ATS	—
Reset	All zeros			

Figure 2-15. Hardware Implementation Dependent Register 1 (HID1)

The HID1 fields are defined in Table 2-11.

Table 2-11. Hardware Implementation Dependent Register 1

Bits	Name	Description
0–15 [32–47]	—	Reserved
16–23 [48–56]	SYSCTL	System Control. These bits are reflected on the outputs of the *p_hid1_sysctl*[0:7] output signals for use in controlling the system. They may need external synchronization.
24 [56]	ATS	Atomic status (read-only). Indicates state of the reservation bit in the load/store unit. See Section 3.5, “Memory Synchronization and Reservation Instructions,” for more detail.
25–31 [57–63]	—	Reserved

2.4.13 Branch Unit Control and Status Register (BUCSR)

The BUCSR register is used for general control and status of the branch target buffer (BTB). BUCSR is shown in Figure 2-16.

SPR 1013

Access: Read/Write

	0 21	22	23 25	26 27	28	29 30	31
R W	—	BBFI	—	BALLOC	—	BPRED	BPEN
Reset	All zeros						

Figure 2-16. Branch Unit Control and Status Register (BUCSR)

The BUCSR fields are defined in Table 2-12.

Table 2-12. Branch Unit Control and Status Register

Bits	Name	Description
0–21 [32–53]	—	Reserved
22 [54]	BBFI	Branch Target Buffer Flash Invalidate. When written to a one, BBFI flash clears the valid bit of all entries in the branch buffer; clearing occurs regardless of the value of the enable bit (BPEN). **Note:** BBFI is always read as zero.
23–25 [55–57]	—	Reserved
26–27 [58–59]	BALLOC	Branch Target Buffer Allocation Control 00 Branch target buffer allocation for all branches is enabled. 01 Branch target buffer allocation is disabled for backward branches. 10 Branch target buffer allocation is disabled for forward branches. 11 Branch target buffer allocation is disabled for both branch directions. This field controls BTB allocation for branch acceleration when BPEN = 1. Note that BTB hits are not affected by the settings of this field. Note that for branches with AA = 1, the MSB of the displacement field is still used to indicate forward/backward, even though the branch is absolute.
28 [60]	—	Reserved

Table 2-12. Branch Unit Control and Status Register

Bits	Name	Description
29–30 [61–62]	BPRED	Branch Prediction Control (Static) 00 Branch predicted taken on BTB miss for all branches. 01 Branch predicted taken on BTB miss only for forward branches. 10 Branch predicted taken on BTB miss only for backward branches. 11 Branch predicted not taken on BTB miss for both branch directions. This field controls operation of static prediction mechanism on a BTB miss. Unless disabled, fetching of the predicted target location will be performed for branch acceleration. BPRED operates independently of BPEN, and with a BPEN setting of 0, will be used to perform static prediction of all unresolved branches. Note that BTB hits are not affected by the settings of this field. Note that for certain applications, setting BPRED to a non-default value may result in improved performance.
31 [63]	BPEN	Branch Target Buffer Prediction Enable. 0 Branch target buffer prediction disabled 1 Branch target buffer prediction enabled (enables BTB to predict branches) When the BPEN bit is cleared, no hits will be generated from the BTB, and no new entries will be allocated. Entries are not automatically invalidated when BPEN is cleared; the BBFI bit controls entry invalidation. BPEN operates independently of BPRED, and will be used even with a BPRED setting of 00.

2.4.14 L1 Cache Control and Status Registers (L1CSR0, L1CSR1)

The L1CSR0 and L1CSR1 registers are used for general control and status of the L1 caches. A description of the L1CSR0 and L1CSR1 registers can be found in Chapter 9, “L1 Cache.”

2.4.15 L1 Cache Configuration Registers (L1CFG0, L1CFG1)

The L1CFG0 and L1CGF1 registers provide configuration information for the L1 caches supplied with this version of the e200 CPU core. A description of the L1CFG0 and L1CGF1 registers can be found in Chapter 9, “L1 Cache.”

2.4.16 L1 Cache Flush and Invalidate Registers (L1FINV0, L1FINV1)

The L1FINV0 and L1FINV1 registers provide software-based flush and invalidation control for the L1 caches supplied with this version of the e200 CPU core. A description of the L1FINV0 and L1FINV1 registers can be found in Chapter 9, “L1 Cache.”

2.4.17 MMU Control and Status Register (MMUCSR0)

The MMUCSR0 register is used for general control of the MMU. A description of the MMUCSR register can be found in Chapter 10, “Memory Management Unit.”

2.4.18 MMU Configuration Register (MMUCFG)

The MMUCFG register provides configuration information for the MMU supplied with this version of the e200 CPU core. A description of the MMUCFG register can be found in Chapter 10, “Memory Management Unit.”

2.4.19 TLB Configuration Registers (TLB0CFG, TLB1CFG)

The TLB0CFG and TLB1CFG registers provide configuration information for the MMU TLBs supplied with this version of the e200 CPU core. A description of these registers can be found in Chapter 10, "Memory Management Unit."

2.5 SPR Register Access

SPRs are accessed with the **mfspr** and **mtspr** instructions. The following sections outline additional access requirements.

2.5.1 Invalid SPR References

System behavior when an invalid SPR is referenced depends on the apparent privilege level of the register (refer to Table 2-13). The register privilege level is determined by bit 5 in the SPR address. If the invalid SPR is accessible in user mode, then an illegal exception is generated. If the invalid SPR is accessible only in supervisor mode and the CPU core is in supervisor mode (MSR[PR] = 0), then an illegal exception is generated. If the invalid SPR address is accessible only in supervisor mode and the CPU is not in supervisor mode (MSR[PR] = 1), then a privilege exception is generated.

Note that writes to read-only SPRs and reads of write-only SPRs are treated as invalid SPR references.

Table 2-13. System Response to Invalid SPR Reference

SPR Address Bit 5	Mode	MSR_{PR}	Response
0	—	—	Illegal exception
1	Supervisor	0	Illegal exception
1	User	1	Privilege exception

2.5.2 Synchronization Requirements for SPRs

With the exception of the following registers, there are no synchronization requirements for accessing SPRs beyond those stated in the Power ISA embedded architecture. A complete description of synchronization requirements are contained in the *EREF*. Software requirements for synchronization before/after accessing these registers are shown in Table 2-14. The notation CSI in the table refers to a context synchronizing instruction which include **sc**, **isync**, **rfi**, **rfci**, and **rfdi**.

Table 2-14. Additional synchronization requirements for SPRs

Context Altering Event or Instruction	Required Before	Required After	Notes
mtmsr[UCLE]	none	CSI	—
mtmsr[SPE]	none	CSI	—
mtmsr[PMM]	none	CSI	—
mfspr			

Table 2-14. Additional synchronization requirements for SPRs (continued)

Context Altering Event or Instruction		Required Before	Required After	Notes
DBCNT	Debug Counter register	**msync**	none	1
DBSR	Debug Status register	**msync**	none	—
HID0	Hardware implementation dependent reg 0	none	none	—
HID1	Hardware implementation dependent reg 1	**msync**	none	—
L1CSR0, L1CSR1	L1 cache control and status registers 0,1	**msync**	none	—
L1FINV0, L1FINV1	L1 cache flush and invalidate control registers 0,1	**msync**	none	—
MMUCSR	MMU control and status register 0	CSI	none	—
mtspr				
BUCSR	Branch Unit Control and Status Register	none	CSI	—
DBCNT	Debug Counter register	none	CSI	1
DBCR0–6	Debug Control Register 0–6	none	CSI	—
DBSR	Debug Status Register	**msync**	none	—
HID0	Hardware implementation dependent reg 0	CSI	isync	—
HID1	Hardware implementation dependent reg 1	**msync, isync**	CSI	—
L1CSR0	L1 cache control and status register 0	**msync, isync**	CSI	—
L1CSR1	L1 cache control and status registers 1	none	CSI	—
L1FINV0, L1FINV1	L1 cache flush and invalidate control registers 0,1	**msync**	CSI	—
MASx	MMU MAS registers	none	CSI	—
MMUCSR	MMU control and status register 0	CSI	CSI	—
PID	PID0 register	none	CSI	—
SPEFSCR	SPEFSCR register	none	CSI[2]	—

Notes:
1. not required if counter is not currently enabled
2. not required for status bit clearing, required for altering exception enable or rounding mode bits

2.5.3 Special Purpose Register Summary

Power ISA embedded category and implementation-specific SPRs for the e200 core are listed in Table 2-15. All registers are 32 bits in size. Register bits are numbered from bit 0 to bit 31 (most-significant to least-significant). Shaded entries represent optional registers. An SPR register may be read or written with the **mfspr** and **mtspr** instructions. In the instruction syntax, compilers should recognize the mnemonic name given in the table below.

Table 2-15. Special Purpose Registers

Mnemonic	Name	SPR Number	Access	Privileged	e200 Specific
BUCSR	Branch unit control and status register	1013	R/W	Yes	Yes
CSRR0	Critical save/restore register 0	58	R/W	Yes	No
CSRR1	Critical save/restore register 1	59	R/W	Yes	No
CTR	Count register	9	R/W	No	No
DAC1	Data address compare 1	316	R/W	Yes	No
DAC2	Data address compare 2	317	R/W	Yes	No
DBCNT	Debug counter register	562	R/W	Yes	Yes
DBCR0	Debug control register 0	308	R/W	Yes	No
DBCR1	Debug control register 1	309	R/W	Yes	No
DBCR2	Debug control register 2	310	R/W	Yes	No
DBCR3	Debug control register 3	561	R/W	Yes	Yes
DBCR4	Debug control register 4	563	R/W	Yes	Yes
DBCR5	Debug control register 5	564	R/W	Yes	Yes
DBCR6	Debug control register 5	603	R/W	Yes	Yes
DBERC0	Debug external resource control register 0	569	Read-only	Yes	Yes
DBSR	Debug status register	304	Read/Clear[1]	Yes	No
DDAM	Debug data acquisition messaging register	576	R/W	No	Yes
DEAR	Data exception address register	61	R/W	Yes	No
DEC	Decrementer	22	R/W	Yes	No
DECAR	Decrementer auto-reload	54	R/W	Yes	No
DEVENT	Debug event register	975	R/W	No	Yes
DSRR0	Debug save/restore register 0	574	R/W	Yes	Yes
DSRR1	Debug save/restore register 1	575	R/W	Yes	Yes
DVC1	Data value compare 1	318	R/W	Yes	No
DVC2	Data value compare 2	319	R/W	Yes	No
ESR	Exception syndrome register	62	R/W	Yes	No
HID0	Hardware implementation dependent reg 0	1008	R/W	Yes	Yes
HID1	Hardware implementation dependent reg 1	1009	R/W	Yes	Yes
IAC1	Instruction address compare 1	312	R/W	Yes	No
IAC2	Instruction address compare 2	313	R/W	Yes	No
IAC3	Instruction address compare 3	314	R/W	Yes	No
IAC4	Instruction address compare 4	315	R/W	Yes	No

Table 2-15. Special Purpose Registers (continued)

Mnemonic	Name	SPR Number	Access	Privileged	e200 Specific
IAC5	Instruction address compare 5	565	R/W	Yes	Yes
IAC6	Instruction address compare 6	566	R/W	Yes	Yes
IAC7	Instruction address compare 7	567	R/W	Yes	Yes
IAC8	Instruction address compare 8	568	R/W	Yes	Yes
IVOR0	Interrupt vector offset register 0	400	R/W	Yes	No
IVOR1	Interrupt vector offset register 1	401	R/W	Yes	No
IVOR2	Interrupt vector offset register 2	402	R/W	Yes	No
IVOR3	Interrupt vector offset register 3	403	R/W	Yes	No
IVOR4	Interrupt vector offset register 4	404	R/W	Yes	No
IVOR5	Interrupt vector offset register 5	405	R/W	Yes	No
IVOR6	Interrupt vector offset register 6	406	R/W	Yes	No
IVOR7	Interrupt vector offset register 7	407	R/W	Yes	No
IVOR8	Interrupt vector offset register 8	408	R/W	Yes	No
IVOR9	Interrupt vector offset register 9	409	R/W	Yes	No
IVOR10	Interrupt vector offset register 10	410	R/W	Yes	No
IVOR11	Interrupt vector offset register 11	411	R/W	Yes	No
IVOR12	Interrupt vector offset register 12	412	R/W	Yes	No
IVOR13	Interrupt vector offset register 13	413	R/W	Yes	No
IVOR14	Interrupt vector offset register 14	414	R/W	Yes	No
IVOR15	Interrupt vector offset register 15	415	R/W	Yes	No
IVOR32	Interrupt vector offset register 32	528	R/W	Yes	Yes
IVOR33	Interrupt vector offset register 33	529	R/W	Yes	Yes
IVOR34	Interrupt vector offset register 34	530	R/W	Yes	Yes
IVOR35	Interrupt vector offset register 35	531	R/W	Yes	Yes
IVPR	Interrupt vector prefix register	63	R/W	Yes	No
LR	Link register	8	R/W	No	No
L1CFG0	L1 cache config register 0	515	Read-only	No	Yes
L1CFG1	L1 cache config register 1	516	Read-only	No	Yes
L1CSR0	L1 cache control and status register 0	1010	R/W	Yes	Yes
L1CSR1	L1 cache control and status register 1	1011	R/W	Yes	Yes
L1FINV0	L1 cache flush and invalidate control register 0	1016	R/W	Yes	Yes
L1FINV1	L1 cache flush and invalidate control register 0	959	R/W	Yes	Yes

Table 2-15. Special Purpose Registers (continued)

Mnemonic	Name	SPR Number	Access	Privileged	e200 Specific
MAS0	MMU assist register 0	624	R/W	Yes	Yes
MAS1	MMU assist register 1	625	R/W	Yes	Yes
MAS2	MMU assist register 2	626	R/W	Yes	Yes
MAS3	MMU assist register 3	627	R/W	Yes	Yes
MAS4	MMU assist register 4	628	R/W	Yes	Yes
MAS6	MMU assist register 6	630	R/W	Yes	Yes
MCAR	Machine check address register	573	R/W	Yes	Yes
MCSR	Machine check syndrome register	572	R/Clear[2]	Yes	Yes
MCSRR0	Machine check save/restore register 0	570	R/W	Yes	Yes
MCSRR1	Machine check save/restore register 1	571	R/W	Yes	Yes
MMUCFG	MMU configuration register	1015	Read-only	Yes	Yes
MMUCSR	MMU control and status register 0	1012	R/W	Yes	Yes
PID0	Process ID register	48	R/W	Yes	No
PIR	Processor ID register	286	R/W	Yes	No
PVR	Processor version register	287	Read-only	Yes	No
SPEFSCR	SPE status and control register	512	R/W	No	No
SPRG0	SPR general 0	272	R/W	Yes	No
SPRG1	SPR general 1	273	R/W	Yes	No
SPRG2	SPR general 2	274	R/W	Yes	No
SPRG3	SPR general 3	275	R/W	Yes	No
SPRG4	SPR general 4	260	Read-only	No	No
		276	R/W	Yes	No
SPRG5	SPR general 5	261	Read-only	No	No
		277	R/W	Yes	No
SPRG6	SPR general 6	262	Read-only	No	No
		278	R/W	Yes	No
SPRG7	SPR general 7	263	Read-only	No	No
		279	R/W	Yes	No
SPRG8	SPR general 8	604	R/W	Yes	Yes
SPRG9	SPR general 9	605	R/W	Yes	Yes
SRR0	Save/restore register 0	26	R/W	Yes	No
SRR1	Save/restore register 1	27	R/W	Yes	No

Table 2-15. Special Purpose Registers (continued)

Mnemonic	Name	SPR Number	Access	Privileged	e200 Specific
SVR	System version register	1023	Read-only	Yes	Yes
TBL	Time base lower	268	Read-only	No	No
		284	Write-only	Yes	No
TBU	Time base upper	269	Read-only	No	No
		285	Write-only	Yes	No
TCR	Timer control register	340	R/W	Yes	No
TLB0CFG	TLB0 configuration register	688	Read-only	Yes	Yes
TLB1CFG	TLB1 configuration register	689	Read-only	Yes	Yes
TSR	Timer status register	336	Read/Clear[3]	Yes	No
USPRG0	User SPR general 0	256	R/W	No	No
XER	Integer exception register	1	R/W	No	No

Note:

[1] The Debug Status Register can be read using *mfspr RT,DBSR*. The Debug Status Register cannot be directly written to. Instead, bits in the Debug Status Register corresponding to ‘1’ bits in GPR(RS) can be cleared using *mtspr DBSR,RS*.

[2] The Machine Check Syndrome Register can be read using *mfspr RT,MCSR*. The Machine Check Syndrome Register cannot be directly written to. Instead, bits in the Machine Check Syndrome Register corresponding to ‘1’ bits in GPR(RS) can be cleared using *mtspr MCSR,RS*.

[3] The Timer Status Register can be read using *mfspr RT,TSR*. The Timer Status Register cannot be directly written to. Instead, bits in the Timer Status Register corresponding to ‘1’ bits in GPR(RS) can be cleared using *mtspr TSR,RS*.

2.6 Reset Settings

Table 2-16 shows the state of the Power ISA embedded category architecture registers and other optional resources immediately following a system reset.

Table 2-16. Reset Settings for e200 Resources

Resource	System Reset Setting
Program Counter	p_rstbase[0:29] || 0b00
GPR	Unaffected[1]
CR	Unaffected[1]
BUCSR	All zeros
CSRR0	Unaffected[1]
CSRR1	Unaffected[1]
CTR	Unaffected[1]
DAC1	All zeros[2]
DAC2	All zeros[2]

Table 2-16. Reset Settings for e200 Resources (continued)

Resource	System Reset Setting
DBCNT	Unaffected[1]
DBCR0	All zeros[2]
DBCR1	All zeros[2]
DBCR2	All zeros[2]
DBCR3	All zeros[2]
DBCR4	All zeros[2]
DBCR5	All zeros[2]
DBCR6	All zeros[2]
DBSR	0x1000_0000[2]
DDAM	All zeros[2]
DEAR	Unaffected[1]
DEC	Unaffected[1]
DECAR	Unaffected[1]
DEVENT	All zeros[2]
DSRR0	Unaffected[1]
DSRR1	Unaffected[1]
DVC1	Unaffected[1]
DVC2	Unaffected[1]
ESR	All zeros
HID0	All zeros
HID1	All zeros
IAC1	All zeros[2]
IAC2	All zeros[2]
IAC3	All zeros[2]
IAC4	All zeros[2]
IAC5	All zeros[2]
IAC6	All zeros[2]
IAC7	All zeros[2]
IAC8	All zeros[2]
IVORxx	Unaffected[1]
IVPR	Unaffected[1]
LR	Unaffected[1]
L1CFG0, L1CFG1[3]	—

Table 2-16. Reset Settings for e200 Resources (continued)

Resource	System Reset Setting
L1CSR0, 1	All zeros
L1FINV0, 1	All zeros
MAS0	Unaffected[1]
MAS1	Unaffected[1]
MAS2	Unaffected[1]
MAS3	Unaffected[1]
MAS4	Unaffected[1]
MAS6	Unaffected[1]
MCAR	Unaffected[1]
MCSR	All zeros
MCSRR0	Unaffected[1]
MCSRR1	Unaffected[1]
MMUCFG[3]	—
MSR	All zeros
PID0	All zeros
PIR	0x00_0000 ǁ p_cpuid[0:7]
PVR[3]	—
SPEFSCR	All zeros
SPRG0	Unaffected[1]
SPRG1	Unaffected[1]
SPRG2	Unaffected[1]
SPRG3	Unaffected[1]
SPRG4	Unaffected[1]
SPRG5	Unaffected[1]
SPRG6	Unaffected[1]
SPRG7	Unaffected[1]
SPRG8	Unaffected[1]
SPRG9	Unaffected[1]
SRR0	Unaffected[1]
SRR1	Unaffected[1]
SVR[3]	—
TBL	Unaffected[1]
TBU	Unaffected[1]

Table 2-16. Reset Settings for e200 Resources (continued)

Resource	System Reset Setting
TCR	All zeros
TSR	All zeros
TLB0CFG[3]	—
TLB1CFG[3]	—
USPRG0	Unaffected[1]
XER	All zeros

[1] Undefined on **m_por** assertion, unchanged on **p_reset_b** assertion

[2] Reset by processor reset **p_reset_b** if DBCR0[EDM] = 0, as well as unconditionally by **m_por**.

[3] Read-only registers

Chapter 3
Instruction Model

This chapter provides additional information about the Power ISA embedded category architecture as it relates specifically to the e200z760n3.

The e200z7 is a 32-bit implementation of the Power ISA embedded category architecture as described in the *EREF*. However, different processor implementations may require clarifications, extensions, or deviations from the architectural descriptions. See the processor-specific reference manuals for details about deviations.

3.1 Unsupported Instructions and Instruction Forms

Because the e200z7 is a 32-bit Power ISA embedded category core, all of the instructions defined for 64-bit implementations of the Power ISA architecture are illegal on the e200. See the *EREF* for more information on 64-bit instructions. The e200 takes an illegal instruction exception type program interrupt upon encountering a 64-bit Power ISA instruction.

Besides the 64-bit instructions, there are other Power ISA embedded category instructions not supported by the e200z7. If one of these instructions is executed on the e200z7, an unimplemented operation or FP (floating-point) unavailable exception is generated.

3.2 Implementation Specific Instructions

Several Power ISA embedded category instructions are implementation-specific. Table 3-1 summarizes these e200 implementation-specific instructions.

Table 3-1. Implementation-Specific Instruction Summary

Mnemonic	Implementation Details
mfapidi	Unimplemented instructions
mfdcrx	
mtdcrx	
stbcx., **sthcx.**, **stwcx.**	Address match with prior **lbarx**, **lharx**, or **lwarx** not required for store to be performed
mfdcr, **mtdcr**[1]	Optionally supported instructions

[1] The e200 CPU will take an illegal instruction exception for unsupported DCR values

3.3 Power ISA Embedded Category Instruction Extensions

This section describes how certain Power ISA embedded category instructions support the Power ISA VLE functionality:

- **rfci**, **rfdi**, **rfi**, **rfmci**—No longer mask bit 62 of CSRR0, DSRR0, or SRR0, respectively. The destination address is [D,C,MC]SRR0[32:62] || 0b0.
- **bclr**, **bclrl**, **bcctr**, **bcctrl**—No longer mask bit 62 of the LR or CTR, respectively. The destination address is [LR,CTR][32:62] || 0b0.

3.4 Memory Access Alignment Support

The e200 core provides hardware support for unaligned memory accesses; however, there is a performance degradation for accesses which cross a 64-bit (8-byte) boundary. For loads that hit in the cache, the throughput of the load/store unit is degraded to 1 misaligned load every 2 cycles. Stores which are misaligned across a 64-bit (8-byte) boundary can be translated at a rate of 2 cycles per store. Frequent use of unaligned memory accesses is discouraged because of the impact on performance.

NOTE

> Accesses which cross a translation boundary may be restarted. A misaligned access which crosses a page boundary is restarted in its entirety in the event of a TLB miss of the second portion of the access. 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.

3.5 Memory Synchronization and Reservation Instructions

The **msync** instruction provides a synchronization function and a memory barrier function. This instruction waits for all preceding instructions and data memory accesses to complete before the **msync** instruction completes. Subsequent instructions in the instruction stream are not initiated until after the **msync** instruction completes to ensure these functions have been performed.

In addition, the **msync** instructions and the **mbar** w/MO = 0 or 1 instructions handshake with the system to ensure that all accesses initiated by this CPU have been “performed” with respect to all other processors and mechanisms prior to completion of the instruction.

On the e200 core, the **mbar** instruction with MO = 0, 1, or 1 behaves similarly to the **msync** instruction, but only waits for previous data memory accesses rather than all previous instructions to complete before completing. The **mbar** instruction with MO = 2 behaves similarly to the **msync** instruction, but only waits for previous data memory accesses rather than all previous instructions to complete before completing, and does not signal synchronizations to other processors through the synchronization port. The **mbar** instruction may be preferred for most memory synchronization operations, since it does not stall instruction execution if no load or store operations remain in the execution pipeline, unlike the **msync** instruction. The **mbar** instruction with the MO field not equal to 0, 1, or 2 is treated as illegal by the e200 core.

The e200 core implements the **lwarx** and **stwcx.** instructions as described in the Power ISA embedded category, as well as the **lharx**, **lbarx**, **sthcx.**, and **stbcx.** instructions defined by the EIS enhanced reservation functionality. If the EA is not a multiple of the access size for these instructions, an alignment interrupt is invoked. The e200 allows reservation instructions to access a page that is marked as write-through required or cache-inhibited, and no data storage interrupt is invoked.

As allowed by the Power ISA embedded category, the e200 core does not require that the EA of the store-type instruction must be to the same reservation granule as the EA of a preceding reservation load-type instruction for a reservation store-type instruction to succeed. Reservation granularity is implementation dependent. The e200 core does not define a reservation granule explicitly; reservation granularity is defined by external logic. When no external logic is provided, the e200 core performs no address comparison checking, thus the effective implementation granularity is null.

The e200 core implements an internal status flag (HID1[ATS]) representing reservation status. This flag is set when a load-type reservation instruction is executed and completes without error, and remains set until it is cleared by one of the following mechanisms:

- Execution of a store-type reservation instruction is completed without error.
- The e200 core *p_rsrv_clr* input signal is asserted.
- The reservation is invalidated when an external input, critical input, or nonmaskable interrupt is signaled and HID0[ICR] is set.

When the e200 core decodes a store-type reservation instruction, it checks the value of the local reservation flag (HID1[ATS]). If the status indicates that no reservation is active, the store-type reservation instruction is treated as a no-op. No exceptions are taken and no access is performed; thus no data breakpoint occurs, regardless of matching the data breakpoint attributes.

The e200 core treats reservation accesses as though they were both cache inhibited and guarded, regardless of page attributes. A cache line corresponding to the address of a reservation access is flushed to memory if dirty, and then invalidated, prior to the reservation access being issued to the bus. This is done to allow external reservation logic to be built, which properly signals a reservation failure.

The e200 core provides the input signal *p_xfail_b*, which is sampled at termination of a **st[b,h,w]cx.** store transfer to allow an external agent or mechanism to indicate that the **st[b,h,w]cx.** instruction has failed to update memory, even though a reservation existed for the store at the time it was issued. This is not considered an error and causes the condition codes for the **st[b,h,w]cx.** instruction to be written as if a reservation did not exist for the **st[b,h,w]cx.** instruction. In addition, any outstanding reservation is cleared.

The *p_rsrv_clr* input signal is not intended for normal use in managing reservations. It is provided for specialized system applications. The normal bus protocol is used to manage reservations using external reservation logic in systems with multiple coherent bus masters, using the transfer type and transfer response signals. In single coherent master systems, no external logic is required, and the internal reservation flag is sufficient to support multitasking applications.

3.6 Branch Prediction

The e200z7 instruction fetching mechanism uses a branch target buffer (BTB) that holds branch target addresses combined with a 2-bit saturating up-down counter scheme for branch prediction. Branch paths are predicted by either the branch target buffer (BTB hit) or a selectable static prediction algorithm (BTB miss) and subsequently checked to see if the prediction was correct. This enables operation beyond a conditional branch without waiting for the branch to be decoded and resolved. The instruction fetch unit predicts the direction of the branch as follows:

- Predict taken for any backward branch whose fetch address hits in the BTB and is predicted taken by the counter or misses in the BTB and static prediction control in BUCSR for backward branches indicates ‘predict taken’. Otherwise, predict not-taken.
- Predict taken for any forward branch whose fetch address hits in the BTB and is predicted taken by the counter or misses in the BTB and static prediction control in BUCSR for forward branches indicates ‘predict taken’. Otherwise, predict not-taken.

3.7 Interruption of Instructions by Interrupt Requests

In general, the e200z7 core samples pending nonmaskable interrupts, external input, and critical input interrupt requests at instruction boundaries. However, in order to reduce interrupt latency, long running instructions may be interrupted prior to completion. Instructions in this class include divides (**divw[uo][.]**, **efsdiv**, **evfsdiv**, **evdivw[su]**), load multiple word (**lmw, e_lmw**), and store multiple word (**stmw**, **e_stmw**). In addition, the **e_lmvgprw**, **e_stmvgprw**, **e_lmvsprw**, and **e_stmvsprw** Volatile Context Save/Restore functionality instructions may also be interrupted prior to completion. When interrupted prior to completion, the value saved in SRR0/CSRR0/MCSRR0 is the address of the interrupted instruction. The instruction is restarted from the beginning after returning to it from the interrupt handler.

3.8 New e200 Functionality

The e200z7core implements the following functionality that may be new to users migrating from earlier implementations of the e200 core family, and these new categories of functionality are listed here to highlight these new features. Many of these instructions are now part of the Power ISA embedded architecture, while others are currently only implemented as EIS functionality in Freescale processors.

- The Power ISA **isel** instruction described in Section 3.9, “ISEL instruction,” and also in the *EREF*.
- The Power ISA Enhanced Debug Functionality and the Debug Notify Halt instructions described in Section 3.10, “Enhanced Debug,” and also in the *EREF*.
- The Power ISA Machine Check functionality described in Section 3.11, “Machine Check,” and also in the *EREF*.
- The Power ISA **wait** instruction described in Section 3.12, “WAIT Instruction.”
- The volatile context save/restore unit, which is described in Section 3.14, “Volatile Context Save/Restore Unit
- The Power ISA embedded floating-point unit, described along with supporting instructions in Chapter 5, “Embedded Floating-Point Unit.” The EFPU is a subset of the SPE.
- The Power ISA Signal Processing Extension (SPE) version, described along with supporting instructions in Chapter 6, “Signal Processing Extension (SPE).

- The Power ISA performance monitor functionality which is described in Chapter 8, “Performance Monitor and also described in the *EREF*.
- The Power ISA cache line-locking functionality described in Section 9.10, “Cache Management Instructions,” and also in the *EREF*.
- The enhanced reservations functionality described in Section 3.13, “Enhanced Reservations.”

3.9 ISEL instruction

The **isel** instruction provides a means to select one of two registers and place the result in a destination register under the control of a predicate value supplied by a bit in the condition register. This instruction can be used to eliminate branches in software and in many cases improve performance. This instruction can also increase program execution time determinism by eliminating the need to predict the target and direction of the branches replaced by the integer select function. The instruction form and definition is as follows.

isel isel

Integer Select

isel **RT, RA, RB, crb**

31	RT	RA	RB	crb	0 1 1 1 1	0
0 5	6 10	11 15	16 20	21 25	26 30	31

```
if RA=0 then a ← $^{32}0$ else a ← GPR(RA)
c = CR$_{crb}$
if c then GPR(RT) ← a
else GPR(RT) ← GPR(RB)
```

For **isel**, if the bit of the CR specified by (crb) is set, the contents of RA | 0 are copied into RT. If the bit of the CR specified by (crb) is clear, the contents of RB are copied into RT.

Other registers altered:

- None

3.10 Enhanced Debug

The e200z7 implements the Power ISA embedded debug architecture to support the capability to handle the debug interrupt as an additional interrupt level. To support this interrupt level, a new return from debug interrupt (**rfdi**, **se_rfdi**) instruction is defined as part of the debug APU, along with a new pair of save/restore registers, DSRR0 and DSRR1.

When the debug capability is enabled (HID0[DAPUEN] = 1), the **rfdi** or **se_rfdi** instruction provides a means to return from a debug interrupt. See Section 2.4.11, “Hardware Implementation Dependent Register 0 (HID0),” for more information about enabling the debug functionality.

The instruction form and definition is as follows:

rfdi rfdi

Return From Debug Interrupt

rfdi

19	///	0 0 0 0 1 0 0 1 1 1	0
0 5	6 20	21 30	31

```
MSR ←DSRR1
PC ←$DSRR0_{0:30}$ || $^{1}0$
```

The **rfdi** instruction is used to return from a debug interrupt, or as a means of simultaneously establishing a new context and synchronizing on that new context.

The contents of debug save/restore register 1 are placed into the machine state register. If the new machine state register value does not enable any pending exceptions, then the next instruction is fetched, under control of the new machine state register value from the address DSRR0[0–30] || 0b0. If the new machine state register value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into save/restore register 0 or critical save/restore register 0 by the interrupt processing mechanism is the address of the instruction that would have been executed next had the interrupt not occurred (that is, the address in debug save/restore register 0 at the time of the execution of the **rfdi**).

Execution of this instruction is privileged and context synchronizing.

Special registers altered:

- MSR

When the debug unit is disabled (HID0[DAPUEN] = 0), this instruction is treated as an illegal instruction.

se_rfdi se_rfdi

Return From Debug Interrupt

se_rfdi

0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0
0																15

```
MSR ←DSRR1
PC ←$DSRR0_{32:62}$ || 0b0
```

The **rfdi or se_rfdi** instruction is used to return from a debug interrupt or as a means of simultaneously establishing a new context and synchronizing on that new context.

The contents of debug save/restore register 1 are place into the machine state register. If the new machine state register value does not enable any pending exceptions, then the next instruction is fetched, under control of the new machine state register value from the address DSRR0[32–62] || 0b0. If the new machine state register value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into save/restore register 0 or critical save/restore register 0 by the interrupt processing mechanism is the address of the instruction that would have been executed next had the interrupt not occurred (that is, the address in debug save/restore register 0 at the time of the execution of the **rfdi** or **se_rfdi**).

Execution of this instruction is privileged and context synchronizing.

Special registers altered:

- MSR

When the debug unit is disabled (HID0[DAPUEN] = 0), this instruction is treated as an illegal instruction.

3.10.1 Debug Notify Halt Instructions

The **dnh, e_dnh,** and **se_dnh** instructions provide a bridge between the execution of instructions on the core in a non-halted mode, and an external debug facility. **dnh**, **e_dnh**, and **se_dnh** allows software to transition the core from a running state to a debug halted state if enabled by an external debugger, and **dnh** provides the external debugger with bits reserved in the instruction itself to pass additional information. For e200z760n3, when the CPU enters a debug halted state due to a **dnh**, **e_dnh**, or **se_dnh** instruction, the instruction is stored in the CPUSCR[IR] portion. The CPUSCR[PC] value points to the instruction. The external debugger should update the CPUSCR prior to exiting the debug halted state to point past the **dnh**, **e_dnh**, or **se_dnh** instruction.

Note that the **dnh** instruction is only available in Power ISA embedded category instruction pages, and the **e_dnh** and **se_dnh** instructions are only available in VLE instruction pages.

dnh

dnh

Debugger Notify Halt

dnh **dui, duis**

0 5	6 10	11 15 16 20	21 30	31
0 1 0 0 1 1	dui	duis	0 0 1 1 0 0 0 1 1 0	/

```
if $EDBCR_{DNH\_EN}$ = 1 then
    implementation dependent register ← dui
    halt processor
else
    illegal instruction exception
```

Execution of the **dnh** instruction causes the processor to halt if the external debug facility has enabled such action by previously setting EDBCR[DNH_EN]. If the processor is halted, the contents of the dui field are provided to the external debug facility to identify the reason for the halt.

If EDBCR[DNH_EN] has not been previously set by the external debug facility, executing the **dnh** instruction produces an illegal instruction exception.

The duis field is provided to pass additional information about the halt, but requires that actions be performed by the external debug facility to access the **dnh** instruction to read the contents of the field.

The **dnh** instruction is not privileged, and executes the same regardless of the state of MSR[PR].

The current state of the processor debug facility, whether the processor is in IDM or EDM mode has no effect on the execution of the **dnh** instruction.

Other registers altered:

- None.

Software Note: After the **dnh** instruction has executed, the instruction itself can be read back by the Illegal Instruction Interrupt handler or the external debug facility if the contents of the dui and duis field are of interest. If the processor entered the Illegal Instruction Interrupt handler, software can use SRR0 to obtain the address of the **dnh** instruction which caused the handler to be invoked. If the processor is halted in debug mode, the external debug facility can access the CPUSCR register to obtain the **dnh** instruction which caused the processor to halt.

e_dnh e_dnh

Debugger Notify Halt

e_dnh **dui, duis**

0 5	6 10	11 15 16 20	21 30	31
0 1 1 1 1 1	dui	duis	0 0 0 1 1 0 0 0 0 1	/

```
if $EDBCR_{DNH\_EN}$ = 1 then
    implementation dependent register ← dui
    halt processor
else
    illegal instruction exception
```

Execution of the **e_dnh** instruction causes the processor to halt if the external debug facility has enabled such action by previously setting EDBCR[DNH_EN]. If the processor is halted, the contents of the dui field are provided to the external debug facility to identify the reason for the halt.

If EDBCR[DNH_EN] has not been previously set by the external debug facility, executing the **e_dnh** instruction produces an illegal instruction exception.

The duis field is provided to pass additional information about the halt, but requires that actions be performed by the external debug facility to access the **e_dnh** instruction to read the contents of the field.

The **e_dnh** instruction is not privileged, and executes the same regardless of the state of MSR[PR].

The current state of the processor debug facility, whether the processor is in IDM or EDM mode has no effect on the execution of the **e_dnh** instruction.

Other registers altered:

- None

se_dnh se_dnh

Debugger Notify Halt

se_dnh

0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
0																15

```
if $\text{EDBCR}_{\text{DNH\_EN}}$ = 1 then
    halt processor
else
    illegal instruction exception
```

Execution of the **se_dnh** instruction causes the processor to halt if the external debug facility has enabled such action by previously setting EDBCR[DNH_EN].

If EDBCR[DNH_EN] has not been previously set by the external debug facility, executing the **se_dnh** instruction produces an illegal instruction exception.

The **se_dnh** instruction is not privileged, and executes the same regardless of the state of MSR[PR].

The current state of the processor debug facility, whether the processor is in IDM or EDM mode has no effect on the execution of the **se_dnh** instruction.

Other registers altered:

- None.

3.11 Machine Check

The e200z7 implements the Power ISA embedded category machine check functionality to support the capability to handle the machine check interrupt as an additional interrupt level. To support this interrupt level, a new Return From Machine Check Interrupt (**rfmci, se_rfmci**) instruction is defined as part of the machine check capability, along with a new pair of save/restore registers (MCSRR0 and MCSRR1), a machine check syndrome register (MCSR), and a machine check address register (MCAR).

The **rfmci** and **se_rfmci** instructions provide a means to return from a machine check interrupt. The instruction form and definitions is as follows:

rfmci rfmci

Return From Machine Check Interrupt

rfmci

19	///	0 0 0 0 1 0 0 1 1 0	0
0 5	6 20	21 30	31

```
MSR ←MCSRR1
PC ←MCSRR0$_{0:30}$ || $^{1}$0
```

The **rfmci** instruction is used to return from a machine check interrupt, or as a means of simultaneously establishing a new context and synchronizing on that new context.

The contents of machine check save/restore register 1 are place into the machine state register. If the new machine state register value does not enable any pending exceptions, then the next instruction is fetched, under control of the new machine state register value from the address MCSRR0[0:30] || 0b0. If the new machine state register value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into the appropriate save/restore register 0 by the interrupt processing mechanism is the address of the instruction that would have been executed next had the interrupt not occurred (that is, the address in machine check save/restore register 0 at the time of the execution of the **rfmci**).

Execution of this instruction is privileged and context synchronizing.

Special registers altered:

- MSR

NOTE

This instruction is only available in 32-bit Power ISA embedded category instruction pages. It is not available in VLE instruction pages.

se_rfmci se_rfmci

Return From Machine Check Interrupt

se_rfmci

0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1
0															15

```
MSR ←MCSRR1
PC ←MCSRR0$_{0:30}$ || $^{1}$0
```

The **se_rfmci** instruction is used to return from a machine check interrupt, or as a means of simultaneously establishing a new context and synchronizing on that new context.

The contents of machine check save/restore register 1 are place into the machine state register. If the new machine state register value does not enable any pending exceptions, then the next instruction is fetched, under control of the new machine state register value from the address MCSRR0[0–30] || 0b0. If the new machine state register value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into the appropriate save/restore register 0 by the interrupt processing mechanism is the address of the instruction that would have been executed next had the interrupt not occurred (that is, the address in machine check save/restore register 0 at the time of the execution of the **se_rfmci**).

Execution of this instruction is privileged and context synchronizing.

Special registers altered:

- MSR

NOTE

This instruction is only available in VLE instruction pages. It is not available in 32-bit Power ISA embedded category instruction pages.

3.12 WAIT Instruction

The **wait** instruction allows software to cease all synchronous activity, waiting for an asynchronous interrupt or debug interrupt to occur. The instruction can be used to cease processor activity in both user and supervisor modes. Asynchronous interrupts which will cause the waiting state to be exited if enabled are critical input, external input, machine check pin (*p_mcp_b*). Nonmaskable interrupts (**p_nmi_b**) also cause the waiting state to be exited.

wait

wait

Wait for Interrupt

wait

0					5	6 10 11 15 16 20	21										31
0	1	1	1	1	1	///	0	0	0	0	1	1	1	1	1	0	/

The **wait** instruction provides an ordering function for the effects of all instructions executed by the processor executing the **wait** instruction and stops synchronous processor activity. Executing a **wait** instruction ensures that all instructions have completed before the **wait** instruction completes, causes processor instruction fetching to cease, and ensures that no subsequent instructions are initiated until an asynchronous interrupt or a debug interrupt occurs.

Once the **wait** instruction has completed, the program counter will point to the next sequential instruction. The saved value in xSRR0 when the processor re-initiates activity will point to the instruction following the **wait** instruction.

Execution of a wait instruction places the CPU in the waiting state and is indicated by assertion of the *p_waiting* output signal. The signal will be negated after leaving the waiting state.

Software must ensure that interrupts responsible for exiting the waiting state are enabled before executing a **wait** instruction.

Architecture Note: The **wait** instruction can be used in verification test cases to signal the end of a test case. The encoding for the instruction is the same in both big- and little-endian modes.

3.13 Enhanced Reservations

The e200 implements the Freescale EIS enhanced reservations functionality that extends the load and reserve and store conditional instructions to support byte and half-word data types. These instructions operate in the same manner as the **lwarx** and **stwcx.** instructions, except for the size of the access.

lbarx lbarx

Load Byte And Reserve Indexed

lbarx **RT,RA,RB** **(X-mode)**

0 1 1 1 1 1	RT	RA	RB	0 0 0 0 1 1 0 1 0 0	/
0	6	11	16	21	31

```
if RA=0 then a ← $^{64}0$ else a ← GPR(RA)
if X-mode then EA ← $^{32}0$ || (a + GPR(RB))$_{32:63}$
RESERVE ← 1
RESERVE_ADDR ← real_addr(EA)
GPR(RT) ← $^{56}0$ || MEM(EA,1)
```

Let the effective address (EA) be calculated as follows:

- For **lbarx**, let EA be 32 zeros concatenated with bits 32–63 of the sum of the contents of GPR(RA), or 64 zeros if RA = 0, and the contents of GPR(RB).

The byte in storage addressed by EA is loaded into GPR(RT)[56–63]. GPR(RT)[0–55] are set to zero.

This instruction creates a reservation for use by a store byte conditional instruction. An address computed from the EA is associated with the reservation and replaces any address previously associated with the reservation.

Special registers altered:

- None

lharx lharx

Load Half Word And Reserve Indexed

lharx **RT,RA,RB** **(X-mode)**

0 1 1 1 1 1	RT	RA	RB	0 0 0 1 1 1 0 1 0 0	/
0	6	11	16	21	31

```
if RA=0 then a ← $^{64}0$ else a ← GPR(RA)
EA ← $^{32}0$ || (a + GPR(RB))$_{32:63}$
RESERVE ← 1
RESERVE_ADDR ← real_addr(EA)
GPR(RT) ← $^{48}0$ || MEM(EA,2)
```

Let the effective address (EA) be calculated as follows:

- For **lharx**, let EA be 32 zeros concatenated with bits 32–63 of the sum of the contents of GPR(RA), or 64 zeros if RA = 0, and the contents of GPR(RB).

The half-word in storage addressed by EA is loaded into GPR(RT)[48–63]. GPR(RT)[0–47] are set to zero.

This instruction creates a reservation for use by a Store Half Word Conditional instruction. An address computed from the EA is associated with the reservation and replaces any address previously associated with the reservation.

EA must be a multiple of 2. If it is not, either an alignment interrupt is invoked or the results are boundedly undefined.

Special registers altered:

- None

stbcx. stbcx.

Store Byte Conditional Indexed

stbcx. **RS,RA,RB** **(X-mode)**

0 1 1 1 1 1	RS	RA	RB	1 0 1 0 1 1 0 1 1 0	1
0	6	11	16	21	31

```
if RA=0 then a ← $^{64}0$ else a ← GPR(RA)
EA ← $^{32}0$ || (a + GPR(RB))$_{32:63}$
if RESERVE then
    if RESERVE_ADDR = real_addr(EA) then
        MEM(EA,1) ← GPR(RS)$_{56:63}$
        CR0 ← 0b00 || 0b1 || XER$_{SO}$
    else
        u ← undefined 1-bit value
        if u then MEM(EA,1) ¨ GPR(RS)$_{56:63}$
        CR0 ← 0b00 || u || XER$_{SO}$
    RESERVE ← 0
else
    CR0 ← 0b00 || 0b0 || XER$_{SO}$
```

Let the effective address (EA) be calculated as follows:

- For **stbcx.**, let EA be 32 zeros concatenated with bits 32–63 of the sum of the contents of GPR(RA), or 64 zeros if RA = 0, and the contents of GPR(RB).

If a reservation exists and the storage address specified by the **stbcx.** is the same as that specified by the **lbarx** instruction that established the reservation, the contents of bits 56–63 of GPR(RS) are stored into the byte in storage addressed by EA and the reservation is cleared.

If a reservation exists but the storage address specified by the **stbcx.** is not the same as that specified by the Load and Reserve instruction that established the reservation, the reservation is cleared, and it is undefined whether the instruction completes without altering storage.

If a reservation does not exist, the instruction completes without altering storage.

CR Field 0 is set to reflect whether the store operation was performed, as follows.

$CR0_{LT\ GT\ EQ\ SO}$ = 0b00 || store_performed || XER_{SO}

Special registers altered:
CR0

sthcx. sthcx.

Store Half Word Conditional Indexed

sthcx. **RS,RA,RB** **(X-mode)**

0 1 1 1 1 1	RS	RA	RB	1 0 1 1 0 1 0 1 1 0	1
0	6	11	16	21	31

```
if RA=0 then a ← $^{64}0$ else a ← GPR(RA)
EA ← $^{32}0$ || (a + GPR(RB))$_{32:63}$
if RESERVE then
    if RESERVE_ADDR = real_addr(EA) then
        MEM(EA,2) ← GPR(RS)$_{48:63}$
        CR0 ← 0b00 || 0b1 || XER$_{SO}$
    else
        u ← undefined 1-bit value
        if u then MEM(EA,2) ← GPR(RS)$_{48:63}$
        CR0 ← 0b00 || u || XER$_{SO}$
    RESERVE ← 0
else
    CR0 ← 0b00 || 0b0 || XER$_{SO}$
```

Let the effective address (EA) be calculated as follows:

- For **sthcx.**, let EA be 32 zeros concatenated with bits 32–63 of the sum of the contents of GPR(RA), or 64 zeros if RA = 0, and the contents of GPR(RB).

If a reservation exists and the storage address specified by the **sthcx.** is the same as that specified by the **lharx** instruction that established the reservation, the contents of bits 48–63 of GPR(RS) are stored into the half-word in storage addressed by EA and the reservation is cleared.

If a reservation exists but the storage address specified by the **sthcx.** is not the same as that specified by the Load and Reserve instruction that established the reservation, the reservation is cleared, and it is undefined whether the instruction completes without altering storage.

If a reservation does not exist, the instruction completes without altering storage.

CR Field 0 is set to reflect whether the store operation was performed, as follows.

$CR0_{LT\ GT\ EQ\ SO}$ = 0b00 || store_performed || XER_{SO}

EA must be a multiple of 2. If it is not, either an alignment interrupt is invoked or the results are boundedly undefined.

Special registers altered:

- CR0

3.14 Volatile Context Save/Restore Unit

The e200 implements the EIS volatile context save/restore unit to support the capability to quickly save and restore volatile register context on entry into an interrupt handler. To support this functionality, a new set of instructions is defined as part of the unit.

- e_lmvgprw, e_stmvgprw—load/store multiple volatile GPRs (r0, r3:r12)
- e_lmvsprw, e_stmvsprw—load/store multiple volatile SPRs (CR, LR, CTR, and XER)
- e_lmvsrrw, e_stmvsrrw—load/store multiple volatile SRRs (SRR0, SRR1)
- e_lmvcsrrw, e_stmvcsrrw—load/store multiple volatile CSRRs (CSRR0, CSRR1)
- e_lmvdsrrw, e_stmvdsrrw—load/store multiple volatile DSRRs (DSRR0, DSRR1)
- e_lmvmcsrrw, e_stmvmcsrrw —load/store multiple volatile MCSRRs (MCSRR0, MCSRR1)

These instructions are available in VLE instruction pages to perform a multiple register load or store to a word aligned memory address.

e_lmvgprw

Load Multiple Volatile GPR Word

e_lmvgprw D8(RA) (D8-mode)

0 0 0 1 1 0	0 0 0 0 0	RA	0 0 0 1 0 0 0 0	D8
0	6	11	16	24 31

```
if RA=0 then EA ← EXTS(D8)
else         EA ← (GPR(RA)+EXTS(D8))

GPR(r0)$_{32:63}$ ← MEM(EA,4)
EA ← (EA+4)

r ← 3
do while r ≤ 12
                GPR(r)$_{32:63}$ ← MEM(EA,4)
    EA ← (EA+4)
    r  ← r + 1
```

Let the effective address (EA) be the sum of the content of GPR(RA) and the sign-extended value of the D8 instruction field. If RA = 0, the content of GPR(RA) equals 0 and EA is the sign-extended value of the D8 instruction field.

Bits 32–63 of registers GPR(R0), and GPR(R3) through GPR(12) are loaded from *n* consecutive words in storage starting at address EA.

EA must be a multiple of 4. If it is not, either an Alignment interrupt is invoked or the results are boundedly undefined.

Special registers altered:
None

e_stmvgprw

e_stmvgprw

Store Multiple Volatile GPR Word

e_stmvgprw **D8(RA)** **(D8-mode)**

0 0 0 1 1 0	0 0 0 0 0	RA	0 0 0 1 0 0 0 1	D8
0	6	11	16	24 31

```
if RA=0 then EA ← EXTS(D8)
else         EA ← (GPR(RA)+EXTS(D8))

MEM(EA,4) ← GPR(r0)$_{32:63}$
EA ← (EA+4)

r ← 3
do while r ≤ 12
    MEM(EA,4) ← GPR(r)$_{32:63}$
    r  ← r + 1
    EA ← (EA+4)
```

Let the effective address (EA) be the sum of the content of GPR(RA) and the sign-extended value of the D8 instruction field. If RA = 0, the content of GPR(RA) equals 0 and EA is the sign-extended value of the D8 instruction field.

Bits 32–63 of registers GPR(R0), and GPR(R3) through GPR(12) are stored in *n* consecutive words in storage starting at address EA.

EA must be a multiple of 4. If it is not, either an Alignment interrupt is invoked or the results are boundedly undefined.

Special registers altered:
None

e_lmvsprw

e_lmvsprw

Load Multiple Volatile SPR Word

e_lmvsprw **D8(RA)** **(D8-mode)**

0 0 0 1 1 0	0 0 0 0 1	RA	0 0 0 1 0 0 0 0	D8
0	6	11	16	24 31

```
if RA=0 then EA ← EXTS(D8)
```

```
else           EA ← (GPR(RA)+EXTS(D8))
CR$_{32:63}$ ← MEM(EA,4)
EA ← (EA+4)

LR$_{32:63}$ ← MEM(EA,4)
EA ← (EA+4)

CTR$_{32:63}$ ← MEM(EA,4)
EA ← (EA+4)

XER$_{32:63}$ ← MEM(EA,4)
```

Let the effective address (EA) be the sum of the content of GPR(RA) and the sign-extended value of the D8 instruction field. If RA = 0, the content of GPR(RA) equals 0 and EA is the sign-extended value of the D8 instruction field.

Bits 32–63 of registers CR, LR, CTR, and XER are loaded from *n* consecutive words in storage starting at address EA.

EA must be a multiple of 4. If it is not, either an Alignment interrupt is invoked or the results are boundedly undefined.

Special registers altered:
CR, LR, CTR, XER

NOTE

If the EA is misaligned and the e_lmvsprw is followed by either a branch to link register or branch to count register within 4 instructions, the core can lock up during exception handling for the misalignment. To avoid this issue, do not do misaligned on e_lmvsprw or ensure there are at least 4 instructions in between the e_lmvsprw and the branch to LR or CTR. This issue does not apply to Book E applications.

e_stmvsprw e_stmvsprw

Store Multiple Volatile SPR Word

e_stmvsprw **D8(RA)** **(D8-mode)**

0 0 0 1 1 0	0 0 0 0 1	RA	0 0 0 1 0 0 0 1	D8
0	6	11	16	24 31

```
if RA=0 then EA ← EXTS(D8)
else         EA ← (GPR(RA)+EXTS(D8))
MEM(EA,4) ← CR$_{32:63}$
EA ← (EA+4)
MEM(EA,4) ← LR$_{32:63}$
EA ← (EA+4)
MEM(EA,4) ← CTR$_{32:63}$
```

```
EA ← (EA+4)

MEM(EA,4) ← XER$_{32:63}$
```

Let the effective address (EA) be the sum of the content of GPR(RA) and the sign-extended value of the D8 instruction field. If RA = 0, the content of GPR(RA) equals 0 and EA is the sign-extended value of the D8 instruction field.

Bits 32–63 of registers CR, LR, CTR, and XER are stored in *n* consecutive words in storage starting at address EA.

EA must be a multiple of 4. If it is not, either an Alignment interrupt is invoked or the results are boundedly undefined.

Special registers altered:
None

e_lmvsrrw e_lmvsrrw

Load Multiple Volatile SRR Word

e_lmvsrrw **D8(RA)** **(D8-mode)**

0 0 0 1 1 0	0 0 1 0 0	RA	0 0 0 1 0 0 0 0	D8
0	6	11	16	24 31

```
if RA=0 then EA ← EXTS(D8)
else         EA ← (GPR(RA)+EXTS(D8))
SRR0$_{32:63}$ ← MEM(EA,4)
EA ← (EA+4)
SRR1$_{32:63}$ ← MEM(EA,4)
```

Let the effective address (EA) be the sum of the content of GPR(RA) and the sign-extended value of the D8 instruction field. If RA = 0, the content of GPR(RA) equals 0 and EA is the sign-extended value of the D8 instruction field.

Bits 32–63 of registers SRR0 and SRR1 are loaded from consecutive words in storage starting at address EA.

EA must be a multiple of 4. If it is not, either an Alignment interrupt is invoked or the results are boundedly undefined.

Special registers altered:
SRR0, SRR1

e_stmvsrrw e_stmvsrrw

Store Multiple Volatile SRR Word

e_stmvsrrw D8(RA) (D8-mode)

0	6	11	16	24 31
0 0 0 1 1 0	0 0 1 0 0	RA	0 0 0 1 0 0 0 1	D8

```
if RA=0 then EA ← EXTS(D8)
else         EA ← (GPR(RA)+EXTS(D8))
MEM(EA,4) ← $SRR0_{32:63}$
EA ← (EA+4)
MEM(EA,4) ← $SRR1_{32:63}$
```

Let the effective address (EA) be the sum of the content of GPR(RA) and the sign-extended value of the D8 instruction field. If RA = 0, the content of GPR(RA) equals 0 and EA is the sign-extended value of the D8 instruction field.

Bits 32–63 of registers SRR0 and SRR1 are stored in consecutive words in storage starting at address EA.

EA must be a multiple of 4. If it is not, either an Alignment interrupt is invoked or the results are boundedly undefined.

Special registers altered:
None

e_lmvcsrrw e_lmvcsrrw

Load Multiple Volatile CSRR Word

e_lmvcsrrw D8(RA) (D8-mode)

0	6	11	16	24 31
0 0 0 1 1 0	0 0 1 0 1	RA	0 0 0 1 0 0 0 0	D8

```
if RA=0 then EA ← EXTS(D8)
else         EA ← (GPR(RA)+EXTS(D8))
$CSRR0_{32:63}$ ← MEM(EA,4)
EA ← (EA+4)
$CSRR1_{32:63}$ ← MEM(EA,4)
```

Let the effective address (EA) be the sum of the content of GPR(RA) and the sign-extended value of the D8 instruction field. If RA = 0, the content of GPR(RA) equals 0 and EA is the sign-extended value of the D8 instruction field.

Bits 32–63 of registers CSRR0 and CSRR1 are loaded from consecutive words in storage starting at address EA.

EA must be a multiple of 4. If it is not, either an Alignment interrupt is invoked or the results are boundedly undefined.

Special registers altered:
CSRR0, CSRR1

e_stmvcsrrw

e_stmvcsrrw

Store Multiple Volatile CSRR Word

e_stmvcsrrw **D8(RA)** **(D8-mode)**

0 0 0 1 1 0	0 0 1 0 1	RA	0 0 0 1 0 0 0 1	D8	
0	6	11	16	24	31

```
if RA=0 then EA ← EXTS(D8)
else         EA ← (GPR(RA)+EXTS(D8))

MEM(EA,4) ← $CSRR0_{32:63}$
EA ← (EA+4)
MEM(EA,4) ← $CSRR1_{32:63}$
```

Let the effective address (EA) be the sum of the content of GPR(RA) and the sign-extended value of the D8 instruction field. If RA = 0, the content of GPR(RA) equals 0 and EA is the sign-extended value of the D8 instruction field.

Bits 32–63 of registers CSRR0 and CSRR1 are stored in consecutive words in storage starting at address EA.

EA must be a multiple of 4. If it is not, either an Alignment interrupt is invoked or the results are boundedly undefined.

Special registers altered:
None

e_lmvdsrrw

e_lmvdsrrw

Load Multiple Volatile DSRR Word

e_lmvdsrrw **D8(RA)** **(D8-mode)**

0 0 0 1 1 0	0 0 1 1 0	RA	0 0 0 1 0 0 0 0	D8	
0	6	11	16	24	31

```
if RA=0 then EA ← EXTS(D8)
else         EA ← (GPR(RA)+EXTS(D8))
$DSRR0_{32:63}$ ← MEM(EA,4)
EA ← (EA+4)
$DSRR1_{32:63}$ ← MEM(EA,4)
```

Let the effective address (EA) be the sum of the content of GPR(RA) and the sign-extended value of the D8 instruction field. If RA = 0, the content of GPR(RA) equals 0 and EA is the sign-extended value of the D8 instruction field.

Bits 32–63 of registers DSRR0 and DSRR1 are loaded from consecutive words in storage starting at address EA.

EA must be a multiple of 4. If it is not, either an Alignment interrupt is invoked or the results are boundedly undefined.

Special registers altered:
DSRR0, DSRR1

e_stmvdsrrw e_stmvdsrrw

Store Multiple Volatile DSRR Word

e_stmvdsrrw **D8(RA)** **(D8-mode)**

0 0 0 1 1 0	0 0 1 1 0	RA	0 0 0 1 0 0 0 1	D8
0	6	11	16	24 31

```
if RA=0 then EA ← EXTS(D8)
else         EA ← (GPR(RA)+EXTS(D8))
MEM(EA,4) ← $DSRR0_{32:63}$
EA ← (EA+4)
MEM(EA,4) ← $DSRR1_{32:63}$
```

Let the effective address (EA) be the sum of the content of GPR(RA) and the sign-extended value of the D8 instruction field. If RA = 0, the content of GPR(RA) equals 0 and EA is the sign-extended value of the D8 instruction field.

Bits 32–63 of registers DSRR0 and DSRR1 are stored in consecutive words in storage starting at address EA.

EA must be a multiple of 4. If it is not, either an Alignment interrupt is invoked or the results are boundedly undefined.

Special registers altered:
None

e_lmvmcsrrw e_lmvmcsrrw

Load Multiple Volatile MCSRR Word

e_lmvmcsrrw D8(RA) (D8-mode)

0 0 0 1 1 0	0 0 1 1 1	RA	0 0 0 1 0 0 0 0	D8
0	6	11	16	24 31

```
if RA=0 then EA ← EXTS(D8)
else         EA ← (GPR(RA)+EXTS(D8))
$MCSRR0_{32:63}$ ← MEM(EA,4)
EA ← (EA+4)
$MCSRR1_{32:63}$ ← MEM(EA,4)
```

Let the effective address (EA) be the sum of the content of GPR(RA) and the sign-extended value of the D8 instruction field. If RA = 0, the content of GPR(RA) equals 0 and EA is the sign-extended value of the D8 instruction field.

Bits 32–63 of registers MCSRR0 and MCSRR1 are loaded from consecutive words in storage starting at address EA.

EA must be a multiple of 4. If it is not, either an Alignment interrupt is invoked or the results are boundedly undefined.

Special registers altered:
MCSRR0, MCSRR1

e_stmvmcsrrw e_stmvmcsrrw

Store Multiple Volatile MCSRR Word

e_stmvmcsrrw D8(RA) (D8-mode)

0 0 0 1 1 0	0 0 1 1 1	RA	0 0 0 1 0 0 0 1	D8
0	6	11	16	24 31

```
if RA=0 then EA ← EXTS(D8)
else         EA ← (GPR(RA)+EXTS(D8))
MEM(EA,4) ← $MCSRR0_{32:63}$
EA ← (EA+4)
MEM(EA,4) ← $MCSRR1_{32:63}$
```

Let the effective address (EA) be the sum of the content of GPR(RA) and the sign-extended value of the D8 instruction field. If RA = 0, the content of GPR(RA) equals 0 and EA is the sign-extended value of the D8 instruction field.

Bits 32–63 of registers MCSRR0 and MCSRR1 are stored in consecutive words in storage starting at address EA.

EA must be a multiple of 4. If it is not, either an Alignment interrupt is invoked or the results are boundedly undefined.

Special registers altered:
None

3.15 Unimplemented SPRs and Read-Only SPRs

The e200 fully decodes the SPR field of the **mfspr** and **mtspr** instructions. If the SPR specified is undefined and not privileged, an illegal instruction exception is generated**.** If the SPR specified is undefined and privileged and the CPU is in user mode (MSR[PR] = 1), a privileged instruction exception is generated. If the SPR specified is undefined and privileged and the CPU is in supervisor mode (MSR[PR] = 0), an illegal instruction exception is generated.

For the **mtspr** instruction, if the SPR specified is read-only and not privileged, an illegal instruction exception is generated**.** If the SPR specified is read-only and privileged and the CPU is in user mode (MSR[PR] = 1), a privileged instruction exception is generated. If the SPR specified is read-only and privileged and the CPU is in supervisor mode (MSR[PR] = 0), an illegal instruction exception is generated.

3.16 Invalid Forms of Instructions

This section discusses invalid forms of instructions.

3.16.1 Load and Store with Update instructions

The Power ISA embedded category defines the case when a load with update instruction specifies the same register in the RT and RA field of the instruction as an invalid format. For this invalid case, the e200 core will perform the instruction and update the register with the load data. In addition, if RA = 0 for any load or store with update instruction, the e200 core will update RA (GPR0).

3.16.2 Load Multiple Word (lmw, e_lmw) instruction

The Power ISA embedded category defines as invalid any form of the **lmw** or **e_lmw** instruction in which RA is in the range of registers to be loaded, including the case in which RA = 0. On the e200, invalid forms of the **lmw** or **e_lmw** instruction are executed as follows:

- Case 1: RA is in the range of RT, RA ≠ 0**.** In this case, address generation for individual loads to register targets is done using the architectural value of RA which existed when beginning execution of this **lmw** or **e_lmw** instruction. RA will be overwritten with a value fetched from memory as if it had not been the base register. Note that if the instruction is interrupted and restarted, the base address may be different if RA has been overwritten.
- Case 2: RA = 0 and RT = 0. In this case, address generation for all loads to register targets RT = 0 to RT = 31 will be done substituting the value of 0 for the RA operand.

3.16.3 Branch Conditional to Count Register Instructions

The Power ISA embedded category defines as invalid any **bcctr** or **bcctrl** instruction which specifies the ‘decrement and test CTR’ ($BO_2 = 0$) option. For these invalid forms of instructions e200 will execute the instruction by decrementing the CTR and branch to the location specified by the pre-decremented CTR value if all CR and CTR conditions are met as specified by the other BO field settings.

3.16.4 Instructions With Reserved Fields Non-Zero

The Power ISA embedded category defines certain bit fields in various instructions as reserved and specifies that these fields be set to zero. Per the Power ISA embedded category recommendation, e200 ignores the value of the reserved field (bit 31) in X-form integer load and store instructions. The e200 ignores the value of the reserved z bits in the BO field of branch instructions. For all other instructions, the e200 generates an illegal instruction exception if a reserved field is non-zero.

3.17 Instruction Summary

Table 3-2 and Table 3-3 list all 32-bit instructions in the Power ISA embedded category architecture as well as certain e200-specific instructions, sorted by mnemonic. The table includes the following: format, opcode, mnemonic, and instruction name. For e200-specific instructions, the page number is not shown. Instructions that are not listed here, but which are part of the Power ISA embedded category, either signal an illegal instruction, unimplemented operation, or FP unavailable exception. Implementation-dependent instructions are noted with a footnote. Instructions that are optionally supported (when an optional function is added to the base core) are shown with shaded entries.

Note that specific areas of functionality are not included in the table below:

- Cache maintenance instructions
- SPE
- VLE
- WAIT instruction
- Enhanced reservation functionality
- Volatile context save/restore

Table 3-2 lists the instruction index sorted by mnemonic.

Table 3-2. Instructions Sorted by Mnemonic

Format	Opcode		Mnemonic	Instruction
	Primary (Inst[0–5])	Extended (Inst[21–31])		
X	011111	01000 01010 0	**add**	Add
X	011111	01000 01010 1	**add.**	Add and Record CR
X	011111	00000 01010 0	**addc**	Add Carrying
X	011111	00000 01010 1	**addc.**	Add Carrying and Record CR

Table 3-2. Instructions Sorted by Mnemonic (continued)

Format	Opcode		Mnemonic	Instruction
	Primary (Inst[0–5])	Extended (Inst[21–31])		
X	011111	10000 01010 0	**addco**	Add Carrying and Record OV
X	011111	10000 01010 1	**addco.**	Add Carrying and Record OV & CR
X	011111	00100 01010 0	**adde**	Add Extended with CA
X	011111	00100 01010 1	**adde.**	Add Extended with CA and Record CR
X	011111	10100 01010 0	**addeo**	Add Extended with CA and Record OV
X	011111	10100 01010 1	**addeo.**	Add Extended with CA and Record OV & CR
D	001110	----- ----- -	**addi**	Add Immediate
D	001100	----- ----- -	**addic**	Add Immediate Carrying
D	001101	----- ----- -	**addic.**	Add Immediate Carrying and Record CR
D	001111	----- ----- -	**addis**	Add Immediate Shifted
X	011111	00111 01010 0	**addme**	Add to Minus One Extended with CA
X	011111	00111 01010 1	**addme.**	Add to Minus One Extended with CA and Record CR
X	011111	10111 01010 0	**addmeo**	Add to Minus One Extended with CA and Record OV
X	011111	10111 01010 1	**addmeo.**	Add to Minus One Extended with CA and Record OV & CR
X	011111	11000 01010 0	**addo**	Add and Record OV
X	011111	11000 01010 1	**addo.**	Add and Record OV and CR
X	011111	00110 01010 0	**addze**	Add to Zero Extended with CA
X	011111	00110 01010 1	**addze.**	Add to Zero Extended with CA and Record CR
X	011111	10110 01010 0	**addzeo**	Add to Zero Extended with CA and Record OV
X	011111	10110 01010 1	**addzeo.**	Add to Zero Extended with CA and Record OV & CR
X	011111	00000 11100 0	**and**	AND
X	011111	00000 11100 1	**and.**	AND and Record CR
X	011111	00001 11100 0	**andc**	AND with Complement
X	011111	00001 11100 1	**andc.**	AND with Complement and Record CR
D	011100	----- ----- -	**andi.**	AND Immediate and Record CR
D	011101	----- ----- -	**andis.**	AND Immediate Shifted and Record CR
I	010010	----- ----0 0	**b**	Branch
I	010010	----- ----1 0	**ba**	Branch Absolute
B	010000	----- ----0 0	**bc**	Branch Conditional
B	010000	----- ----1 0	**bca**	Branch Conditional Absolute
XL	010011	10000 10000 0	**bcctr**	Branch Conditional to Count Register

Table 3-2. Instructions Sorted by Mnemonic (continued)

Format	Opcode		Mnemonic	Instruction
	Primary (Inst[0–5])	Extended (Inst[21–31])		
XL	010011	10000 10000 1	**bcctrl**	Branch Conditional to Count Register and Link
B	010000	----- ----0 1	**bcl**	Branch Conditional and Link
B	010000	----- ----1 1	**bcla**	Branch Conditional and Link Absolute
XL	010011	00000 10000 0	**bclr**	Branch Conditional to Link Register
XL	010011	00000 10000 1	**bclrl**	Branch Conditional to Link Register & Link
I	010010	----- ----0 1	**bl**	Branch and Link
I	010010	----- ----1 1	**bla**	Branch and Link Absolute
X	011111	00000 00000 /	**cmp**	Compare
D	001011	----- ----- -	**cmpi**	Compare Immediate
X	011111	00001 00000 /	**cmpl**	Compare Logical
D	001010	----- ----- -	**cmpli**	Compare Logical Immediate
X	011111	00000 11010 0	**cntlzw**	Count Leading Zeros Word
X	011111	00000 11010 1	**cntlzw.**	Count Leading Zeros Word & Record CR
XL	010011	01000 00001 /	**crand**	Condition Register AND
XL	010011	00100 00001 /	**crandc**	Condition Register AND with Complement
XL	010011	01001 00001 /	**creqv**	Condition Register Equivalent
XL	010011	00111 00001 /	**crnand**	Condition Register NAND
XL	010011	00001 00001 /	**crnor**	Condition Register NOR
XL	010011	01110 00001 /	**cror**	Condition Register OR
XL	010011	01101 00001 /	**crorc**	Condition Register OR with Complement
XL	010011	00110 00001 /	**crxor**	Condition Register XOR
X	011111	10111 10110 /	**dcba**	Data Cache Block Allocate
X	011111	00010 10110 /	**dcbf**	Data Cache Block Flush
X	011111	01110 10110 /	**dcbi**	Data Cache Block Invalidate
X	011111	01100 00110 /	**dcblc**	Data Cache Block Lock Clear
X	011111	00001 10110 /	**dcbst**	Data Cache Block Store
X	011111	01000 10110 /	**dcbt**	Data Cache Block Touch
X	011111	00101 00110 /	**dcbtls**	Data Cache Block Touch and Lock Set
X	011111	00111 10110 /	**dcbtst**	Data Cache Block Touch for Store
X	011111	00100 00110 /	**dcbtstls**	Data Cache Block Touch for Store and Lock Set
X	011111	11111 10110 /	**dcbz**	Data Cache Block Set to Zero

Table 3-2. Instructions Sorted by Mnemonic (continued)

Format	Opcode		Mnemonic	Instruction
	Primary (Inst[0–5])	Extended (Inst[21–31])		
X	011111	01111 01011 0	**divw**	Divide Word
X	011111	01111 01011 1	**divw.**	Divide Word and Record CR
X	011111	11111 01011 0	**divwo**	Divide Word and Record OV
X	011111	11111 01011 1	**divwo.**	Divide Word and Record OV & CR
X	011111	01110 01011 0	**divwu**	Divide Word Unsigned
X	011111	01110 01011 1	**divwu.**	Divide Word Unsigned and Record CR
X	011111	11110 01011 0	**divwuo**	Divide Word Unsigned and Record OV
X	011111	11110 01011 1	**divwuo.**	Divide Word Unsigned and Record OV & CR
X	011111	01000 11100 0	**eqv**	Equivalent
X	011111	01000 11100 1	**eqv.**	Equivalent and Record CR
X	011111	11101 11010 0	**extsb**	Extend Sign Byte
X	011111	11101 11010 1	**extsb.**	Extend Sign Byte and Record CR
X	011111	11100 11010 0	**extsh**	Extend Sign Half Word
X	011111	11100 11010 1	**extsh.**	Extend Sign Half Word and Record CR
X	011111	11110 10110 /	**icbi**	Instruction Cache Block Invalidate
X	011111	00111 00110 /	**icblc**	Instruction Cache Block Lock Clear
X	011111	00000 10110 /	**icbt**	Instruction Cache Block Touch
X	011111	01111 00110 /	**icbtls**	Instruction Cache Block Touch and Lock Set
??	011111	----- 01111 /	**isel**	Integer Select
XL	010011	00100 10110 /	**isync**	Instruction Synchronize
D	100010	----- ----- -	**lbz**	Load Byte & Zero
D	100011	----- ----- -	**lbzu**	Load Byte & Zero with Update
X	011111	00011 10111 /	**lbzux**	Load Byte & Zero with Update Indexed
X	011111	00010 10111 /	**lbzx**	Load Byte & Zero Indexed
D	101010	----- ----- -	**lha**	Load Half Word Algebraic
D	101011	----- ----- -	**lhau**	Load Half Word Algebraic with Update
X	011111	01011 10111 /	**lhaux**	Load Half Word Algebraic with Update Indexed
X	011111	01010 10111 /	**lhax**	Load Half Word Algebraic Indexed
X	011111	11000 10110 /	**lhbrx**	Load Half Word Byte-Reverse Indexed
D	101000	----- ----- -	**lhz**	Load Half Word & Zero
D	101001	----- ----- -	**lhzu**	Load Half Word & Zero with Update

Table 3-2. Instructions Sorted by Mnemonic (continued)

Format	Opcode		Mnemonic	Instruction
	Primary (Inst[0–5])	Extended (Inst[21–31])		
X	011111	01001 10111 /	**lhzux**	Load Half Word & Zero with Update Indexed
X	011111	01000 10111 /	**lhzx**	Load Half Word & Zero Indexed
D	101110	----- ----- -	**lmw**	Load Multiple Word
X	011111	00000 10100 /	**lwarx**	Load Word & Reserve Indexed
X	011111	10000 10110 /	**lwbrx**	Load Word Byte-Reverse Indexed
D	100000	----- ----- -	**lwz**	Load Word & Zero
D	100001	----- ----- -	**lwzu**	Load Word & Zero with Update
X	011111	00001 10111 /	**lwzux**	Load Word & Zero with Update Indexed
X	011111	00000 10111 /	**lwzx**	Load Word & Zero Indexed
X	011111	11010 10110 /	**mbar**	Memory Barrier
XL	010011	00000 00000 /	**mcrf**	Move Condition Register Field
X	011111	10000 00000 /	**mcrxr**	Move to Condition Register from XER
X	011111	00000 10011 /	**mfcr**	Move From Condition Register
XFX	011111	01010 00011 /	**mfdcr**	Move From Device Control Register
X	011111	00010 10011 /	**mfmsr**	Move From Machine State Register
XFX	011111	01010 10011 /	**mfspr**	Move From Special Purpose Register
X	011111	10010 10110 /	**msync**	Memory Synchronize
XFX	011111	00100 10000 /	**mtcrf**	Move To Condition Register Fields
XFX	011111	01110 00011 /	**mtdcr**	Move To Device Control Register
X	011111	00100 10010 /	**mtmsr**	Move To Machine State Register
XFX	011111	01110 10011 /	**mtspr**	Move To Special Purpose Register
X	011111	/0010 01011 0	**mulhw**	Multiply High Word
X	011111	/0010 01011 1	**mulhw.**	Multiply High Word & Record CR
X	011111	/0000 01011 0	**mulhwu**	Multiply High Word Unsigned
X	011111	/0000 01011 1	**mulhwu.**	Multiply High Word Unsigned & Record CR
D	000111	----- ----- -	**mulli**	Multiply Low Immediate
X	011111	00111 01011 0	**mullw**	Multiply Low Word
X	011111	00111 01011 1	**mullw.**	Multiply Low Word & Record CR
X	011111	10111 01011 0	**mullwo**	Multiply Low Word & Record OV
X	011111	10111 01011 1	**mullwo.**	Multiply Low Word & Record OV & CR
X	011111	01110 11100 0	**nand**	NAND

Table 3-2. Instructions Sorted by Mnemonic (continued)

Format	Opcode		Mnemonic	Instruction
	Primary (Inst[0–5])	Extended (Inst[21–31])		
X	011111	01110 11100 1	**nand.**	NAND & Record CR
X	011111	00011 01000 0	**neg**	Negate
X	011111	00011 01000 1	**neg.**	Negate & Record CR
X	011111	10011 01000 0	**nego**	Negate & Record OV
X	011111	10011 01000 1	**nego.**	Negate & Record OV & Record CR
X	011111	00011 11100 0	**nor**	NOR
X	011111	00011 11100 1	**nor.**	NOR & Record CR
X	011111	01101 11100 0	**or**	OR
X	011111	01101 11100 1	**or.**	OR & Record CR
X	011111	01100 11100 0	**orc**	OR with Complement
X	011111	01100 11100 1	**orc.**	OR with Complement & Record CR
D	011000	----- ----- -	**ori**	OR Immediate
D	011001	----- ----- -	**oris**	OR Immediate Shifted
XL	010011	00001 10011 /	**rfci**	Return From Critical Interrupt
XL	010011	00001 00111 /	**rfdi**	Return From Debug Interrupt
XL	010011	00001 10010 /	**rfi**	Return From Interrupt
XL	010011	00001 00110 /	**rfmci**	Return From Machine Check Interrupt
M	010100	----- ----- 0	**rlwimi**	Rotate Left Word Immediate then Mask Insert
M	010100	----- ----- 1	**rlwimi.**	Rotate Left Word Immediate then Mask Insert & Record CR
M	010101	----- ----- 0	**rlwinm**	Rotate Left Word Immediate then AND with Mask
M	010101	----- ----- 1	**rlwinm.**	Rotate Left Word Immediate then AND with Mask & Record CR
M	010111	----- ----- 0	**rlwnm**	Rotate Left Word then AND with Mask
M	010111	----- ----- 1	**rlwnm.**	Rotate Left Word then AND with Mask & Record CR
SC	010001	///// ////1 /	**sc**	System Call
X	011111	00000 11000 0	**slw**	Shift Left Word
X	011111	00000 11000 1	**slw.**	Shift Left Word & Record CR
X	011111	11000 11000 0	**sraw**	Shift Right Algebraic Word
X	011111	11000 11000 1	**sraw.**	Shift Right Algebraic Word & Record CR
X	011111	11001 11000 0	**srawi**	Shift Right Algebraic Word Immediate
X	011111	11001 11000 1	**srawi.**	Shift Right Algebraic Word Immediate & Record CR
X	011111	10000 11000 0	**srw**	Shift Right Word

Table 3-2. Instructions Sorted by Mnemonic (continued)

Format	Opcode		Mnemonic	Instruction
	Primary (Inst[0–5])	Extended (Inst[21–31])		
X	011111	10000 11000 1	**srw.**	Shift Right Word & Record CR
D	100110	----- ----- -	**stb**	Store Byte
D	100111	----- ----- -	**stbu**	Store Byte with Update
X	011111	00111 10111 /	**stbux**	Store Byte with Update Indexed
X	011111	00110 10111 /	**stbx**	Store Byte Indexed
D	101100	----- ----- -	**sth**	Store Half Word
X	011111	11100 10110 /	**sthbrx**	Store Half Word Byte-Reverse Indexed
D	101101	----- ----- -	**sthu**	Store Half Word with Update
X	011111	01101 10111 /	**sthux**	Store Half Word with Update Indexed
X	011111	01100 10111 /	**sthx**	Store Half Word Indexed
D	101111	----- ----- -	**stmw**	Store Multiple Word
D	100100	----- ----- -	**stw**	Store Word
X	011111	10100 10110 /	**stwbrx**	Store Word Byte-Reverse Indexed
X	011111	00100 10110 1	**stwcx.**	Store Word Conditional Indexed & Record CR
D	100101	----- ----- -	**stwu**	Store Word with Update
X	011111	00101 10111 /	**stwux**	Store Word with Update Indexed
X	011111	00100 10111 /	**stwx**	Store Word Indexed
X	011111	00001 01000 0	**subf**	Subtract From
X	011111	00001 01000 1	**subf.**	Subtract From & Record CR
X	011111	00000 01000 0	**subfc**	Subtract From Carrying
X	011111	00000 01000 1	**subfc.**	Subtract From Carrying & Record CR
X	011111	10000 01000 0	**subfco**	Subtract From Carrying & Record OV
X	011111	10000 01000 1	**subfco.**	Subtract From Carrying & Record OV & CR
X	011111	00100 01000 0	**subfe**	Subtract From Extended with CA
X	011111	00100 01000 1	**subfe.**	Subtract From Extended with CA & Record CR
X	011111	10100 01000 0	**subfeo**	Subtract From Extended with CA & Record OV
X	011111	10100 01000 1	**subfeo.**	Subtract From Extended with CA & Record OV & CR
D	001000	----- ----- -	**subfic**	Subtract From Immediate Carrying
X	011111	00111 01000 0	**subfme**	Subtract From Minus One Extended with CA
X	011111	00111 01000 1	**subfme.**	Subtract From Minus One Extended with CA & Record CR
X	011111	10111 01000 0	**subfmeo**	Subtract From Minus One Extended with CA & Record OV

Table 3-2. Instructions Sorted by Mnemonic (continued)

Format	Opcode		Mnemonic	Instruction
	Primary (Inst[0–5])	Extended (Inst[21–31])		
X	011111	10111 01000 1	**subfmeo.**	Subtract From Minus One Extended with CA & Record OV & CR
X	011111	10001 01000 0	**subfo**	Subtract From & Record OV
X	011111	10001 01000 1	**subfo.**	Subtract From & Record OV & CR
X	011111	00110 01000 0	**subfze**	Subtract From Zero Extended with CA
X	011111	00110 01000 1	**subfze.**	Subtract From Zero Extended with CA & Record CR
X	011111	10110 01000 0	**subfzeo**	Subtract From Zero Extended with CA & Record OV
X	011111	10110 01000 1	**subfzeo.**	Subtract From Zero Extended with CA & Record OV & CR
X	011111	11000 10010 /	**tlbivax**	TLB Invalidate Virtual Address Indexed
X	011111	11101 10010 /	**tlbre**	TLB Read Entry
X	011111	11100 10010 ?	**tlbsx**	TLB Search Indexed
X	011111	10001 10110 /	**tlbsync**	TLB Synchronize
X	011111	11110 10010 /	**tlbwe**	TLB Write Entry
X	011111	00000 00100 /	**tw**	Trap Word
D	000011	----- ----- -	**twi**	Trap Word Immediate
X	011111	00100 00011 /	**wrtee**	Write External Enable
X	011111	00101 00011 /	**wrteei**	Write External Enable Immediate
X	011111	01001 11100 0	**xor**	XOR
X	011111	01001 11100 1	**xor.**	XOR and Record CR
D	011010	----- ----- -	**xori**	XOR Immediate
D	011011	----- ----- -	**xoris**	XOR Immediate Shifted

Note:

- Don’t care, usually part of an operand field.

/ Reserved bit, invalid instruction form if encoded as 1.

? Allocated for implementation-dependent use. See user’s manual for the implementation.

Table 3-3 lists the instruction index sorted by opcode.

Table 3-3. Instructions Sorted by Opcode

Format	Opcode		Mnemonic	Instruction
	Primary (Inst[0–5])	Extended (Inst[21–31])		
D	000011	----- ----- -	**twi**	Trap Word Immediate
D	000111	----- ----- -	**mulli**	Multiply Low Immediate

Table 3-3. Instructions Sorted by Opcode (continued)

Format	Opcode		Mnemonic	Instruction
	Primary (Inst[0–5])	Extended (Inst[21–31])		
D	001000	----- ----- -	**subfic**	Subtract From Immediate Carrying
D	001010	----- ----- -	**cmpli**	Compare Logical Immediate
D	001011	----- ----- -	**cmpi**	Compare Immediate
D	001100	----- ----- -	**addic**	Add Immediate Carrying
D	001101	----- ----- -	**addic.**	Add Immediate Carrying & Record CR
D	001110	----- ----- -	**addi**	Add Immediate
D	001111	----- ----- -	**addis**	Add Immediate Shifted
B	010000	----- ----0 0	**bc**	Branch Conditional
B	010000	----- ----0 1	**bcl**	Branch Conditional & Link
B	010000	----- ----1 0	**bca**	Branch Conditional Absolute
B	010000	----- ----1 1	**bcla**	Branch Conditional & Link Absolute
SC	010001	///// ////1 /	**sc**	System Call
I	010010	----- ----0 0	**b**	Branch
I	010010	----- ----0 1	**bl**	Branch & Link
I	010010	----- ----1 0	**ba**	Branch Absolute
I	010010	----- ----1 1	**bla**	Branch & Link Absolute
XL	010011	00000 00000 /	**mcrf**	Move Condition Register Field
XL	010011	00000 10000 0	**bclr**	Branch Conditional to Link Register
XL	010011	00000 10000 1	**bclrl**	Branch Conditional to Link Register & Link
XL	010011	00001 00001 /	**crnor**	Condition Register NOR
XL	010011	00001 00110 /	**rfmci**	Return From Machine Check Interrupt
XL	010011	00001 00111 /	**rfdi**	Return From Debug Interrupt
XL	010011	00001 10010 /	**rfi**	Return From Interrupt
XL	010011	00001 10011 /	**rfci**	Return From Critical Interrupt
XL	010011	00100 00001 /	**crandc**	Condition Register AND with Complement
XL	010011	00100 10110 /	**isync**	Instruction Synchronize
XL	010011	00110 00001 /	**crxor**	Condition Register XOR
XL	010011	00111 00001 /	**crnand**	Condition Register NAND
XL	010011	01000 00001 /	**crand**	Condition Register AND
XL	010011	01001 00001 /	**creqv**	Condition Register Equivalent
XL	010011	01101 00001 /	**crorc**	Condition Register OR with Complement

Table 3-3. Instructions Sorted by Opcode (continued)

Format	Opcode		Mnemonic	Instruction
	Primary (Inst[0–5])	Extended (Inst[21–31])		
XL	010011	01110 00001 /	**cror**	Condition Register OR
XL	010011	10000 10000 0	**bcctr**	Branch Conditional to Count Register
XL	010011	10000 10000 1	**bcctrl**	Branch Conditional to Count Register & Link
M	010100	----- ----- 0	**rlwimi**	Rotate Left Word Immediate then Mask Insert
M	010100	----- ----- 1	**rlwimi.**	Rotate Left Word Immediate then Mask Insert & Record CR
M	010101	----- ----- 0	**rlwinm**	Rotate Left Word Immediate then AND with Mask
M	010101	----- ----- 1	**rlwinm.**	Rotate Left Word Immediate then AND with Mask & Record CR
M	010111	----- ----- 0	**rlwnm**	Rotate Left Word then AND with Mask
M	010111	----- ----- 1	**rlwnm.**	Rotate Left Word then AND with Mask & Record CR
D	011000	----- ----- -	**ori**	OR Immediate
D	011001	----- ----- -	**oris**	OR Immediate Shifted
D	011010	----- ----- -	**xori**	XOR Immediate
D	011011	----- ----- -	**xoris**	XOR Immediate Shifted
D	011100	----- ----- -	**andi.**	AND Immediate & Record CR
D	011101	----- ----- -	**andis.**	AND Immediate Shifted & Record CR
??	011111	----- 01111 /	**isel**	Integer Select
X	011111	00000 00000 /	**cmp**	Compare
X	011111	00000 00100 /	**tw**	Trap Word
X	011111	00000 01000 0	**subfc**	Subtract From Carrying
X	011111	00000 01000 1	**subfc.**	Subtract From Carrying & Record CR
X	011111	00000 01010 0	**addc**	Add Carrying
X	011111	00000 01010 1	**addc.**	Add Carrying & Record CR
X	011111	/0000 01011 0	**mulhwu**	Multiply High Word Unsigned
X	011111	/0000 01011 1	**mulhwu.**	Multiply High Word Unsigned & Record CR
X	011111	00000 10011 /	**mfcr**	Move From Condition Register
X	011111	00000 10100 /	**lwarx**	Load Word & Reserve Indexed
X	011111	00000 10110 /	**icbt**	Instruction Cache Block Touch
X	011111	00000 10111 /	**lwzx**	Load Word & Zero Indexed
X	011111	00000 11000 0	**slw**	Shift Left Word
X	011111	00000 11000 1	**slw.**	Shift Left Word & Record CR
X	011111	00000 11010 0	**cntlzw**	Count Leading Zeros Word

Table 3-3. Instructions Sorted by Opcode (continued)

Format	Opcode		Mnemonic	Instruction
	Primary (Inst[0–5])	Extended (Inst[21–31])		
X	011111	00000 11010 1	**cntlzw.**	Count Leading Zeros Word & Record CR
X	011111	00000 11100 0	**and**	AND
X	011111	00000 11100 1	**and.**	AND & Record CR
X	011111	00001 00000 /	**cmpl**	Compare Logical
X	011111	00001 01000 0	**subf**	Subtract From
X	011111	00001 01000 1	**subf.**	Subtract From & Record CR
X	011111	00001 10110 /	**dcbst**	Data Cache Block Store
X	011111	00001 10111 /	**lwzux**	Load Word & Zero with Update Indexed
X	011111	00001 11100 0	**andc**	AND with Complement
X	011111	00001 11100 1	**andc.**	AND with Complement & Record CR
X	011111	/0010 01011 0	**mulhw**	Multiply High Word
X	011111	/0010 01011 1	**mulhw.**	Multiply High Word & Record CR
X	011111	00010 10011 /	**mfmsr**	Move From Machine State Register
X	011111	00010 10110 /	**dcbf**	Data Cache Block Flush
X	011111	00010 10111 /	**lbzx**	Load Byte & Zero Indexed
X	011111	00011 01000 0	**neg**	Negate
X	011111	00011 01000 1	**neg.**	Negate & Record CR
X	011111	00011 10111 /	**lbzux**	Load Byte & Zero with Update Indexed
X	011111	00011 11100 0	**nor**	NOR
X	011111	00011 11100 1	**nor.**	NOR & Record CR
X	011111	00100 00011 /	**wrtee**	Write External Enable
X	011111	00100 00110 /	**dcbtstls**	Data Cache Block Touch for Store and Lock Set
X	011111	00100 01000 0	**subfe**	Subtract From Extended with CA
X	011111	00100 01000 1	**subfe.**	Subtract From Extended with CA & Record CR
X	011111	00100 01010 0	**adde**	Add Extended with CA
X	011111	00100 01010 1	**adde.**	Add Extended with CA & Record CR
XFX	011111	00100 10000 /	**mtcrf**	Move to Condition Register Fields
X	011111	00100 10010 /	**mtmsr**	Move to Machine State Register
X	011111	00100 10110 1	**stwcx.**	Store Word Conditional Indexed & Record CR
X	011111	00100 10111 /	**stwx**	Store Word Indexed
X	011111	00101 00011 /	**wrteei**	Write External Enable Immediate

Table 3-3. Instructions Sorted by Opcode (continued)

Format	Opcode		Mnemonic	Instruction
	Primary (Inst[0–5])	Extended (Inst[21–31])		
X	011111	00101 00110 /	**dcbtls**	Data Cache Block Touch and Lock Set
X	011111	00101 10111 /	**stwux**	Store Word with Update Indexed
X	011111	00110 01000 0	**subfze**	Subtract From Zero Extended with CA
X	011111	00110 01000 1	**subfze.**	Subtract From Zero Extended with CA & Record CR
X	011111	00110 01010 0	**addze**	Add to Zero Extended with CA
X	011111	00110 01010 1	**addze.**	Add to Zero Extended with CA & Record CR
X	011111	00110 10111 /	**stbx**	Store Byte Indexed
X	011111	00111 00110 /	**icblc**	Instruction Cache Block Lock Clear
X	011111	00111 01000 0	**subfme**	Subtract From Minus One Extended with CA
X	011111	00111 01000 1	**subfme.**	Subtract From Minus One Extended with CA & Record CR
X	011111	00111 01010 0	**addme**	Add to Minus One Extended with CA
X	011111	00111 01010 1	**addme.**	Add to Minus One Extended with CA & Record CR
X	011111	00111 01011 0	**mullw**	Multiply Low Word
X	011111	00111 01011 1	**mullw.**	Multiply Low Word & Record CR
X	011111	00111 10110 /	**dcbtst**	Data Cache Block Touch for Store
X	011111	00111 10111 /	**stbux**	Store Byte with Update Indexed
X	011111	01000 01010 0	**add**	Add
X	011111	01000 01010 1	**add.**	Add & Record CR
X	011111	01000 10110 /	**dcbt**	Data Cache Block Touch
X	011111	01000 10111 /	**lhzx**	Load Half Word & Zero Indexed
X	011111	01000 11100 0	**eqv**	Equivalent
X	011111	01000 11100 1	**eqv.**	Equivalent & Record CR
X	011111	01001 10111 /	**lhzux**	Load Half Word & Zero with Update Indexed
X	011111	01001 11100 0	**xor**	XOR
X	011111	01001 11100 1	**xor.**	XOR & Record CR
XFX	011111	01010 00011 /	**mfdcr**	Move From Device Control Register
XFX	011111	01010 10011 /	**mfspr**	Move From Special Purpose Register
X	011111	01010 10111 /	**lhax**	Load Half Word Algebraic Indexed
X	011111	01011 10111 /	**lhaux**	Load Half Word Algebraic with Update Indexed
X	011111	01100 00110 /	**dcblc**	Data Cache Block Lock Clear
X	011111	01100 10111 /	**sthx**	Store Half Word Indexed

Table 3-3. Instructions Sorted by Opcode (continued)

Format	Opcode		Mnemonic	Instruction
	Primary (Inst[0–5])	Extended (Inst[21–31])		
X	011111	01100 11100 0	**orc**	OR with Complement
X	011111	01100 11100 1	**orc.**	OR with Complement & Record CR
X	011111	01101 10111 /	**sthux**	Store Half Word with Update Indexed
X	011111	01101 11100 0	**or**	OR
X	011111	01101 11100 1	**or.**	OR & Record CR
XFX	011111	01110 00011 /	**mtdcr**	Move to Device Control Register
X	011111	01110 01011 0	**divwu**	Divide Word Unsigned
X	011111	01110 01011 1	**divwu.**	Divide Word Unsigned & Record CR
XFX	011111	01110 10011 /	**mtspr**	Move to Special Purpose Register
X	011111	01110 10110 /	**dcbi**	Data Cache Block Invalidate
X	011111	01110 11100 0	**nand**	NAND
X	011111	01110 11100 1	**nand.**	NAND & Record CR
X	011111	01111 00110 /	**icbtls**	Instruction Cache Block Touch and Lock Set
X	011111	01111 01011 0	**divw**	Divide Word
X	011111	01111 01011 1	**divw.**	Divide Word & Record CR
X	011111	10000 00000 /	**mcrxr**	Move to Condition Register from XER
X	011111	10000 01000 0	**subfco**	Subtract From Carrying & Record OV
X	011111	10000 01000 1	**subfco.**	Subtract From Carrying & Record OV & CR
X	011111	10000 01010 0	**addco**	Add Carrying & Record OV
X	011111	10000 01010 1	**addco.**	Add Carrying & Record OV & CR
X	011111	10000 10110 /	**lwbrx**	Load Word Byte-Reverse Indexed
X	011111	10000 11000 0	**srw**	Shift Right Word
X	011111	10000 11000 1	**srw.**	Shift Right Word & Record CR
X	011111	10001 01000 0	**subfo**	Subtract From & Record OV
X	011111	10001 01000 1	**subfo.**	Subtract From & Record OV & CR
X	011111	10001 10110 /	**tlbsync**	TLB Synchronize
X	011111	10010 10110 /	**msync**	Memory Synchronize
X	011111	10011 01000 0	**nego**	Negate & Record OV
X	011111	10011 01000 1	**nego.**	Negate & Record OV & Record CR
X	011111	10100 01000 0	**subfeo**	Subtract From Extended with CA & Record OV
X	011111	10100 01000 1	**subfeo.**	Subtract From Extended with CA & Record OV & CR

Table 3-3. Instructions Sorted by Opcode (continued)

Format	Opcode		Mnemonic	Instruction
	Primary (Inst[0–5])	Extended (Inst[21–31])		
X	011111	10100 01010 0	**addeo**	Add Extended with CA & Record OV
X	011111	10100 01010 1	**addeo.**	Add Extended with CA & Record OV & CR
X	011111	10100 10110 /	**stwbrx**	Store Word Byte-Reverse Indexed
X	011111	10110 01000 0	**subfzeo**	Subtract From Zero Extended with CA & Record OV
X	011111	10110 01000 1	**subfzeo.**	Subtract From Zero Extended with CA & Record OV & CR
X	011111	10110 01010 0	**addzeo**	Add to Zero Extended with CA & Record OV
X	011111	10110 01010 1	**addzeo.**	Add to Zero Extended with CA & Record OV & CR
X	011111	10111 01000 0	**subfmeo**	Subtract From Minus One Extended with CA & Record OV
X	011111	10111 01000 1	**subfmeo.**	Subtract From Minus One Extended with CA & Record OV & CR
X	011111	10111 01010 0	**addmeo**	Add to Minus One Extended with CA & Record OV
X	011111	10111 01010 1	**addmeo.**	Add to Minus One Extended with CA & Record OV & CR
X	011111	10111 01011 0	**mullwo**	Multiply Low Word & Record OV
X	011111	10111 01011 1	**mullwo.**	Multiply Low Word & Record OV & CR
X	011111	10111 10110 /	**dcba**	Data Cache Block Allocate
X	011111	11000 01010 0	**addo**	Add & Record OV
X	011111	11000 01010 1	**addo.**	Add & Record OV & CR
X	011111	11000 10010 /	**tlbivax**	TLB Invalidate Virtual Address Indexed
X	011111	11000 10110 /	**lhbrx**	Load Half Word Byte-Reverse Indexed
X	011111	11000 11000 0	**sraw**	Shift Right Algebraic Word
X	011111	11000 11000 1	**sraw.**	Shift Right Algebraic Word & Record CR
X	011111	11001 11000 0	**srawi**	Shift Right Algebraic Word Immediate
X	011111	11001 11000 1	**srawi.**	Shift Right Algebraic Word Immediate & Record CR
X	011111	11010 10110 /	**mbar**	Memory Barrier
X	011111	11100 10010 ?	**tlbsx**	TLB Search Indexed
X	011111	11100 10110 /	**sthbrx**	Store Half Word Byte-Reverse Indexed
X	011111	11100 11010 0	**extsh**	Extend Sign Half Word
X	011111	11100 11010 1	**extsh.**	Extend Sign Half Word & Record CR
X	011111	11101 10010 /	**tlbre**	TLB Read Entry
X	011111	11101 11010 0	**extsb**	Extend Sign Byte
X	011111	11101 11010 1	**extsb.**	Extend Sign Byte & Record CR
X	011111	11110 01011 0	**divwuo**	Divide Word Unsigned & Record OV

Table 3-3. Instructions Sorted by Opcode (continued)

Format	Opcode		Mnemonic	Instruction
	Primary (Inst[0–5])	Extended (Inst[21–31])		
X	011111	11110 01011 1	**divwuo.**	Divide Word Unsigned & Record OV & CR
X	011111	11110 10010 /	**tlbwe**	TLB Write Entry
X	011111	11110 10110 /	**icbi**	Instruction Cache Block Invalidate
X	011111	11111 01011 0	**divwo**	Divide Word & Record OV
X	011111	11111 01011 1	**divwo.**	Divide Word & Record OV & CR
X	011111	11111 10110 /	**dcbz**	Data Cache Block set to Zero
D	100000	----- ----- -	**lwz**	Load Word & Zero
D	100001	----- ----- -	**lwzu**	Load Word & Zero with Update
D	100010	----- ----- -	**lbz**	Load Byte & Zero
D	100011	----- ----- -	**lbzu**	Load Byte & Zero with Update
D	100100	----- ----- -	**stw**	Store Word
D	100101	----- ----- -	**stwu**	Store Word with Update
D	100110	----- ----- -	**stb**	Store Byte
D	100111	----- ----- -	**stbu**	Store Byte with Update
D	101000	----- ----- -	**lhz**	Load Half Word & Zero
D	101001	----- ----- -	**lhzu**	Load Half Word & Zero with Update
D	101010	----- ----- -	**lha**	Load Half Word Algebraic
D	101011	----- ----- -	**lhau**	Load Half Word Algebraic with Update
D	101100	----- ----- -	**sth**	Store Half Word
D	101101	----- ----- -	**sthu**	Store Half Word with Update
D	101110	----- ----- -	**lmw**	Load Multiple Word
D	101111	----- ----- -	**stmw**	Store Multiple Word

Notes:

\- Don’t care, usually part of an operand field.

/ Reserved bit, invalid instruction form if encoded as 1.

? Allocated for implementation-dependent use. See user’s manual for the implementation.

Chapter 4
Instruction Pipeline and Execution Timing

This chapter describes the e200 instruction pipeline and instruction timing information. The core is partitioned into the following subsystems:

- Instruction unit
- Control unit
- Integer units
- Load/store unit
- Core interface

4.1 Overview of Operation

A block diagram of the e200z7 core is shown in Figure 4-1.

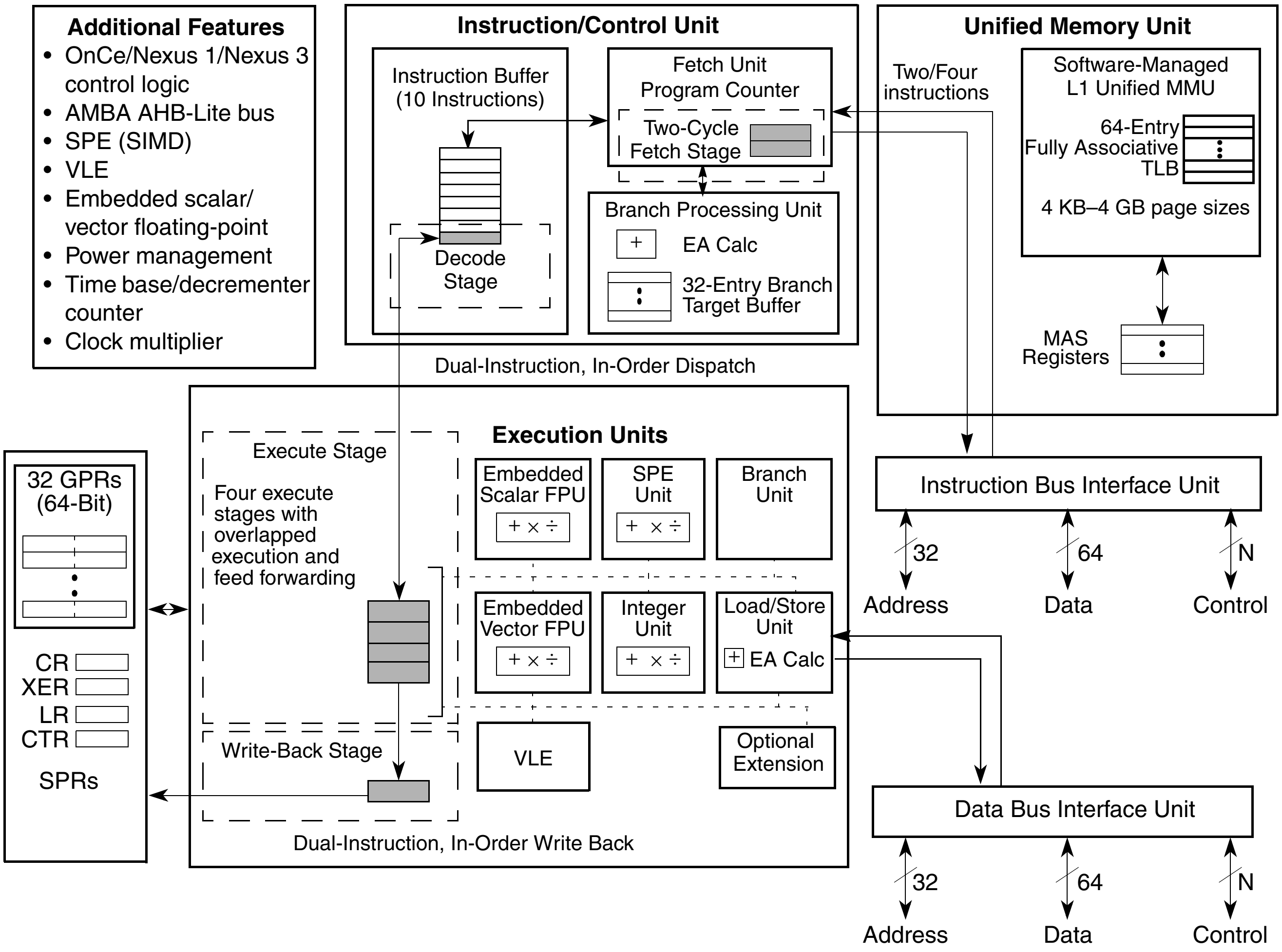


Figure 4-1. e200z7 Block Diagram

The instruction fetch unit prefetches instructions from memory into the instruction buffers. The decode unit decodes each instruction and generates information needed by the branch unit and the execution units. Prefetched instructions are written into the instruction buffers.

The instruction issue unit attempts to issue a pair of instructions each cycle to the execution units. Source operands for each of the instructions are provided from the general purpose registers (GPRs) or from the operand feed-forward muxes. Data or resource hazards may create stall conditions which cause instruction issue to be stalled for one or more cycles until the hazard is eliminated.

The execution units write the result of a finished instruction onto the proper result bus and into the destination registers. The write-back logic retires an instruction when the instruction has finished execution. Up to three results can be simultaneously written, depending on the size of the result.

Two execution units are provided to allow dual issue of most instructions. Only a single load/store unit is provided. Only a single integer divide unit is provided, thus a pair of divide instructions cannot issue simultaneously. In addition, the divide unit is blocking.

Table 4-1 shows the e200z7 concurrent instruction issue capabilities. Note that data dependencies between instructions will generally preclude dual-issue. in particular, read after write dependencies are handled by stalling the issue pipeline as required to ensure the proper execution ordering.

Table 4-1. Concurrent Instruction Issue Capabilities

Class of Instruction	Branch	Load/Store	Scalar Integer	Scalar Float	Vector Integer	Vector Float	Special
Branch	—	√	√	√	√	√	—
Load/store	√	—	√	√	√	√	—
Scalar integer	√	√	√[1]	√	√[2]	√	—
Scalar float	√	√	√	√	√	—	—
Vector integer	√	√	√[2]	√	√[3]	√	—
Vector float	√	√	√	—	√	—	—
Special	—	—	—	—	—	—	—

[1] Excludes divide class instructions occurring in both issue slots.

[2] Excludes vector MAC/multiply class instructions occurring with scalar multiply, or divide class instructions occurring in both issue slots.

[3] Excludes vector MAC/multiply class instructions occurring in both issue slots, or divide class instructions occurring in both issue slots.

4.1.1 Control Unit

The control unit coordinates the instruction fetch unit, branch unit, instruction decode unit, instruction issue unit, completion unit, and exception handling logic.

4.1.2 Instruction Unit

The instruction unit controls the flow of instructions from the cache to the instruction buffers and decode unit. Ten instruction prefetch buffers allow the instruction unit to fetch instructions ahead of actual execution, and serve to decouple memory and the execution pipeline.

4.1.3 Branch Unit

The branch unit executes branch instructions, predicts conditional branches, and provides branch target addresses for instruction fetches. It contains a 32-entry branch target buffer (BTB) to accelerate execution of branch instructions as well as a 3-entry Return Stack used for subroutine return address prediction.

4.1.4 Instruction Decode Unit

The decode unit includes the instruction buffers. A pair of instructions can be decoded each cycle. The major functions of the decode logic are:

- Opcode decoding to determine the instruction class and resource requirements for each instruction being decoded.
- Source and destination register dependency checking.
- Execution unit assignment.
- Determine any decode serializations, and inhibit subsequent instruction decoding.

The decode unit operates in a single processor clock cycle.

4.1.5 Exception Handling

The exception handling unit includes logic to handle exceptions, interrupts, and traps.

4.2 Execution Units

The core data execution units consist of the integer units, SPE units, EFPU floating-point units, and the load/store unit. Included in the execution units section are the 32- by 64-bit GPRs. Instructions with data dependencies begin execution when all such dependencies are resolved.

4.2.1 Integer Execution Units

Each integer execution unit is used to process arithmetic and logical instructions. Adds, subtracts, compares, count leading zeros, shifts and rotates execute in a single cycle. Integer multiply and divides execute in multiple clock cycles.

Multiply instructions have a latency of 3 cycles for result data and 4 cycles for condition codes for record forms, with a throughput of 1 per cycle.

Divide instructions have a variable latency (4–15 cycles) depending on the operand data. The worst case integer divide will take 15 cycles. While the divide is running, the rest of the pipeline is unavailable for additional instructions (blocking divide).

4.2.2 Load/Store Unit

The load/store unit executes instructions that move data between the GPRs and the memory subsystem. When free of data dependencies, loads execute with a maximum throughput of 1 per cycle and a 3-cycle latency. Stores also execute with a maximum throughput of 1 per cycle and a 3-cycle latency. Store data can be fed-forward from an immediately preceding load with no stall.

4.2.3 Embedded Floating-point Execution Units

The embedded floating-point execution units are used to process EFPU floating-point arithmetic instructions. Adds, subtracts, compares, multiply, and multiply-accumulate pipelines have a latency of

4 cycles with a maximum throughput of 1 per cycle. EFPU floating-point divide and square root instructions have a latency of 9 cycles. While the divide is running, the rest of the pipeline is unavailable for additional instructions (blocking divide).

4.3 Instruction Pipeline

The processor pipeline consists of stages for instruction fetch, instruction decode, register read, execution, and result writeback. Certain stages involve multiple clock cycles of execution. The processor also contains an instruction prefetch buffer to allow buffering of instructions prior to the decode stage. Instructions proceed from this buffer to the instruction decode stage by entering the instruction decode register IR.

Table 4-2 describes the pipeline stages.

Table 4-2. Pipeline Stages

Stage	Description
IFETCH0	Instruction fetch from memory, stage 0
IFETCH1	Instruction fetch from memory, stage 1
IFETCH2	Instruction fetch from memory, stage 2
DECODE0	Instruction decode, stage 0
DECODE1/RF READ	Instruction Decode, stage 1/Register read/Operand forwarding/ Memory effective address generation
EXECUTE0/MEM0	Instruction execution stage 0/Memory access stage 0
EXECUTE1/MEM1	Instruction execution stage 1/Memory access stage 1
EXECUTE2/MEM2	Instruction execution stage 2/Memory Access stage 2
EXECUTE3	Instruction execution stage 3
WB	Write back to registers

Figure 4-2 shows a pipeline diagram.

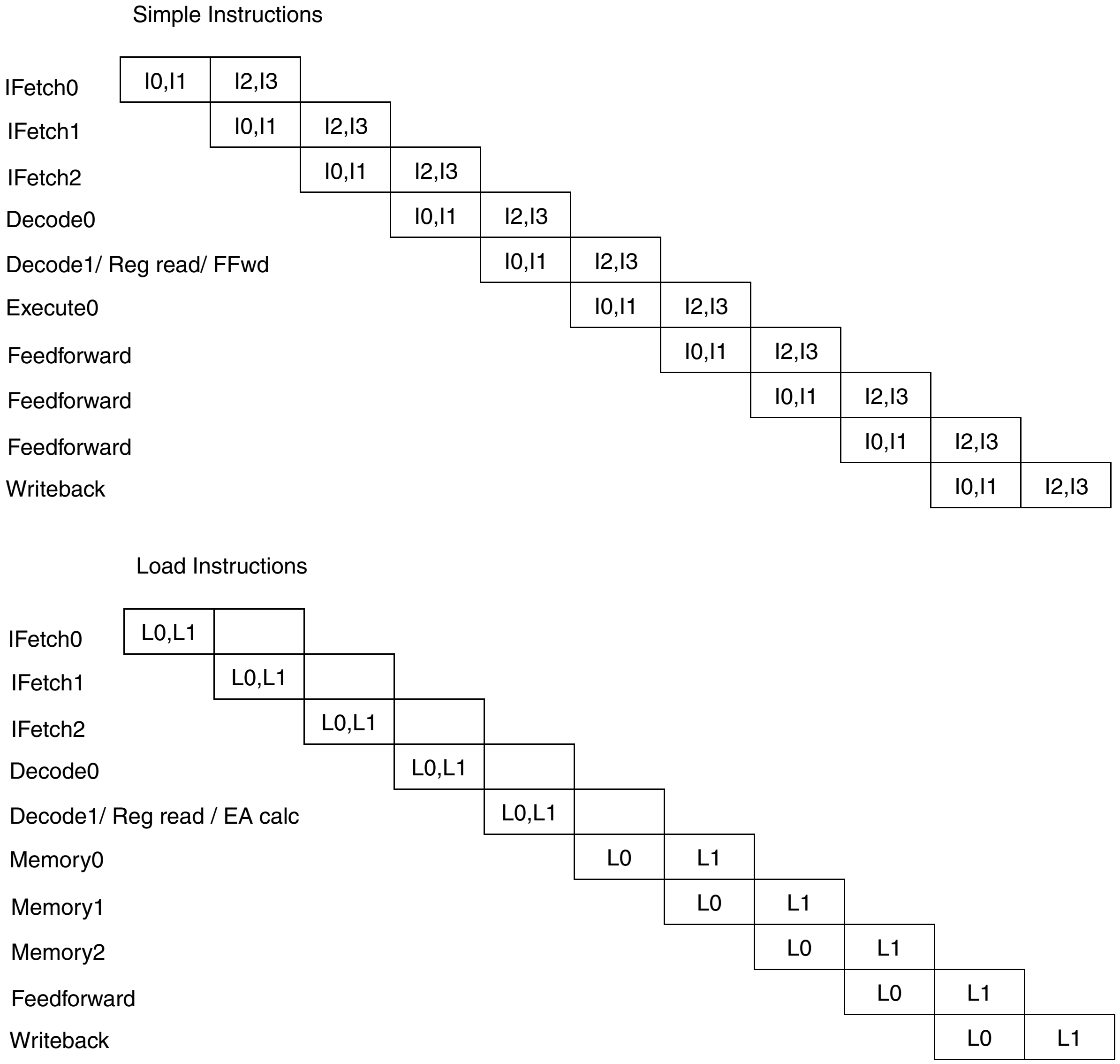


Figure 4-2. Pipeline Diagram

4.3.1 Description of Pipeline Stages

The fetch pipeline stages retrieve instructions from the memory system and determine where the next instruction fetch is performed. Up to two 32-bit instructions or four 16-bit instructions are sent from memory to the instruction buffers each cycle.

The decode pipeline stages decodes instructions, read operands from the register file, and performs dependency checking.

Execution occurs in one or more of the four execute pipeline stages in each execution unit (perhaps over multiple cycles). Execution of most load/store instructions is pipelined. The load/store unit has the following four pipeline stages:

- EA Calc—effective address calculation
- MEM0—memory access
- MEM1—memory access
- MEM2—data format and forward

Simple integer instructions complete execution in the Execute0 stage of the pipeline. Multiply instructions require all four execute stages but may be pipelined as well. Most condition-setting instructions complete in the Execute0 stage of the pipeline, thus conditional branches dependent on a condition-setting instruction may be resolved by an instruction in this stage.

Result feed-forward hardware forwards the result of one instruction into the source operand(s) of a following instruction so that the execution of data-dependent instructions do not wait until the completion of the result writeback. Feed forward hardware is supplied to allow bypassing of completed instructions from all four execute stages into the first execution stage for a subsequent data-dependent instruction.

4.3.2 Instruction Prefetch Buffers and Branch Target Buffer

The e200 contains a 10-entry instruction prefetch buffer that supplies instructions into the instruction register (IR) for decoding. Each slot in the prefetch buffer is 32 bits wide, capable of holding a single 32-bit instruction, or a pair of 16-bit instructions. Figure 4-3 shows the instruction prefetch buffers.

Instruction prefetches request a 64-bit double word and the prefetch buffer is filled with a pair of instructions at a time, except for the case of a change of flow fetch where the target is to the second (odd) word. In that case, only a 32-bit prefetch is performed to load the instruction prefetch buffer. This 32-bit fetch may be immediately followed by a 64-bit prefetch to fill slots 0 and 1 in the event that the branch is resolved to be taken.

In normal sequential execution, instructions are loaded into the IR from prefetch buffer slots 0 and 1, and as a pair of slots are emptied, they are refilled. Whenever a pair of slots is empty, a 64-bit prefetch is initiated which fills the earliest empty slot pairs beginning with slot 0.

If the instruction prefetch buffer empties, the instruction issue stalls and the buffer is refilled. The first returned instruction is forwarded directly to the IR. Open cycles on the memory bus are utilized to keep the buffer full when possible.

Figure 4-3 shows the instruction prefetch buffer.

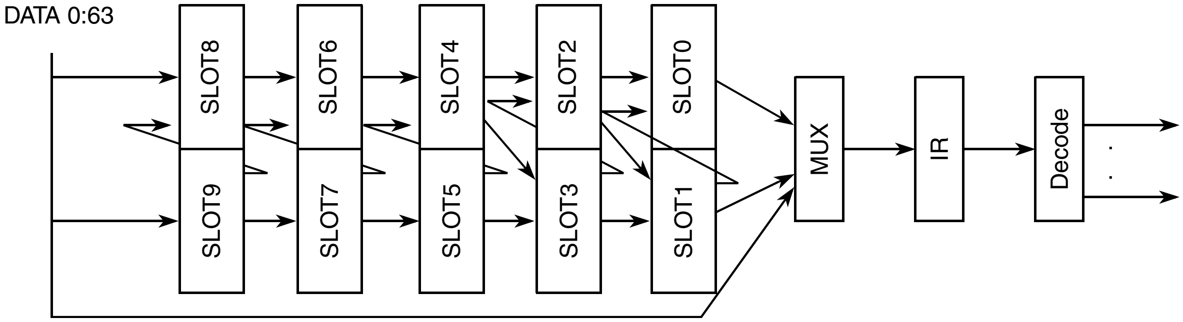


Figure 4-3. e200 Instruction Prefetch Buffers

NOTE

- The e200z7 core can prefetch up to 2 cache lines (64 bytes total) beyond the current instruction execution point. Executing code within the last 64 bytes of a memory region such as internal SRAM or Flash may cause a bus error when pre-fetching occurs past the end of memory. Do not place code to be executed within the last 64 bytes of a memory region.

- An ECC exception can occur if pre-fetches occur at locations that are valid but not yet initialized for ECC. When executing code from internal ECC SRAM, initialize memory beyond the end of the code until the next 32-byte aligned address and then an additional 64 bytes to ensure that pre-fetches cannot land in uninitialized SRAM.

- The Boot Assist Module (BAM) is located at the end of the address space and so may cause instruction pre-fetches to wrap-around to address 0 in internal flash memory. If this first block of flash memory contains ECC errors, such as from an aborted program or erase operation, a machine-check exception will be asserted. At this point in the boot procedure, exceptions are disabled, but the machine-check will remain pending and the exception vector will be taken if user application code subsequently enables the machine check interrupt. To guard against the possibility of the BAM causing a machine-check exception to be taken, user application code should write all 1s to the Machine Check Syndrome Register (MCSR) to clear it before enabling the machine check interrupt.

To resolve branch instructions and improve the accuracy of branch predictions, the e200 implements a dynamic branch prediction mechanism using a 32-entry branch target buffer (BTB).

An entry is allocated in the BTB whenever a normal branch resolves as taken and the BTB is enabled. Certain other branches do not allocate BTB entries: **bctr**, **bcctr**. Entries in the BTB are allocated on taken branches using a FIFO replacement algorithm.

Each BTB entry holds the branch target address, and a 2-bit branch history counter whose value is incremented or decremented on a BTB hit, depending on whether the branch was taken. The counter can assume four different values: strongly taken, weakly taken, weakly not taken, and strongly not taken. On initial allocation of an entry to the BTB for a taken branch, the counter is initialized to the weakly-taken state.

A branch is predicted as taken on a hit in the BTB with a counter value of strongly or weakly taken. In this case the target address contained in the BTB is used to redirect the instruction fetch stream to the target of the branch prior to the branch reaching the instruction decode stage. In the case of a BTB miss, static prediction is used to predict the outcome of the branch. In the case of a mispredicted branch, the instruction fetch stream will return to the proper instruction stream after the branch has been resolved.

When a branch is predicted taken and the branch is later resolved (in the branch execute stage), the value of the appropriate BTB counter is updated. If a branch whose counter indicates weakly taken is resolved as taken, the counter increments so that the prediction becomes strongly taken. If the branch resolves as not taken, the prediction changes to weakly not-taken. The counter saturates in the strongly taken states when the prediction is correct.

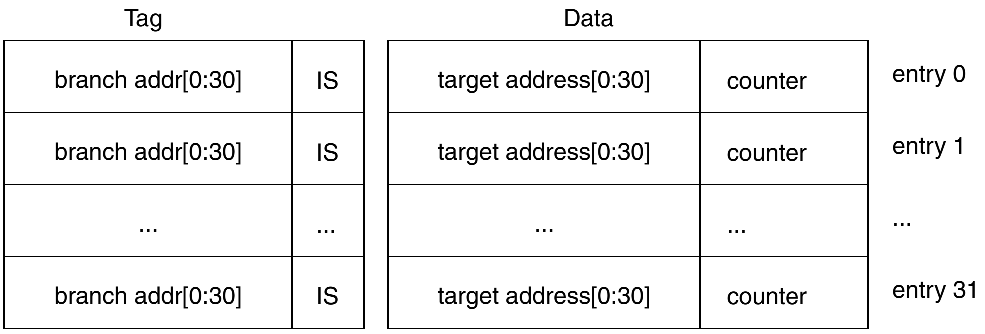
The e200 does not implement the static branch prediction that is defined by the Power ISA embedded category architecture. The BO prediction bit in branch encodings is ignored.

Dynamic branch prediction is enabled by setting BUCSR[BPEN]. Allocation of branch target buffer entries may be controlled using BUCSR[BALLOC] to control whether forward or backward branches (or both) are candidates for entry into the BTB, and thus for branch prediction. Once a branch is in the BTB, BUCSR[ALLOC] has no further effect on that branch entry. Clearing BUCSR[BPEN] disables dynamic branch prediction, in which case the e200 reverts to a static prediction mechanism using BUCSR[BPRED] to control whether forward or backward branches (or both) are predicted taken or not taken.

The BTB uses virtual addresses for performing tag comparisons. On allocation of a BTB entry, the effective address of a taken branch, along with the current Instruction Space (as indicated by MSR[IS]) is loaded into the entry and the counter value is set to weakly taken. The current PID value is not maintained as part of the tag information.

The e200 does support automatic flushing of the BTB when the current PID value is updated by a **mtcr** PID0 instruction. Software is otherwise responsible for maintaining coherency in the BTB when a change in effective to real (virtual to physical) address mapping is changed. This is supported by the BUCSR[BBFI] control bit.

Figure 4-4 shows the branch target buffer.



Tag		Data		
branch addr[0:30]	IS	target address[0:30]	counter	entry 0
branch addr[0:30]	IS	target address[0:30]	counter	entry 1
...	...	...	...	...
branch addr[0:30]	IS	target address[0:30]	counter	entry 31

IS = Instruction Space

Figure 4-4. e200 Branch Target Buffer

NOTE

Under certain conditions, if a static branch prediction and a dynamic return prediction (which uses the subroutine return address stack) occur simultaneously in the BTB, the e200z7 core can issue an errant fetch address to the memory system (instruction fetched from wrong address). This can only happen when the static branch prediction is "taken" but the branch actually resolves to "not taken". To prevent the issue from occurring, set BUCSR[BPRED] = 0b11 to configure static branch prediction to "not taken". This issue does not apply to VLE.

4.3.3 Single-Cycle Instruction Pipeline Operation

Sequences of single-cycle execution instructions follow the flow shown in Figure 4-5. Instructions are issued and completed in program order. Most arithmetic and logical instructions fall into this category.

Time Slot →

1st Inst(s).	IF0	IF1	IF2	D0	D1/ RR	E0	FF	FF	FF	WB			
2nd Inst(s).		IF0	IF1	IF2	D0	D1/ RR	E0	FF	FF	FF	WB		
3rd Inst(s).			IF0	IF1	IF2	D0	D1/ RR	E0	FF	FF	FF	WB	
4th Inst(s).				IF0	IF1	IF2	D0	D1/ RR	E0	FF	FF	FF	WB

Figure 4-5. Basic Pipe Line Flow, Single Cycle Instructions

4.3.4 Basic Load and Store Instruction Pipeline Operation

Figure 4-6 shows the basic pipeline flow for load and store instructions. The effective address is calculated in the EA Calc stage, and memory is accessed in the MEM0–MEM1 stages. Data selection and alignment is performed in MEM2, and the result is available at the end of MEM2.

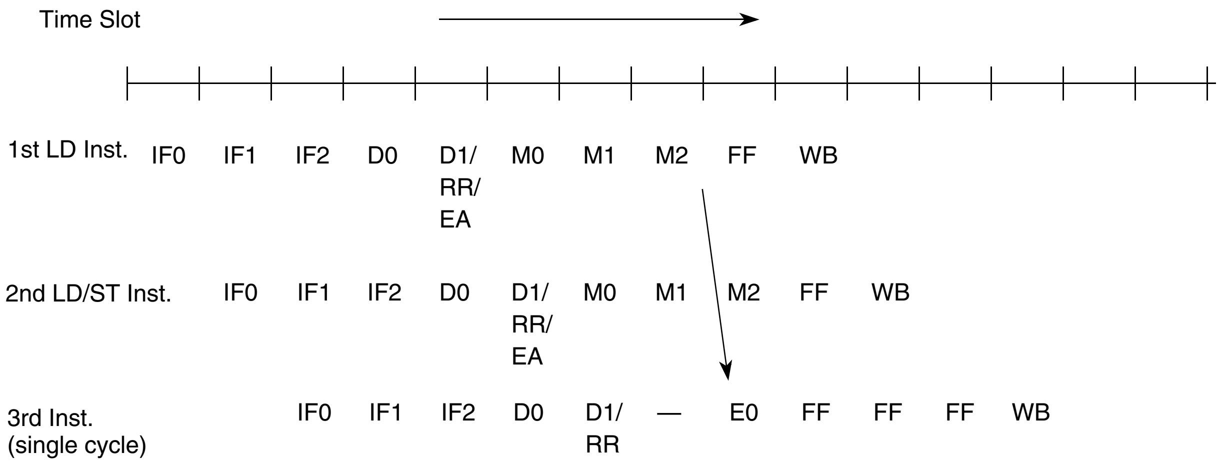


Figure 4-6. Basic Pipeline Flow, Load/Store Instructions

4.3.5 Change-of-Flow Instruction Pipeline Operation

Figure 4-7 shows a pipeline flow with no prediction. Simple change of flow instructions require 4 cycles to refill the pipeline with the target instruction for taken branches and branch and link instructions with no BTB hit and no prediction required (condition resolved prior to branch decode).

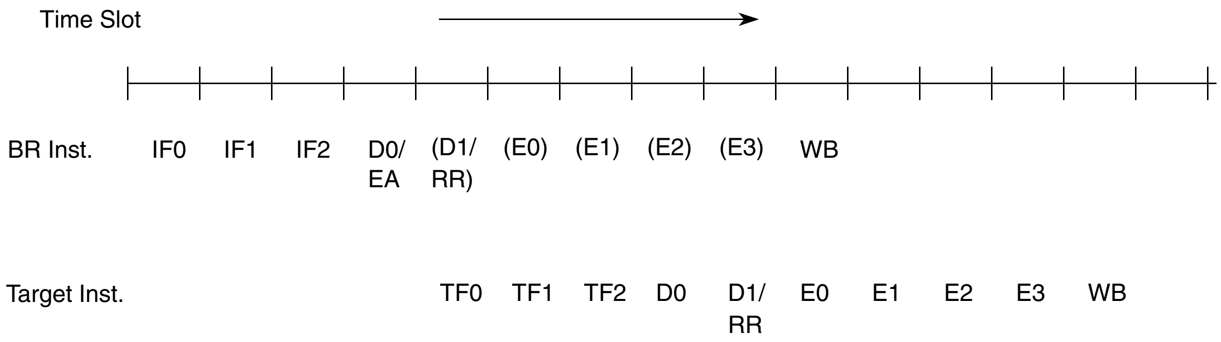


Figure 4-7. Basic Pipeline Flow, Branch Instructions, No Prediction

Figure 4-7 shows a pipeline flow with correct prediction and a branch taken. For branch type instructions, this 4-cycle timing may be reduced in some situations by performing the target fetch speculatively while the branch instruction is still being fetched into the instruction buffer if the branch target address can be obtained from the BTB. The resulting branch timing reduces to a single clock when the target fetch is initiated early enough and the branch is correctly predicted.

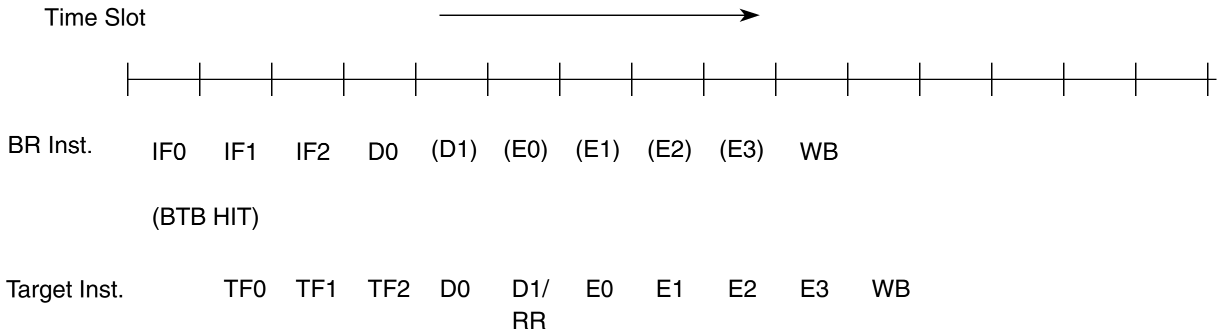


Figure 4-8. Basic Pipeline Flow, Branch Instructions, BTB Hit, Correct Prediction, Branch Taken

Figure 4-9 shows a case where the branch is incorrectly predicted, and 6 cycles are required to correct the misprediction outcome.

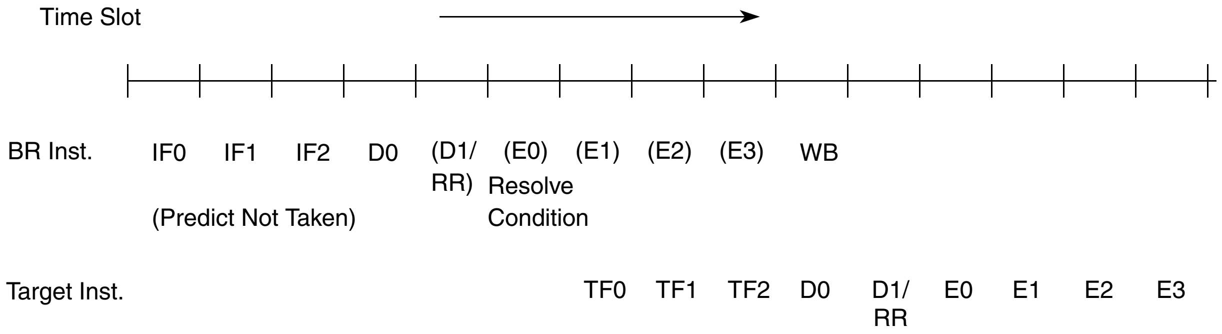


Figure 4-9. Basic Pipeline Flow, Branch Instructions, Predict Not Taken, Incorrect Prediction

Figure 4-10 shows **bcctr** and **e_bctr** cases where the branch is correctly predicted as taken, and 5 cycles are required to execute the branch.

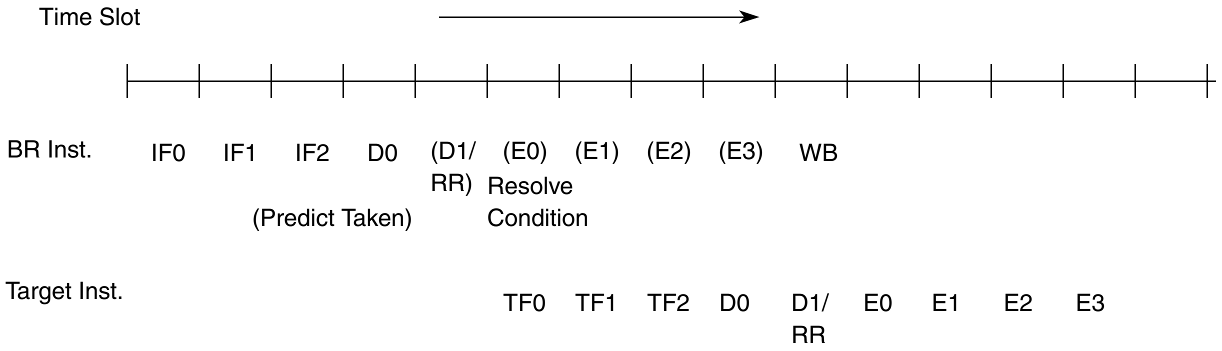


Figure 4-10. Basic Pipeline Flow, bcctr Instruction, Predict Taken, Incorrect Prediction, Instruction Buffer Not Empty

Figure 4-11 shows the **bcctr** and **e_bctr** cases where the branch is incorrectly predicted as taken, but the fall-through instruction is already in the instruction buffer. 3 cycles are required to execute this branch.

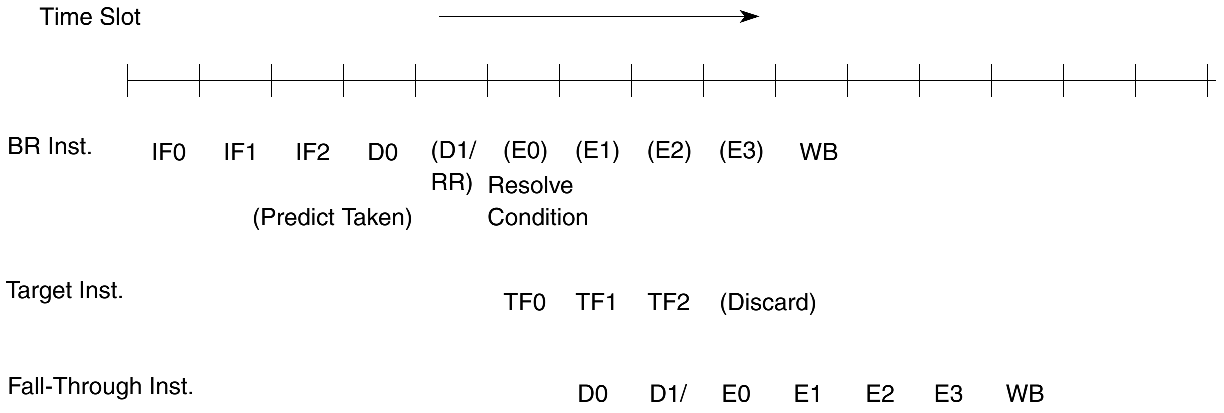


Figure 4-11. Basic Pipeline Flow, bcctr Instruction, Predict Taken, Incorrect Prediction, Instruction Buffer Not Empty

Figure 4-12 shows **bcctr** and **e_bctr** cases where the branch is incorrectly predicted as taken, and the fall-through instruction is not already in the instruction buffer (a rare case). 6 cycles are required to execute this branch.

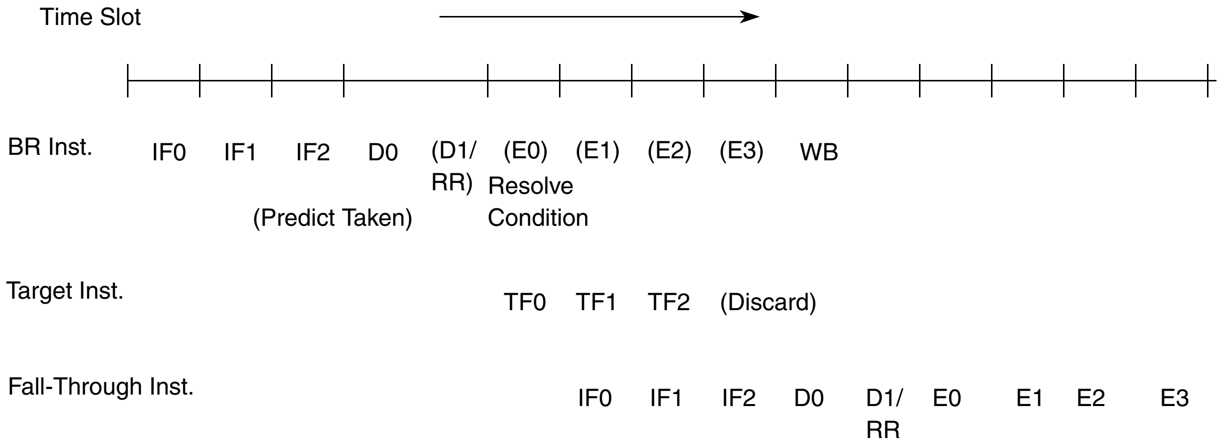
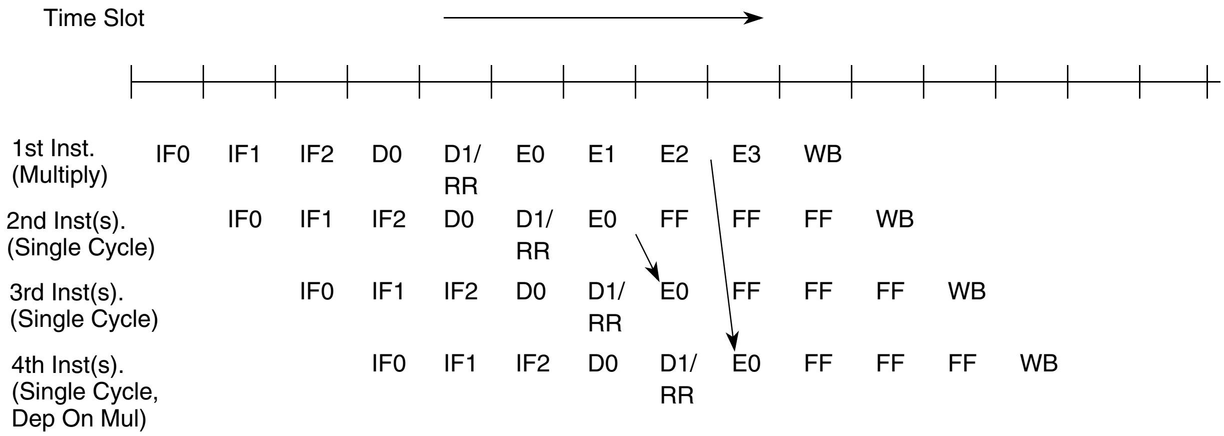


Figure 4-12. Basic Pipelne Flow, bcctr Instruction, Predict Taken, Incorrect Prediction, Instruction Buffer Empty

4.3.6 Basic Multicycle Instruction Pipeline Operation

Most multicycle instructions may be pipelined so that the effective execution time is smaller than the overall number of clocks spent in execution. The restrictions to this execution overlap are that no data dependencies between the instructions can be present and that instructions must complete and write back results in order. A single cycle instruction that follows a multicycle instruction must wait for completion of the multicycle instruction prior to its writeback in order to meet the in-order requirement. Result feed-forward paths are provided so that execution may continue prior to result writeback.

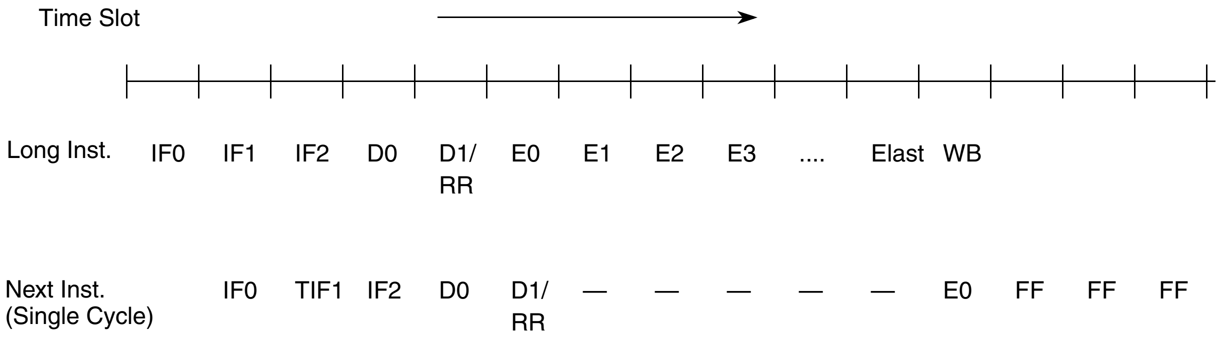
Figure 4-13 shows the basic pipeline flow for integer multiply class instructions.



1st Inst. (Multiply)	IF0	IF1	IF2	D0	D1/ RR	E0	E1	E2	E3	WB				
2nd Inst(s). (Single Cycle)		IF0	IF1	IF2	D0	D1/ RR	E0	FF	FF	FF	WB			
3rd Inst(s). (Single Cycle)			IF0	IF1	IF2	D0	D1/ RR	E0	FF	FF	FF	WB		
4th Inst(s). (Single Cycle, Dep On Mul)				IF0	IF1	IF2	D0	D1/ RR	E0	FF	FF	FF	WB	

Figure 4-13. Basic Pipeline Flow, Integer Multiply Class Instructions

The divide and load and store multiple instructions require multiple cycles in the execute stage, as shown in Figure 4-14.



Long Inst.	IF0	IF1	IF2	D0	D1/ RR	E0	E1	E2	E3	....	Elast	WB			
Next Inst. (Single Cycle)		IF0	TIF1	IF2	D0	D1/ RR	—	—	—	—	—	E0	FF	FF	FF

Figure 4-14. Basic Pipeline Flow, Long Instruction

4.3.7 Additional Examples of Instruction Pipeline Operation for Load and Store

Figure 4-15 shows an example of pipelining two non-data-dependent load or store instructions with a following load target data-dependent single cycle instruction. While the first load or store begins accessing memory in the M0 stage, the next load can be calculating a new effective address in the D1/EA stage. The

add in this example stalls for 2 cycles since a data dependency exists on the target register of the second load.

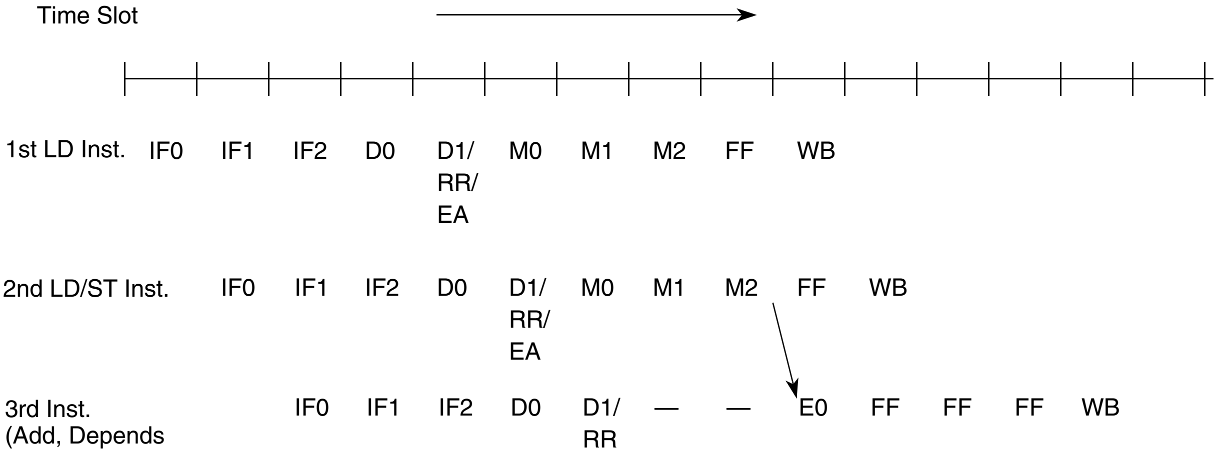


Figure 4-15. Pipelined Load Instructions with Load Target Data Dependency

Figure 4-16 shows an example of pipelining a data-dependent add instruction following a load with update instruction. While the first load begins accessing memory in the M0 stage, the next load with update can be calculating a new effective address in the EA Calc stage. Following the EA Calc, the updated base register value can be fed-forward to subsequent instructions. The **add** in this example does not stall, even though a data dependency exists on the updated base register of the load with update.

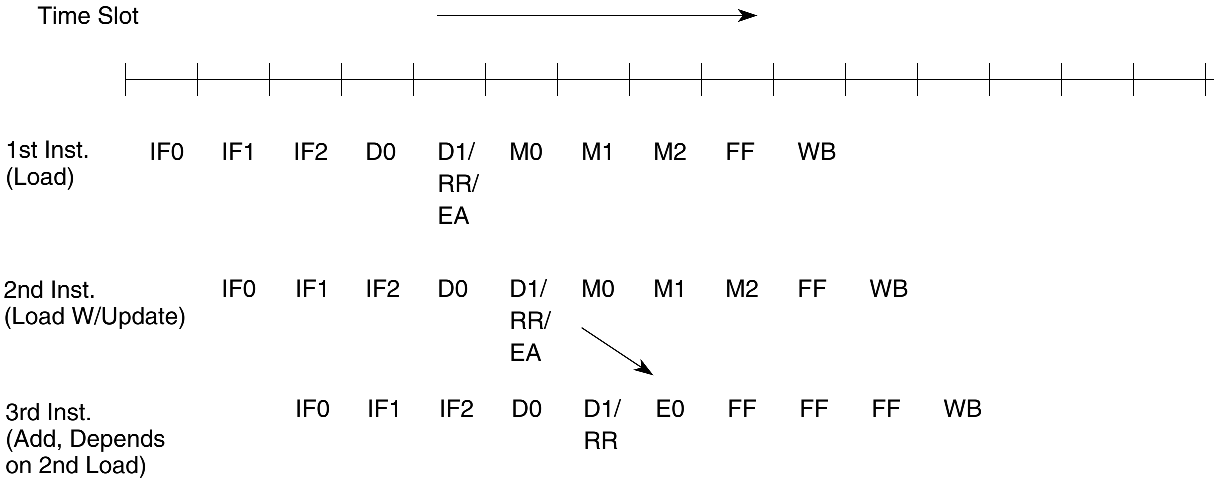


Figure 4-16. Pipelined Instructions with Base Register Update Data Dependency

Figure 4-17 shows an example of pipelining a data-dependent store instruction following a load instruction. While the first load begins accessing memory in the M0 stage, the store can be calculating a new effective address in the D1/EA stage. The **store** in this example will not stall due to the data dependency existing on the load data of the load instruction.

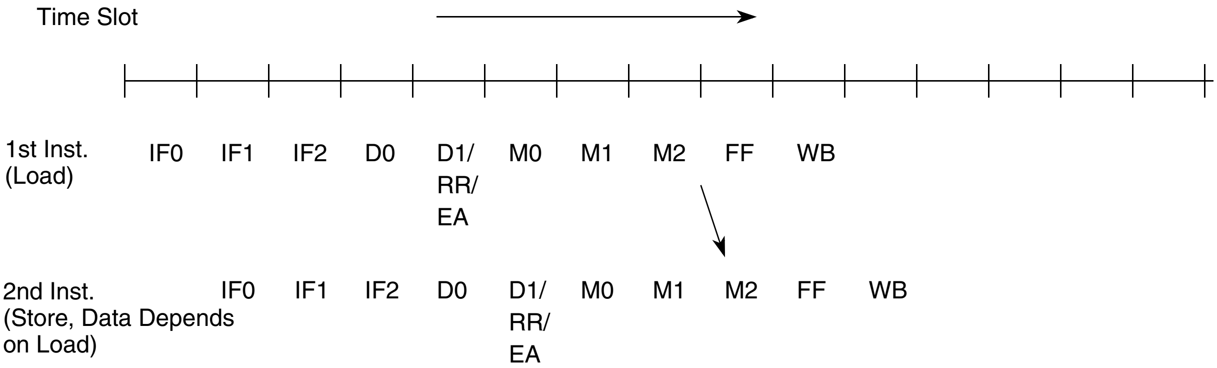


Figure 4-17. Pipelined Store Instruction with Store Data Dependency

4.3.8 Move to/from SPR Instruction Pipeline Operation

Many **mtspr** and **mfspr** instructions are treated like single cycle instructions in the pipeline, and do not cause stalls. Exceptions are for the MSR, the debug SPRs, the SPE unit, and cache/MMU SPRs which do cause stalls. Figure 4-18 through Figure 4-20 show examples of **mtspr** and **mfspr** instruction timing.

Figure 4-18 applies to the debug SPRs and the SPE unit's SPEFSCR. These instructions do not begin execution until all previous instructions have finished their execute stage(s). In addition, execution of subsequent instructions is stalled until the **mfspr** and **mtspr** instructions complete.

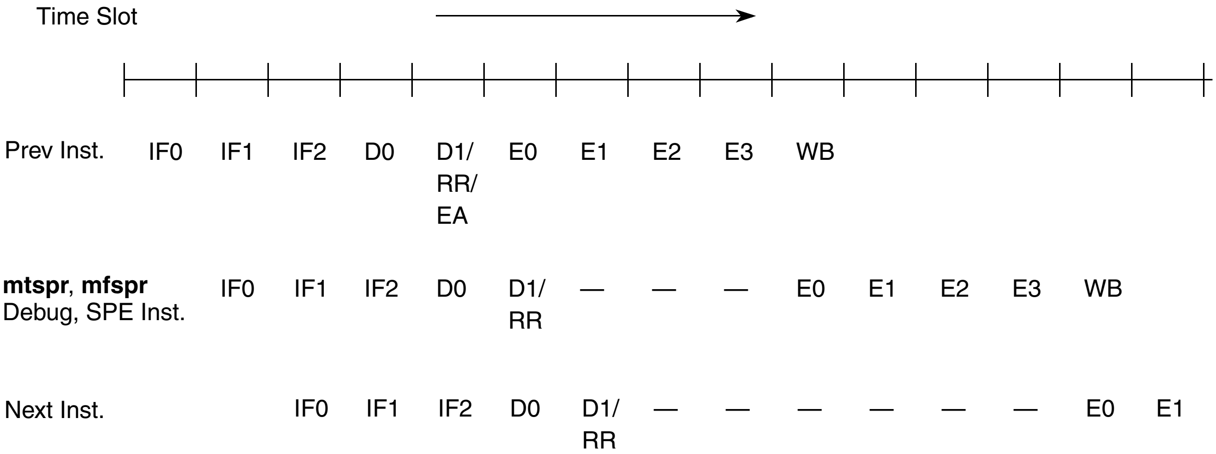
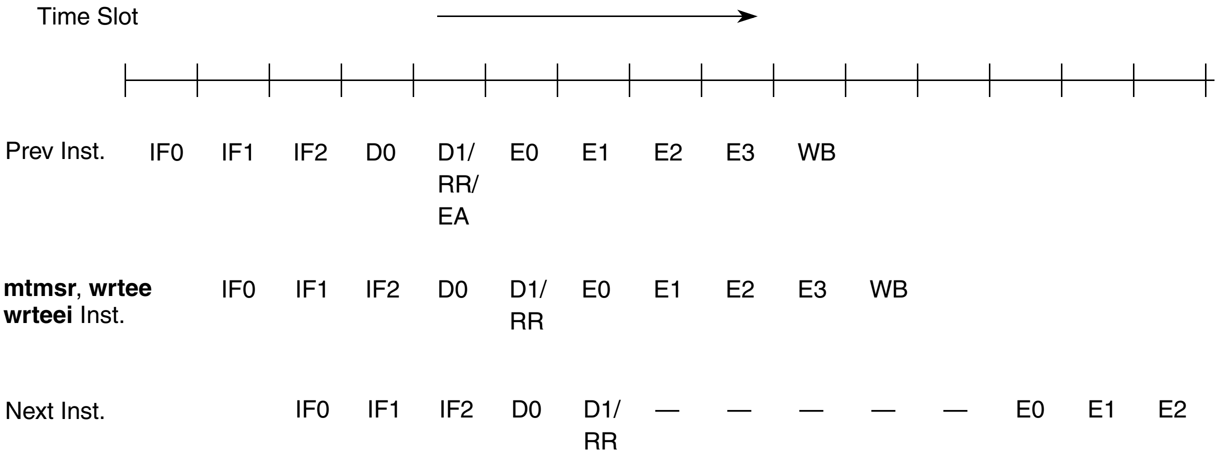


Figure 4-18. mtspr, mfspr Instruction Execution, Debug, and SPE SPRs

Figure 4-19 applies to the **mtmsr** instruction and the **wrtee** and **wrteei** instructions. Execution of subsequent instructions is stalled until the cycle after these instructions writeback.



Prev Inst.	IF0	IF1	IF2	D0	D1/ RR/ EA	E0	E1	E2	E3	WB					
mtmsr, **wrtee** **wrteei** Inst.		IF0	IF1	IF2	D0	D1/ RR	E0	E1	E2	E3	WB				
Next Inst.			IF0	IF1	IF2	D0	D1/ RR	—	—	—	—	—	E0	E1	E2

Figure 4-19. mtmsr, wrtee[i] Instruction Execution

Access to cache and MMU SPRs are stalled until all outstanding bus accesses have completed on both interfaces and the caches and MMU are idle (*p_[d,i]_cmbusy* negated) to allow an access window where no translations or cache cycles are required. Figure 4-20 shows an example where an outstanding bus access causes **mtspr**/**mfspr** execution to be delayed until the bus becomes idle. Other situations such as a cache linefill may cause the cache to be busy even when the processor interface is idle (*p_[d,i]_tbusy[0]_b* is negated). In these cases execution stalls until the cache and MMU are idle as signaled by negation of *p_[d,i]_cmbusy*. Processor access requests will be held off during execution of a cache/MMU SPR instruction. A subsequent access request may be generated the cycle following the last execute stage (that is, during the WB cycle). This same protocol applies to cache and MMU management instructions (for example, **dcbz**, **dcbf**, etc., **tlbre**, **tlbwe**, etc.).

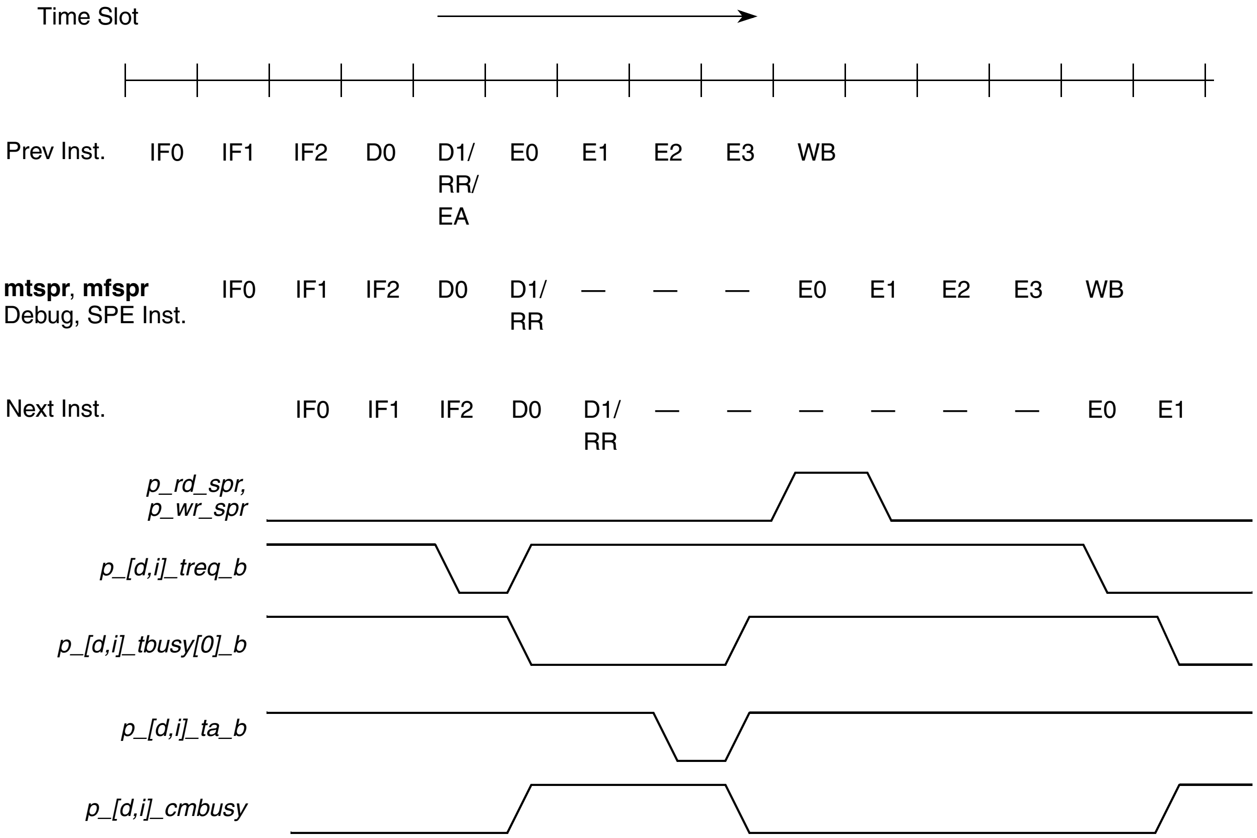


Figure 4-20. Cache/MMU mtspr, mfspr, and Management Instruction Execution

4.4 Control Hazards

Internal control hazards exist in the e200 that can cause certain instruction sequences to incur one or more stall cycles. For example, when an **mtspr** instruction precedes an **mfspr** instruction, the issue stalls until the **mtspr** completes.

4.5 Instruction Serialization

There are three types of serialization required by the core, as follows:

- Completion serialization
- Dispatch (decode/issue) serialization
- Refetch serialization

4.5.1 Completion Serialization

A completion serialized instruction is held for execution until all prior instructions have completed. The instruction will then execute once it is next to complete in program order. Results from these instructions will not be available for or forwarded to subsequent instructions until the instruction completes. Instructions which are completion serialized are as follows:

- Instructions that access or modify system control or status registers. For example, **mcrxr**, **mtmsr**, **wrtee**, **wrteei**, **mtspr, mfspr** (except to CTR/LR).
- Instructions that manage caches and TLBs
- Instructions defined by the architecture as context or execution synchronizing: **isync**, **se_isync**, **msync**, **rfi**, **rfci**, **rfdi**, **rfmci**, **se_rfi**, **se_rfci**, **se_rfdi**, **se_rfmci**, **sc**, **se_sc**.
- **wait**

4.5.2 Dispatch Serialization

Some instructions are dispatch-serialized by the core. An instruction that is dispatch-serialized prevents the next instruction from decoding until all instructions up to and including the dispatch-serialized instruction completes. Instructions which are dispatch serialized are: **isync**, **se_isync**, **msync**, **rfi**, **rfci**, **rfdi**, **rfmci**, **se_rfi**, **se_rfci**, **se_rfdi**, **se_rfmci**, **sc**, **se_sc**.

The **mbar** instruction is pseudo-dispatch serialized; it prevents the next instruction from decoding until all previous load and store class instructions have completed.

4.5.3 Refetch Serialization

Refetch serialized instructions inhibit dispatching of subsequent instructions and force a pipeline refill to refetch subsequent instructions after completion. These include the following:

- The context synchronizing instructions **isync**, **se_isync**.
- The **rfi**, **rfci**, **rfdi**, **rfmci**, **se_rfi**, **se_rfci**, **se_rfdi**, **se_rfmci**, **sc**, **se_sc** instructions.

4.6 Concurrent Instruction Execution

The core effectively has the following execution units:

- Branch unit
- Dual scalar integer units
- Dual vector integer units
- Dual scalar embedded floating-point units/single vector embedded floating-point unit
- Load/store unit

These executions units are pipelined and support overlapped execution of instructions. In certain cases, the branch unit predicts branches and supplies a speculative instruction stream to the instruction buffer unit.

Section 4.7, “Instruction Timings,” accurately indicates the number of cycles an instruction executes in the appropriate unit. However, determining the elapsed time or cycles to execute a sequence of instructions is beyond the scope of this document.

4.7 Instruction Timings

Instruction timing in number of processor clock cycles for various instruction classes is shown in Table 4-3. Pipelined instructions are shown with cycles of total latency and throughput cycles. Divide instructions are not pipelined and block other instructions from executing during divide execution. Timing for SPE instructions is detailed in Section 6.3, “SPE Instruction Timing.”

Load/store multiple instruction cycles are represented as a fixed number of cycles plus a variable number of cycles where *n* is the number of words accessed by the instruction. In addition, cycle times marked with an ‘&’ require variable number of additional cycles due to serialization.

Table 4-3. Instruction Class Cycle Counts

Class of Instructions	Latency	Throughput	Special Notes
Integer: **add**, **sub**, **shift**, **rotate**, **logical**, **cntlzw**	1	1	—
Integer: compare	1	1	—
Branch	6/4/1	6/4/1	Correct branch lookahead allows single cycle execution. Worst-case mispredicted branch is 6 cycles.
Multiply	3/4	1	Result data is available after 3 cycles, record form conditions are available after fourth cycle.
Divide	4–15	4–15	Data dependent timing
CR logical	1	1	—
Loads (non-multiple)	3	1	—
Load multiple	3 + *n*/2 (max)	1 + *n*/2 (max)	Actual timing depends on *n* and address alignment.
Stores (non-multiple)	3	1	—
Store multiple	3 + *n*/2 (max)	1 + *n*/2 (max)	Actual timing depends on *n* and address alignment.
mtmsr, **wrtee**, **wrteei**	6&	6	—
mcrf	1	1	—
mfspr, **mtspr**	4&	4&	applies to debug SPRs, optional unit SPRS
mfspr, **mfmsr**	1	1	Applies to internal, non-debug SPRs
mfcr, **mtcr**	1	1	—
rfi, **rfci**, **rfdi**, **rfmci**	6	—	—
sc	4	—	—
tw, **twi**	4	—	Trap taken timing

Table 4-4 shows detailed timing for each instruction mnemonic along with its serialization requirements.

Table 4-4. Instruction Timing by Mnemonic

Mnemonic	Latency	Serialization
add[**o**][.]	1	None
addc[**o**][.]	1	None
adde[o][.]	1	None
addi	1	None
addic[.]	1	None
addis	1	None
addme[**o**][.]	1	None
addze[**o**][.]	1	None
and[.]	1	None
andc[.]	1	None
andi.	1	None
andis.	1	None
b[**l**][**a**]	6/4/1	None
bc[**l**][**a**]	6/4/1	None
bcctr[**l**]	6/5/3/1	None
bclr[**l**]	6/5/3/1	None
cmp	1	None
cmpi	1	None
cmpl	1	None
cmpli	1	None
cntlzw[.]	1	None
crand	1	None
crandc	1	None
creqv	1	None
crnand	1	None
crnor	1	None
cror	1	None
crorc	1	None
crxor	1	None
divw[**o**][.]	4–15[1]	None
divwu[**o**][.]	4–15[1]	None
eqv[.]	1	None

Table 4-4. Instruction Timing by Mnemonic (continued)

Mnemonic	Latency	Serialization
extsb[.]	1	None
extsh[.]	1	None
isel	1	None
isync	6[2]	Refetch
lbarx	3	None
lbz	3[3]	None
lbzu	3[3]	None
lbzux	3[3]	None
lbzx	3[3]	None
lha	3[3]	None
lharx	3	None
lhau	3[3]	None
lhaux	3[3]	None
lhax	3[3]	None
lhbrx	3[3]	None
lhz	3[3]	None
lhzu	3[3]	None
lhzux	3[3]	None
lhzx	3[3]	None
lmw	3 + (*n*/2)	None
lwarx	3	None
lwbrx	3[3]	None
lwz	3[3]	None
lwzu	3[3]	None
lwzux	3[3]	None
lwzx	3[3]	None
mbar	1[2]	Pseudo-dispatch
mcrf	1	None
mcrxr	1	Completion
mfcr	1	None
mfmsr	1	None
mfspr (except DEBUG)	1	None

Table 4-4. Instruction Timing by Mnemonic (continued)

Mnemonic	Latency	Serialization
mfspr (DEBUG)	3^2	Completion
msync	1^2	Completion
mtcrf	2	None
mtmsr	6^2	Completion
mtspr (DEBUG)	4^2	Completion
mtspr (except DEBUG, **msr**, hid0/1)	1	None
mulhw[.]	3/4	None
mulhwu[.]	3/4	None
mulli	3/4	None
mullw[**o**][.]	3/4	None
nand[.]	1	None
neg[**o**][.]	1	None
nop (**ori** r0,r0,0)	1	None
nor[.]	1	None
or[.]	1	None
orc[.]	1	None
ori	1	None
oris	1	None
rfci	6	Refetch
rfdi	6	Refetch
rfi	6	Refetch
rfmci	6	Refetch
rlwimi[.]	1	None
rlwinm[.]	1	None
rlwnm[.]	1	None
sc	4	Refetch
slw[.]	1	None
sraw[.]	1	None
srawi[.]	1	None
srw[.]	1	None
stb	3^3	None
stbcx.	3	None
stbu	3^3	None

Table 4-4. Instruction Timing by Mnemonic (continued)

Mnemonic	Latency	Serialization
stbux	3^3	None
stbx	3^3	None
sth	3^3	None
sthbrx	3^3	None
sthcx.	3	None
sthu	3^3	None
sthux	3^3	None
sthx	3^3	None
stmw	3 + (*n*/2)	None
stw	3^3	None
stwbrx	3^3	None
stwcx.	3	None
stwu	3^3	None
stwux	3^3	None
stwx	3^3	None
subf[**o**][**.**]	1	None
subfc[**o**][**.**]	1	None
subfe[**o**][**.**]	1	None
subfic	1	None
subfme[**o**][**.**]	1	None
subfze[**o**][**.**]	1	None
tw	4	None
twi	4	None
wrtee	6	Completion
wrteei	6	Completion
xor[**.**]	1	None
xori	1	None
xoris	1	None

[1] With early-out capability, timing is data dependent.

[2] Plus additional synchronization time.

[3] Aligned.

4.8 Operand Placement on Performance

The placement (location and alignment) of operands in memory affects relative performance of memory accesses, and in some cases, affects it significantly. Table 4-5 indicates the effects for the e200 core.

In Table 4-5, optimal means that one effective address (EA) calculation occurs during the memory operation. Good means that multiple EA calculations occur during the memory operation which may cause additional bus activities with multiple bus transfers. Poor means that an alignment interrupt is generated by the storage operation.

Table 4-5. Performance Effects of Storage Operand Placement

Operand		Boundary Crossing		
Size	**Byte Align.**	**None**	**Cache Line**	**Protection Boundary**
4 Bytes	4 <4	Optimal Good	— Good	— Good
2 Bytes	2 <2	Optimal Good	— Good	— Good
1 Byte	1	Optimal	—	—
lmw, **stmw**	4 <4	Good Poor	Good Poor	Good Poor
string	N/A	—	—	—

Note:
Optimal: One EA calculation occurs.
Good: Multiple EA calculations occur which may cause additional bus activities with multiple bus transfers.
Poor: Alignment Interrupt occurs.

Chapter 5
Embedded Floating-Point Unit

This chapter describes the instruction set architecture of the embedded floating-point unit version 2 (EFPU) implemented on the e200z7. This unit implements scalar and vector single-precision floating-point instructions to accelerate signal processing and other algorithms. In comparison to version 1.1 of the EFPU architecture, version 2 of the architecture implements additional operations such as minimum, maximum, and square root, as well as an extensive set of vector operations with permuted operands and mixed add/sub, sum, and differences. For the remainder of this chapter, the term EFPU implies version 2 of the architecture unless otherwise noted.

5.1 Nomenclature and Conventions

The following conventions regarding nomenclature are used in this chapter:

- Bits 0 to 31 of a 64-bit register are referenced as field 0, upper half, or high-order element of the register. Bits 32–63 are referred to as field 1, lower half, or lower-order element of the register. Each half is an element of a GPR.
- Mnemonics for EFPU instructions begin with the letters ‘evfs’ (embedded vector floating single) or ‘efs’ (embedded (scalar) floating single).

5.2 EFPU Programming Model

The e200z7 core provides a register file with thirty-two 64-bit registers. The Power ISA embedded category 32-bit instructions operate on the lower (least significant) 32 bits of the 64-bit register. EFPU instructions are defined that view the 64-bit register as being composed of a vector of two 32-bit elements, or a single scalar 32-bit element. Vector floating-point instructions operate on a vector of two 32-bit single-precision floating-point numbers resident in the 64-bit GPRs. Scalar single-precision floating-point instructions operate on the lower half of GPRs. The floating-point instructions do not have a separate register file; there is a single shared register file for all instructions.

There are no record forms of EFPU instructions. EFPU compare instructions store the result of the comparison into the condition register (CR). The meaning of the CR bits are now overloaded for the vector operations. Floating-point compare instructions treat NaNs, Infinity and Denorm as normalized numbers for the comparison calculation when default results are provided.

5.2.1 Signal Processing Extension/Embedded Floating-Point Status and Control Register (SPEFSCR)

Status and control for embedded floating-point uses the SPEFSCR register. This register is also used by the SPE unit. Status and control bits are shared for vector floating-point operations, scalar floating-point operations and SPE vector operations. The SPEFSCR register is implemented as special purpose register (SPR) number 512 and is read and written by the **mfspr** and **mtspr** instructions. The SPEFSCR is shown in Figure 5-1.

SPR 512 Access: Read/Write

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R W	SOVH	OVH	FGH	FXH	FINVH	FDBZH	FUNFH	FOVFH	—	RM	FINXS	FINVS	FDBZS	FUNFS	FOVFS	MODE
Reset	All zeros															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R W	SOV	OV	FG	FX	FINV	FDBZ	FUNF	FOVF	—	FINXE	FINVE	FDBZE	FUNFE	FOVFE	FRMC	
Reset	All zeros															

Figure 5-1. SPE/EFPU Status and Control Register (SPEFSCR)

The SPEFSCR bits are defined in Table 5-1.

Table 5-1. SPE /EFPU Status and Control Register

Bits	Name	Description
0 (32)	SOVH	Summary Integer Overflow High. Defined by SPE.
1 (33)	OVH	Integer Overflow High. Defined by SPE.
2 (34)	FGH	Embedded Floating-Point Guard Bit High FGH is supplied for use by the floating-point round exception handler. FGH is zeroed if a floating-point data exception occurs for the high element(s). FGH corresponds to the high element result. FGH is cleared by a scalar floating-point instruction.
3 (35)	FXH	Embedded Floating-Point Sticky Bit High FXH is supplied for use by the Floating-point round exception handler. FXH is zeroed if a floating-point data exception occurs for the high element(s). FXH corresponds to the high element result. FXH is cleared by a scalar floating-point instruction.
4 (36)	FINVH	Embedded Floating-Point Invalid Operation/Input Error High In mode 0, the FINVH bit is set to 1 if the A or B high element operand of a floating-point instruction is Infinity, NaN, or Denorm, or if the operation is a divide and the high element dividend and divisor are both 0. In mode 1, the FINVH bit is set on an IEEE Std 754 invalid operation (see IEEE 754-1985, Section 7.1) in the high element. FINVHH is cleared by a scalar floating-point instruction.
5 (37)	FDBZH	Embedded Floating-Point Divide by Zero High The FDBZH bit is set to 1 when a floating-point divide instruction executed with a high element divisor of 0, and the high element dividend is a finite non-zero number. FDBZH is cleared by a scalar floating point instruction.

Table 5-1. SPE /EFPU Status and Control Register (continued)

Bits	Name	Description
6 (38)	FUNFH	Embedded Floating-Point Underflow High The FUNFH bit is set to 1 when the execution of a floating-point instruction results in an underflow in the high element. FUNFH is cleared by a scalar floating-point instruction.
7 (39)	FOVFH	Embedded Floating-Point Overflow High The FOVFH bit is set to 1 when the execution of a floating-point instruction results in an overflow in the high element. FOVFH is cleared by a scalar floating-point instruction.
8– (40–)	—	Reserved
9 (41)	RM	Rounding Mode—Fixed point Defined by SPE
10 (42)	FINXS	Embedded Floating-Point Inexact Sticky Flag The FINXS bit is set to 1 whenever the execution of a floating-point instruction delivers an inexact result for either the low or high element and no floating-point data exception is taken for either element, or if the result of a floating-point instruction results in overflow (FOVF = 1 or FOVFH = 1), but floating-point overflow exceptions are disabled (FOVFE = 0), or if the result of a floating-point instruction results in underflow (FUNF=1 or FUNFH=1), but floating-point underflow exceptions are disabled (FUNFE = 0), and no floating-point data exception occurs. The FINXS bit remains set until it is cleared by a **mtspr** instruction specifying the SPEFSCR register.
11 (43)	FINVS	Embedded Floating-Point Invalid Operation Sticky Flag The FINVS bit is set to a 1 when a floating-point instruction sets the FINVH or FINV bit to 1. The FINVS bit remains set until it is cleared by a **mtspr** instruction specifying the SPEFSCR register.
12 (44)	FDBZS	Embedded Floating-Point Divide by Zero Sticky Flag The FDBZS bit is set to 1 when a floating-point divide instruction sets the FDBZH or FDBZ bit to 1. The FDBZS bit remains set until it is cleared by a **mtspr** instruction specifying the SPEFSCR register.
13 (45)	FUNFS	Embedded Floating-Point Underflow Sticky Flag The FUNFS bit is set to 1 when a floating-point instruction sets the FUNFH or FUNF bit to 1. The FUNFS bit remains set until it is cleared by a **mtspr** instruction specifying the SPEFSCR register.
14 (46)	FOVFS	Embedded Floating-Point Overflow Sticky Flag The FOVFS bit is set to 1 when a floating-point instruction sets the FOVFH or FOVF bit to 1. The FOVFS bit remains set until it is cleared by a **mtspr** instruction specifying the SPEFSCR register.
15 (47)	MODE	Embedded Floating-Point Operating Mode 0 Default hardware results operating mode 1 IEEE 754 standard hardware results operating mode (not supported by the e200) This bit controls the operating mode of the EFPU. The e200 supports only mode 0. Software should read the value of this bit after writing it to determine if the implementation supports the selected mode. Implementations will return the value written if the selected mode is a supported mode, otherwise the value read will indicate the hardware supported mode.
16 (48)	SOV	Summary Integer Overflow. Defined by SPE.
17 (49)	OV	Integer Overflow. Defined by SPE.
18 (50)	FG	Embedded Floating-Point Guard Bit FG is supplied for use by the floating-point round exception handler. FG is zeroed if a floating-point data exception occurs for the low element(s). FG corresponds to the low element result.

Table 5-1. SPE /EFPU Status and Control Register (continued)

Bits	Name	Description
19 (51)	FX	Embedded Floating-Point Sticky Bit FX is supplied for use by the floating-point round exception handler.FX is zeroed if a floating-point data exception occurs for the low element(s). FX corresponds to the low element result.
20 (52)	FINV	Embedded Floating-Point Invalid Operation/Input Error In mode 0, the FINV bit is set to 1 if the A or B low element operand of a floating-point instruction is Infinity, NaN, or Denorm, or if the operation is a divide and the low element dividend and divisor are both 0. In mode 1, the FINV bit is set on an IEEE 754 standard invalid operation (IEEE 754-1985, Section 7.1) in the low element.
21 (53)	FDBZ	Embedded Floating-Point Divide by Zero The FDBZ bit is set to 1 when a floating-point divide instruction executed with a low element divisor of 0, and the low element dividend is a finite non-zero number.
22 (54)	FUNF	Embedded Floating-Point Underflow The FUNF bit is set to 1 when the execution of a floating-point instruction results in an underflow in the low element.
23 (55)	FOVF	Embedded Floating-Point Overflow The FOVF bit is set to 1 when the execution of a floating-point instruction results in an overflow in the low element.
24 (56)	—	Reserved
25 (57)	FINXE	Embedded Floating-Point Inexact Exception Enable 0 Exception disabled 1 Exception enabled If the exception is enabled, a floating-point round exception is taken if for both elements, the result of a floating-point instruction does not result in overflow or underflow, and the result for either element is inexact (FG | FX = 1, or FGH | FXH = 1), or if the result of a floating-point instruction does result in overflow (FOVF = 1 or FOVFH = 1) for either element, but floating-point overflow exceptions are disabled (FOVFE = 0), or if the result of a floating-point instruction results in underflow (FUNF = 1 or FUNFH = 1), but floating-point underflow exceptions are disabled (FUNFE = 0), and no floating-point data exception occurs.
26 (58)	FINVE	Embedded Floating-Point Invalid Operation/Input Error Exception Enable 0 Exception disabled 1 Exception enabled If the exception is enabled, a floating-point data exception is taken if the FINV or FINVH bit is set by a floating-point instruction.
27 (59)	FDBZE	Embedded Floating-Point Divide by Zero Exception Enable 0 Exception disabled 1 Exception enabled If the exception is enabled, a floating-point data exception is taken if the FDBZ or FDBZH bit is set by a floating-point instruction.
28 (60)	FUNFE	Embedded Floating-Point Underflow Exception Enable 0 Exception disabled 1 Exception enabled If the exception is enabled, a floating-point data exception is taken if the FUNF or FUNFH bit is set by a floating-point instruction.

Table 5-1. SPE /EFPU Status and Control Register (continued)

Bits	Name	Description
29 (61)	FOVFE	Embedded Floating-Point Overflow Exception Enable 0 Exception disabled 1 Exception enabled If the exception is enabled, a Floating-point data exception is taken if the FOVF or FOVFH bit is set by a floating-point instruction.
30–31 (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

5.2.2 GPRs and Power ISA Embedded Category Instructions

The e200z7 core implements the 32-bit forms of the Power ISA embedded category instructions. These 32-bit instructions operate upon the lower half of the 64-bit GPR. These instructions do not affect the upper half of a GPR.

5.2.3 SPE/EFPU Available Bit in MSR

MSR[SPE] is defined as the SPE/EFPU available bit. If this bit is clear and software attempts to execute any of the EFPU vector instructions ($\textbf{evfs}_{xxx}$) that affect the upper 32-bits of a GPR, the EFPU Unavailable exception is taken. If this bit is set, software can execute any of the EFPU instructions.

5.2.4 Embedded Floating-point Exception Bit in ESR

ESR[SPE] is defined as the SPE/EFPU exception bit. This bit is set whenever the processor takes an exception related to the execution of a SPE instruction. This bit is also set whenever the processor takes an interrupt related to the execution of the embedded floating-point instructions. (Note that the same bit is used for SPE exceptions. Thus, SPE and embedded floating-point interrupts are indistinguishable in the ESR).

5.2.5 EFPU Exceptions

The architecture defines the following embedded floating-point exceptions:

- SPE/EFPU unavailable exception
- EFPU floating-point data exception
- EFPU floating-point round exception

Three new interrupt vector offset registers (IVORs), IVOR32, IVOR33, and IVOR34, are used by the exception model. The SPR numbers are as follows:

- 528 for IVOR32
- 529 for IVOR33
- 530 for IVOR34

These registers are privileged.

5.2.5.1 EFPU Unavailable Exception

The EFPU unavailable exception is taken if MSR[SPE] is cleared and execution of an EFPU vector instruction (**evfs**$_{\mathbf{xxx}}$) is attempted. When the EFPU Unavailable exception occurs, the processor suppresses execution of the instruction causing the exception. The SRR0, SRR1, MSR, and ESR registers are modified as follows:

- SRR0 is set to the effective address of the instruction causing the exception.
- SRR1 is set to the contents of the MSR at the time of the exception.
- MSR[CE, ME, DE] are unchanged. All other bits are cleared.
- ESR[SPE] is set. All other ESR bits are cleared.

Instruction execution resumes at address IVPR[0–15]||IVOR32[16–27]||0b0000.

5.2.5.2 Embedded Floating-point Data Exception

The embedded floating-point data exception vector is used for enabled floating-point invalid operation/input error, underflow, overflow, and divide by zero exceptions (collectively called floating-point data exceptions). When one of these enabled floating-point exceptions occurs, the processor suppresses execution of the instruction causing the exception. The SRR0, SRR1, MSR, ESR and SPEFSCR registers are modified as follows:

- SRR0 is set to the effective address of the instruction causing the exception.
- SRR1 is set to the contents of the MSR at the time of the exception.
- MSR bits CE, ME and DE are unchanged. All other bits are cleared.
- ESR[SPE] is set. All other ESR bits are cleared.
- One or more SPEFSCR status bits are set to indicate the type of exception. The affected bits are FINVH, FINV, FDBZH, FDBZ, FOVFH, FOVF, FUNFH, and FUNF. SPEFSCR[FG, FGH, FX, FXH] are cleared.

Instruction execution resumes at address IVPR[0–15]||IVOR33[16–27]||0b0000.

5.2.5.3 Embedded Floating-point Round Exception

If SPEFSCR[FINXE] is set, the embedded floating-point round exception occurs in any of the following conditions as long as no floating-point data exception is taken:

- The unrounded result of an operation is not exact.
- An overflow occurs and overflow exceptions are disabled (FOVF or FOVFH set with FOVFE cleared).
- An underflow occurs and underflow exceptions are disabled (FUNF set with FUNFE cleared).

The embedded floating-point round exception does not occur if an enabled embedded floating-point data exception occurs. When the embedded floating-point round exception occurs, the unrounded (truncated) result of an inexact high or low element is placed in the target register. If only a single element is inexact, the other exact element is updated with the correctly rounded result. The FG and FX bits corresponding to the other exact element are both 0.

The bits FG and FX are provided so that an exception handler can round the result as it desires. FG (called the “guard” bit) is the value of the bit immediately to the right of the lsb of the destination format mantissa from the infinitely precise intermediate calculation before rounding. FX (called the “sticky” bit) is the value of the “or” of all the bits to the right of the guard bit (FG) of the destination format mantissa from the infinitely precise intermediate calculation before rounding.

The SRR0, SRR1, MSR, ESR and SPEFSCR registers are modified as follows:

- SRR0 is set to the effective address of the instruction following the instruction causing the exception.
- SRR1 is set to the contents of the MSR at the time of the exception.
- MSR bits CE, ME and DE are unchanged. All other bits are cleared.
- ESR[SPE] is set. All other ESR bits are cleared.
- SPEFSCR[FGH, FG, FXH, FX] are set appropriately. SPEFSCR[FINXS] is set.

Instruction execution resumes at address IVPR[0–15]||IVOR34[16–27]||0b0000.

5.2.6 Exception Priorities

The following list shows the priority order in which exceptions are taken:

1. EFPU unavailable exception
2. EFPU floating-point data exception
3. EFPU floating-point round exception

An embedded floating-point data exception is taken if either element generates an embedded floating-point data exception. An embedded floating-point round exception is taken if either element generates an embedded floating-point round exception and neither element generates an EFPU floating-point data exception.

5.3 Embedded Floating-Point Unit Operations

The e200z7 implements floating-point instructions that operate upon the contents of a 64-bit register that is a vector of two single-precision floating-point elements. The floating-point unit shares the same register file as the integer unit. There is no separate floating-point register file. Floating-point instructions are also provided to perform scalar single precision floating-point operations on the low elements of registers, without affecting the high-order portion. The Power ISA floating-point instructions are not implemented in the e200z7.

The Freescale EIS architecture definition for embedded floating-point defines two operating modes: a real-time, default results oriented mode (mode 0) and a “true IEEE 754 standard results” operating mode (mode 1). Implementations of the embedded floating-point unit may choose to implement one or both of these modes. The e200z7 hardware implements mode 0. The IEEE 754-compatible operation is still available in mode 0 with assistance of a software envelope.

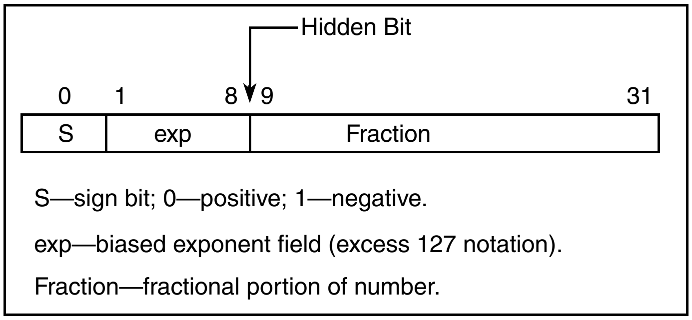
5.3.1 Floating-point Data Formats

The EFPU supports single-precision scalar and single-precision vector floating-point data operations and conversions. In addition, conversions between single-precision floating-point and the half-precision floating-point storage format are supported. These formats are described in the following subsections.

5.3.1.1 Single-Precision Floating-point Format

Each single-precision floating-point data element is 32 bits wide with one sign bit (s), 8 bits of biased exponent (*e*) and 23 bits of fraction (*f*).

In IEEE 754, floating point values are represented in a format consisting of three explicit fields (sign field, biased exponent field, and fraction field) and an implicit hidden bit.



S—sign bit; 0—positive; 1—negative.

exp—biased exponent field (excess 127 notation).

Fraction—fractional portion of number.

Figure 5-2. Single-Precision Data Format

For normalized numbers, the biased exponent value ‘*e*’ lies in the range of 1 to 254 corresponding to an actual exponent value E in the range –126 to +127, the hidden bit is a ‘1’ (for normalized numbers), and the value of the number is interpreted as in the following equation:

$$(-1)^S \times 2^E \times (1.\mathtt{fraction})$$ **Eqn. 5-1**

where E is the unbiased exponent and 1.fraction is the significand consisting of a leading ‘1’ (the hidden bit) and a fractional part (fraction field). With this format, the maximum positive normalized number (*pmax*) is represented by the encoding `0x7F7FFFFF`, which is approximately 3.4E + 38 (2^{128}). The minimum positive normalized value (*pmin*) is represented by the encoding `0x00800000`, which is approximately 1.2E – 38 (2^{-126}).

Two specific values of the biased exponent are reserved, 0 and 255, for encoding special values of ±0, ±∞, NaN, and Denorm, as follows:

- 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 biased exponent value of 255 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 exception handler is invoked, or a default value is defined, depending on the operating mode.

- Not a Numbers (NaNs) are represented by a biased exponent value *e* of 255 and a fraction *f* which is non-zero.

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*. 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. If *e* = 255 and *f*! = 0, then the value is a NaN. If *e* = 0 and *f* = 0, then the value is a signed 0.

The EFPU hardware does not produce +∞, –∞, NaN, or a denormalized number. If the result of an instruction overflows and floating-point overflow exceptions are disabled (SPEFSCR[FOVFE] is cleared), *pmax* or *nmax* is generated as the result of that instruction depending upon the sign of the result. If the result of an instruction underflows and floating-point underflow exceptions are disabled (SPEFSCR[FUNFE] is cleared), +0 or –0 is generated as the result of that instruction based upon the sign of the result.

5.3.1.2 Half-Precision Floating-point Format

Half-precision floating-point storage format is supported by the EFPU with conversion operations to and from single-precision floating-point format. No computational operations are defined for half-precision format numbers.

Each half-precision floating-point data element is 16 bits wide with one sign bit (s), 5 bits of biased exponent (*e*) and 10 bits of fraction (*f*).

In the IEEE 754r proposal, half-precision floating point values are represented in a format consisting of three explicit fields (sign field, biased exponent field, and fraction field) and an implicit hidden bit, as shown in Figure 5-3.

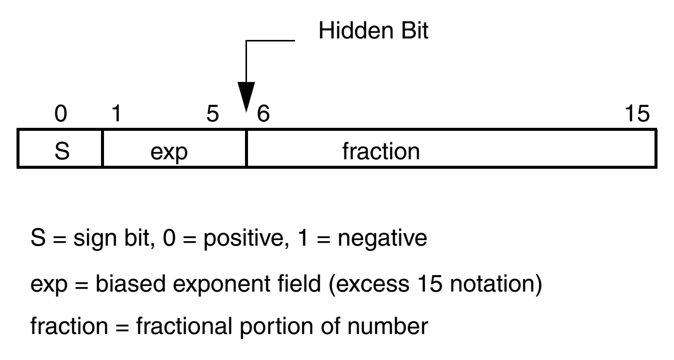


Figure 5-3. Half-Precision Data Format

For normalized numbers, the biased exponent value ‘*e*’ lies in the range of 1 to 30 corresponding to an actual exponent value E in the range –14 to +15; the hidden bit is a 1; and the value of the number is interpreted as in the following equation.

$$-1)^{S} \times 2^{E} \times (1.\mathrm{fraction} \qquad \textbf{Eqn. 5-2}$$

where E is the unbiased exponent and 1.fraction is the significand consisting of a leading ‘1’ (the hidden bit) and a fractional part (fraction field).

With this format, the maximum positive normalized number ($pmax_{hp}$) is represented by the encoding `0x7BFF,` which is 65504, and the minimum positive normalized value ($pmin_{hp}$) is represented by the encoding `0x0400,` which is approximately 6.1E-5 (2^{-14}).

Two specific values of the biased exponent are reserved; 0, and 31, for encoding special values of ±0, ±∞, NaN, and Denorm, as follows:

- 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 biased exponent value of 31 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 to be ‘0’.
- Not a Numbers (*NaNs*) are represented by a biased exponent value *e* of 31 and a fraction *f* which is non-zero.

Defining $pmax_{hp}$ to be the most positive normalized value (farthest from zero), $pmin_{hp}$ the smallest positive normalized value (closest to zero), $nmax_{hp}$ the most negative normalized value (farthest from zero) and $nmin_{hp}$ the smallest normalized negative value (closest to zero), an overflow is said to have occurred if the numerically correct result of a conversion is such that $r > pmax_{hp}$ or $r < nmax_{hp}$. An underflow is said to have occurred if the numerically correct result of a conversion is such that $0 < r < pmin_{hp}$ or $nmin_{hp} < r < 0$. In this case, r may be denormalized, or may be smaller than the smallest denormalized number. If $e = 31$ and $f \neq 0$, then the value is a NaN. If $e = 0$ and $f = 0$, then the value is a signed 0.

The EFPU hardware does not produce +∞, –∞, NaN, or a denormalized number. If the result of a conversion to half-precision format overflows and floating-point overflow exceptions are disabled (SPEFSCR[FOVFE] is cleared), then $pmax_{hp}$ or $nmax_{hp}$ is generated as the result of that instruction depending upon the sign of the result. If the result of conversion to half-precision format underflows and floating-point underflow exceptions are disabled (SPEFSCR[FUNFE] is cleared), then +0 or –0 is generated as the result of that instruction based upon the sign of the result. Conversions from half-precision format to single-precision format are always exact, unless the source operand is a NaN, Inf, or Denorm. In such cases, if floating-point invalid input exceptions are disabled (SPEFSCR[FINVE] is cleared), the conversion results in a properly signed max norm or zero default result.

5.3.2 IEEE 754 Compliance

The Freescale EIS architecture specifies that the EFPU implements a single-precision floating-point system as defined in ANSI/IEEE 754-1985 but may rely on software support in order to conform fully with the standard. Thus, whenever an input operand of the floating-point instruction has data values that are +∞, –∞, denormalized, NaN, or when the result of an operation produces an overflow or an underflow, an exception may be taken. The exception handler is responsible for delivering IEEE 754-compatible behavior, if desired.

When floating-point invalid input exceptions are disabled (SPEFSCR[FINVE] is cleared), default results are provided by the hardware when an infinity, denormalized, or NaN input is received, or for the operation 0/0. When floating-point underflow exceptions are disabled (SPEFSCR[FUNFE] is cleared) and the result of a floating-point operation underflows, a signed zero result is produced. The inexact exception is also signaled for this condition. When floating-point overflow exceptions are disabled (SPEFSCR[FOVFE] is cleared) and the result of a floating-point operation overflows, a *pmax* or *nmax* result is produced. The inexact exception is also signaled for this condition. An exception enable flag (SPEFSCR[FINXE]) is also provided for generating an exception when an inexact result is produced, which allows a software handler to conform to the IEEE 754 standard. A divide by zero exception enable flag (SPEFSCR[FDBZE]) is also provided for generating an exception when a divide by zero operation is attempted to allow a software handler to conform to the IEEE 754 standard. All of these exceptions may be disabled, and the hardware then delivers an appropriate default result.

Overflow and underflow conditions are determined after rounding on e200 implementations.

5.3.3 Floating-Point Exceptions

See Section 5.2.5, “EFPU Exceptions.”

5.3.4 Embedded Scalar Single-Precision Floating-Point Instructions

The instruction descriptions in this section, use the following conventions:

- sa = the sign of operand A
- ea = the biased exponent value of operand A
- sb = the sign of operand B
- eb = the biased exponent value of operand B
- ei = an intermediate exponent value
- r = a result value.

efsabs efsabs

Floating-Point Single-Precision Absolute Value

efsabs **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 0 0 0 1 0 0

$RD_{32:63}$ = 0b0 || $RA_{33:63}$

Description:

The sign bit of the low element of RA is set to 0 and the result is placed into the low element of RD.

Exceptions:

If the low element of RA is Infinity, Denorm, or NaN, SPEFSCR[FINV] is set, and FG and FX are cleared. FGH and FXH are cleared as well. If Floating-point Invalid Input exceptions are enabled, an exception is taken, and the destination register is not updated.

efsadd efsadd

Floating-Point Single-Precision Add

efsadd rD,rA,rB

0 5	6 10	11 15	16 20	21 31
0 0 0 1 0 0	RD	RA	RB	0 1 0 1 1 0 0 0 0 0 0

$$RD_{32:63} = RA_{32:63} +_{sp} RB_{32:63}$$

Description:

The low element of RA is added to the low element of RB and the result is stored in the low element of RD. If RA is NaN or infinity, the result is either *pmax* (`sa==0`), or *nmax* (`sa==1`). Otherwise, If RB is NaN or infinity, the result is either *pmax* (`sb==0`), or *nmax* (`sb==1`). Otherwise, if an overflow occurs, then *pmax* or *nmax* (as appropriate) is stored in RD. If an underflow occurs, then +0 (for rounding modes RN, RZ, RP) or –0 (for rounding mode RM) is stored in RD.

Exceptions:

If the contents of RA or RB are Infinity, Denorm, or NaN, SPEFSCR[FINV] is set. If SPEFSCR[FINVE] is set, an exception is taken, and the destination register is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF] is set, and if an underflow occurs, SPEFSCR[FUNF] is set. If either underflow or overflow exceptions are enabled and the corresponding bit is set, an exception is taken. If any of these exceptions are taken, the destination register is not updated.

If the result of this instruction is inexact or if an overflow occurs but overflow exceptions are disabled, and no other exception is taken, SPEFSCR[FINXS] is set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the exception handler, and the FGH and FXH bits are cleared.

FGH, FXH, FG, and FX are cleared if an overflow, underflow, or invalid operation/input error is signaled, regardless of enabled exceptions.

efscfh

efscfh

Convert Floating-Point Single-Precision from Half-Precision

efscfh **r**D,**r**B

0 5	6 10	11 15	16 20	21 31
0 0 0 1 0 0	RD	0 0 1 0 0	RB	0 1 0 1 1 0 1 0 0 0 1

```
FP16format f;
FP32format result;

f ← $rB_{48:63}$

if ($f_{exp}$ = 0) & ($f_{frac}$ = 0)) then
    result ← $f_{sign}$ || $^{31}0$   // signed zero value
else if Isa16NaNorInfinity(f) then
    $SPEFSCR_{FINV}$ ← 1
    result ← $f_{sign}$ || 0b11111110 || $^{23}1$   // max value
else if Isa16Denorm(f) then
    $SPEFSCR_{FINV}$ ← 1
    result ← $f_{sign}$ || $^{31}0$
else
    $result_{sign}$ ← $f_{sign}$
    $result_{exp}$ ← $f_{exp}$ - 15 + 127
    $result_{frac}$ ← $f_{frac}$ || $^{13}0$

$rD_{32:63}$ = result
```

The half-precision FP number in the low half of the low element in RB is converted to a single-precision floating-point value and the result is placed into the low element of RD. The rounding mode is not used since this conversion is always exact.

Exceptions:

If the source element of rB is Infinity, Denorm, or NaN, SPEFSCR[FINV] is set. If SPEFSCR[FINVE] is set, an interrupt is taken; the destination register is not updated; and the FGH, FXH, FG, and FX bits are cleared.

efscfsf

efscfsf

Convert Floating-Point Single-Precision from Signed Fraction

efscfsf 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 1 0 0 1 1

Description:

```
bl = $RB_{32:63}$
$RD_{32:63}$ = CnvtSF32ToFP32(bl)
```

The signed fractional low element in RB is converted to a single-precision floating-point value using the current rounding mode and the result is placed into the low element of RD.

Exceptions:

This instruction can signal an inexact status and set SPEFSCR[FINXS] if the conversion is not exact. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the exception handler, and the FGH and FXH bits are cleared.

efscfsi efscfsi

Convert Floating-Point Single-Precision from Signed Integer

efscfsi **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 0 1 0 0 0 1

Description:

```
$bl = RB_{32:63}$
$RD_{32:63}$ = CnvtSI32ToFP32(bl)
```

The signed integer low element in RB is converted to a single-precision floating-point value using the current rounding mode and the result is placed into the low element of RD.

Exceptions:

This instruction can signal an inexact status and set SPEFSCR[FINXS] if the conversion is not exact. If the floating-point inexact exception is enabled, an exception is taken using the floating-point round exception vector. In this case, the destination register is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the exception handler, and the FGH and FXH bits are cleared.

efscfuf

efscfuf

Convert Floating-Point Single-Precision from Unsigned Fraction

efscfuf 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 1 0 0 1 0

Description:

```
bl = $RB_{32:63}$
$RD_{32:63}$ = CnvtUF32ToFP32(bl)
```

The unsigned fractional low element in RB is converted to a single-precision floating-point value using the current rounding mode and the result is placed into the low element of RD.

Exceptions:

This instruction can signal an inexact status and set SPEFSCR[FINXS] if the conversion is not exact. If the floating-point inexact exception is enabled, an exception is taken using the floating-point round exception vector. In this case, the destination register is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the exception handler, and the FGH and FXH bits are cleared.

efscfui

efscfui

Convert Floating-Point Single-Precision from Unsigned Integer

efscfui **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 0 1 0 0 0 0

Description:

```
bl = RB$_{32:63}$
RD$_{32:63}$ = CnvtUI32ToFP32(bl)
```

The unsigned integer low element in RB is converted to a single-precision floating-point value using the current rounding mode and the result is placed into the low element of RD.

Exceptions:

This instruction can signal an inexact status and set SPEFSCR[FINXS] if the conversion is not exact. If the floating-point inexact exception is enabled, an exception is taken using the floating-point round exception vector. In this case, the destination register is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the exception handler, and the FGH and FXH bits are cleared.

efscmpgt

efscmpgt

Floating-Point Single-Precision Compare Greater Than

efscmpgt **crf**D,**r**A,**r**B

0 5	6 8	9 10	11 15	16 20	21 31
0 0 0 1 0 0	crfD	0 0	RA	RB	0 1 0 1 1 0 0 1 1 0 0

Description:

```
al = $RA_{32:63}$
bl = $RB_{32:63}$
if (al > bl) then cl = 1
else cl = 0
$CR_{4*crfD:4*crfD+3}$ = undefined || cl || undefined || undefined
```

The low element of RA is compared against the low element of RB. If RA is greater than RB, then the bit in the crfD is set, otherwise it is cleared. Comparison ignores the sign of 0 (+0 = –0).

Exceptions:

If the contents of RA or RB 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 then an exception 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.

efscmpeq efscmpeq

Floating-Point Single-Precision Compare Equal

efscmpeq **crf**D,**r**A,**r**B

0 5	6 8	9 10	11 15	16 20	21 31
0 0 0 1 0 0	crfD	0 0	RA	RB	0 1 0 1 1 0 0 1 1 1 0

Description:

```
al = $RA_{32:63}$
bl = $RB_{32:63}$
if (al == bl) then cl = 1
else cl = 0
$CR_{4*crfD:4*crfD+3}$ = undefined || cl || undefined || undefined
```

The low element of RA is compared against the low element of RB. If RA is equal to RB, then the bit in the crfD is set, otherwise it is cleared. Comparison ignores the sign of 0 (+0 = –0).

Exceptions:

If the contents of RA or RB 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 exception 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.

efscmplt

efscmplt

Floating-Point Single-Precision Compare Less Than

efscmplt **crf**D,**r**A,**r**B

0					5	6 8	9 10	11 15	16 20	21										31
0	0	0	1	0	0	crfD	0 0	RA	RB	0	1	0	1	1	0	0	1	1	0	1

Description:

```
al = $RA_{32:63}$
bl = $RB_{32:63}$
if (al < bl) then cl = 1
else cl = 0
$CR_{4*crfD:4*crfD+3}$ = undefined || cl || undefined || undefined
```

The low element of RA is compared against the low element of RB. If RA is less than RB, then the bit in the crfD is set, otherwise it is cleared. Comparison ignores the sign of 0 (+0 = –0).

Exceptions:

If the contents of RA or RB 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 then an exception 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.

efscth efscth

Convert Floating-Point Single-Precision to Half-Precision

efscth rD,rB

0 … 5	6 … 10	11 … 15	16 … 20	21 … 31
0 0 0 1 0 0	RD	0 0 1 0 0	RB	0 1 0 1 1 0 1 0 1 0 1

```
FP32format f;
FP16format result;

f ← $rB_{32:63}$

if ($f_{exp}$ = 0) & ($f_{frac}$ = 0)) then
    result ← $f_{sign}$ || $^{15}0$    // signed zero value
else if Isa32NaNorInfinity(f) then
    $SPEFSCR_{FINV}$ ← 1
    result ← $f_{sign}$ || 0b11110 || $^{10}1$    // max value
else if Isa32Denorm(f) then
    $SPEFSCR_{FINV}$ ← 1
    result ← $f_{sign}$ || $^{15}0$
else
    unbias ← $f_{exp}$ - 127
    if unbias > 15 then
        result ← $f_{sign}$ || 0b11110 || $^{10}1$    // max value
        $SPEFSCR_{FOVF}$ ← 1
    else if unbias < -14 && (result would not round up to bmin) then
        result ← $f_{sign}$ || $^{15}0$    // like-signed zero value
        $SPEFSCR_{FUNF}$ ← 1
    else
        $result_{sign}$ ← $f_{sign}$
        $result_{exp}$ ← unbias + 15
        $result_{frac}$ ← $f_{frac[0:9]}$
        guard ← $f_{frac[10]}$
        sticky ← ($f_{frac[11:22]}$ ≠ 0)
        result ← Round16(result, LOWER, guard, sticky)
        $SPEFSCR_{FG}$ ← guard
        $SPEFSCR_{FX}$ ← sticky
        if guard | sticky then
            $SPEFSCR_{FINXS}$ ← 1

$rD_{32:63}$ = $^{16}0$ || result
```

The single-precision FP number in the low element in RB is converted to a half-precision floating-point value using the current rounding mode. The result is then prepended with 16 zeros, and placed into the low element of RD.

Exceptions:

If the source element of rB is Infinity, Denorm, or NaN, SPEFSCR[FINV] is set. If SPEFSCR[FINVE] is set, an interrupt is taken, the destination register is not updated, and the FGH, FXH, FG, and FX bits are cleared. Otherwise, if an overflow occurs, SPEFSCR[FOVF] is set, and 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, the destination register 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, the destination register is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler, and the FGH and FXH bits are cleared.

FGH, FXH, FG, and FX are cleared if an overflow, underflow, or invalid operation/input error is signaled, regardless of enabled exceptions.

efsctsf efsctsf

Convert Floating-Point Single-Precision to Signed Fraction

efsctsf 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 1 0 1 1 1

Description:

```
bl = RB$_{32:63}$
if (bl == Denorm) then
    RD$_{32:63}$ = 0
else if ((bl == +0) || (bl == -0)) // zero cases
    RD$_{32:63}$ = 0
else if (ebl < 127) then
    RD$_{32:63}$ = CnvtFP32ToSF32Sat(bl)
else if ((ebl == 127) && (sbl == 1) && (fbl==0)) then
    RD$_{32:63}$ = 0x80000000 // max negative, no overflow
else if (bl == NAN) then RD$_{32:63}$ = 0
else // Overflow
    if (sbl == 0) then // Positive
        RD$_{32:63}$ = 0x7FFFFFFF
    else
        RD$_{32:63}$ = 0x80000000
```

The single-precision floating-point low element in RB 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 contents of RB are Infinity, Denorm, or NaN, or if an overflow occurs, SPEFSCR[FINV] is set, and the FGH, FXH, FG, and FX bits are cleared. If SPEFSCR[FINVE] is set, an exception is taken, and the destination register is not updated.

This instruction can signal an inexact status and set SPEFSCR[FINXS] if the conversion is not exact. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the exception handler, and the FGH and FXH bits are cleared.

efsctsi efsctsi

Convert Floating-Point Single-Precision to Signed Integer

efsctsi 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 1 0 1 0 1

Description:

```
bl = $RB_{32:63}$
if (bl == Denorm) then
    $RD_{32:63}$ = 0
else if (ebl < 158) then
    $RD_{32:63}$ = CnvtFP32ToSI32Sat(al)
else if ((ebl == 158) && (sbl == 1) && (fbl==0)) then
    $RD_{32:63}$ = 0x80000000 // max negative, no overflow
else if (bl == NAN) then $RD_{32:63}$ = 0
else // Overflow
    if (sbl == 0) then // Positive
        $RD_{32:63}$ = 0x7FFFFFFF
    else
        $RD_{32:63}$ = 0x80000000
```

The single-precision floating-point low element in RB 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 the contents of RB are Infinity, Denorm, or NaN, or if an overflow occurs, SPEFSCR[FINV] is set, and the FGH, FXH, FG, and FX bits are cleared. If SPEFSCR[FINVE] is set, an exception is taken, the destination register is not updated, and no other status bits are set.

This instruction can signal an inexact status and set SPEFSCR[FINXS] if the conversion is not exact. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the exception handler, and the FGH and FXH bits are cleared.

efsctsiz efsctsiz

Convert Floating-Point Single-Precision to Signed Integer with Round toward Zero

efsctsiz 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 1 1 0 1 0

Description:

```
bl = $RB_{32:63}$
if (bl == Denorm) then
    $RD_{32:63}$ = 0
else if (ebl < 158) then
    $RD_{32:63}$ = CnvtFP32ToSI32Sat(bl)
else if ((ebl == 158) && (sbl == 1) && (fbl==0)) then
    $RD_{32:63}$ = 0x80000000 // max negative, no overflow
else if (bl == NAN) then $RD_{32:63}$ = 0
else // Overflow
    if (sbl == 0) then // Positive
        $RD_{32:63}$ = 0x7FFFFFFF
    else
        $RD_{32:63}$ = 0x80000000
```

The single-precision floating-point low element in RB 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 RB are Infinity, Denorm, or NaN, or if an overflow occurs, SPEFSCR[FINV] is set, and the FGH, FXH, FG, and FX bits are cleared. If SPEFSCR[FINVE] is set, an exception is taken, the destination register is not updated, and no other status bits are set.

This instruction can signal an inexact status and set SPEFSCR[FINXS] if the conversion is not exact. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the exception handler, and the FGH and FXH bits are cleared.

efsctuf

Convert Floating-Point Single-Precision to Unsigned Fraction

efsctuf 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 1 0 1 1 0

Description:

```
bl = RB$_{32:63}$
if (bl == Denorm) then // force denorm to zero
    RD$_{32:63}$ = 0
else if ((bl == +0) || (bl == -0)) // zero cases
    RD$_{32:63}$ = 0
else if (sbl == 1) // Negative
    RD$_{32:63}$ = 0
else if (ebl < 127)
    RD$_{32:63}$ = CnvtFP32ToUF32Sat(bl)
else if (bl == NAN) then RD$_{32:63}$ = 0
else // Overflow
    RD$_{32:63}$ = 0xFFFFFFFF
```

The single-precision floating-point low element in RB 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 RB are Infinity, Denorm, or NaN, or if an overflow occurs, SPEFSCR[FINV] is set, and the FGH, FXH, FG, and FX bits are cleared. If SPEFSCR[FINVE] is set, an exception is taken, and the destination register is not updated.

This instruction can signal an inexact status and set SPEFSCR[FINXS] if the conversion is not exact. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the exception handler, and the FGH and FXH bits are cleared.

efsctui efsctui

Convert Floating-Point Single-Precision to Unsigned Integer

efsctui 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 1 0 1 0 0

Description:

```
bl = RB$_{32:63}$
if (bl == Denorm) then // force denorm to zero
    RD$_{32:63}$ = 0
else if ((bl == +0) || (bl == -0)) // zero cases
    RD$_{32:63}$ = 0
else if (sbl == 1) // Negative
    RD$_{32:63}$ = 0
else if (ebl <= 158)
    RD$_{32:63}$ = CnvtFP32ToUI32Sat(bl)
else if (bl == NAN) then RD$_{32:63}$ = 0
else // Overflow
    RD$_{32:63}$ = 0xFFFFFFFF
```

The single-precision floating-point low element in RB 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 the contents of RB are Infinity, Denorm, or NaN, or if an overflow occurs, SPEFSCR[FINV] is set, and the FGH, FXH, FG, and FX bits are cleared. If SPEFSCR[FINVE] is set, an exception is taken, and the destination register is not updated.

This instruction can signal an inexact status and set SPEFSCR[FINXS] if the conversion is not exact. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the exception handler, and the FGH and FXH bits are cleared.

efsctuiz

efsctuiz

Convert Floating-Point Single-Precision to Unsigned Integer with Round toward Zero

efsctui 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 1 1 0 0 0

Description:

```
bl = $RB_{32:63}$
if (bl == Denorm) then // force denorm to zero
    $RD_{32:63}$ = 0
else if ((bl == +0) || (bl == -0)) // zero cases
    $RD_{32:63}$ = 0
else if (sbl == 1) // Negative
    $RD_{32:63}$ = 0
else if (ebl <= 158)
    $RD_{32:63}$ = CnvtFP32ToUI32Sat(bl)
else if (bl == NAN) then $RD_{32:63}$ = 0
else // Overflow
    $RD_{32:63}$ = 0xFFFFFFFF
```

The single-precision floating-point low element in RB 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 the contents of RB are Infinity, Denorm, or NaN, or if an overflow occurs, SPEFSCR[FINV] is set, and the FGH, FXH, FG, and FX bits are cleared. If SPEFSCR[FINVE] is set, an exception is taken, and the destination register is not updated.

This instruction can signal an inexact status and set SPEFSCR[FINXS] if the conversion is not exact. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the exception handler, and the FGH and FXH bits are cleared.

efsdiv efsdiv

Floating-Point Single-Precision Divide

efsdiv rD**,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 0 0 1 0 0 1

$RD_{32:63} = RA_{32:63} \div_{sp} RB_{32:63}$

Description:

The low element of RA is divided by the low element of RB and the result is stored in the low element of RD. If RB is a NaN or infinity, the result is a properly signed zero. Otherwise, if RB is a denormalized number or a zero, or if RA is either NaN or infinity, the result is either *pmax* (`sa==sb`), or *nmax* (`sa!=sb`). Otherwise, if an overflow occurs, then *pmax* or *nmax* (as appropriate) is stored in RD. If an underflow occurs, then +0 or –0 (as appropriate) is stored in RD.

Exceptions:

If the contents of RA or RB are Infinity, Denorm, or NaN, or if both RA and RB are ±0, SPEFSCR[FINV] is set. If SPEFSCR[FINVE] is set, an exception is taken, and the destination register is not updated. Otherwise, if the content of RB is ±0 and the content of RA is a finite normalized non-zero number, SPEFSCR[FDBZ] is set. If Floating-point Divide by Zero exceptions are enabled, an exception 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 exception is taken. If any of these exceptions are taken, the destination register is not updated.

If the result of this instruction is inexact or if an overflow occurs but overflow exceptions are disabled, and no other exception is taken, SPEFSCR[FINXS] is set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the exception handler, and the FGH and FXH bits are cleared.

FGH, FXH, FG and FX will be cleared if an overflow, underflow, divide by zero, or invalid operation/input error is signaled, regardless of enabled exceptions.

efsmadd efsmadd

Floating-Point Single-Precision Multiply-Add

efsmadd **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 0 0 0 0 1 0

$RD_{32:63} = ((RA_{32:63} \times_{fp} RB_{32:63}) +_{sp} RD_{32:63})$

The low element of **r**A is multiplied by the low element of **r**B, the intermediate product is added to the low element of **r**D, and the result is stored in the low element of **r**D. If RA or RB are either zero or denormalized, the intermediate product is a properly signed zero. Otherwise, if RA or RB are either NaN or infinity, the intermediate product is either *pmax* (`sa==sb`), or *nmax* ($sa \neq sb$), and this value is used for the result and stored into RD. Otherwise, the intermediate product is added to the corresponding element of RD. If RD is NaN or infinity, the result is either *pmax* (`sd==0`), or *nmax* (`sd==1`). Otherwise, if an overflow occurs, then *pmax* or *nmax* (as appropriate) is stored in RD. If an underflow occurs, then +0 (for rounding modes RN, RZ, RP) or –0 (for rounding mode RM) is stored in RD.

Exceptions:

If the contents of RA or RB are Infinity, Denorm, or NaN, SPEFSCR[FINV] is set. If SPEFSCR[FINVE] is set, an exception is taken, and the destination register is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF] is set, and if an underflow occurs, SPEFSCR[FUNF] is set. If either underflow or overflow exceptions are enabled and the corresponding bit is set, an exception is taken. If any of these exceptions are taken, the destination register is not updated.

If the result of this instruction is inexact, or if an overflow occurs on the add but overflow exceptions are disabled and no other exception is taken, SPEFSCR[FINXS] is set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the exception handler, and the FGH and FXH bits are cleared.

FGH, FXH, FG and FX will be cleared if an overflow, underflow, or invalid operation/input error is signaled, regardless of enabled exceptions.

efsmax efsmax

Floating-Point Single-Precision Maximum

efsmax **rD,rA,rB**

0 5	6 8 9 10	11 15	16 20	21 31
0 0 0 1 0 0	RD	RA	RB	0 1 0 1 0 1 1 0 0 0 0

```
al ← $rA_{32:63}$
bl ← $rB_{32:63}$
if (al < bl) then temp ← bl
else temp ← al
if (isnan(al) & ~(isnan(bl))) then temp ← bl
if (isnan(bl) & ~(isnan(al))) then temp ← al
$rD_{32:63}$ ← temp
```

The low element of rA is compared against the low element of rB. The larger element is selected and placed into the low element of rD. The maximum of +0 and –0 is +0.

Exceptions:

If the contents of rA or rB are Infinity, Denorm, or NaN, SPEFSCR[FINV] is set, and the FGH, FXH, FG and FX bits are cleared. If SPEFSCR[FINVE] is set, an interrupt is taken, and the destination 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. If one of the elements is a NaN and the other is not, the non-NaN element is selected rather than the comparison result. If the selected element is denorm, the result is a same signed zero. If the selected element is +NaN or +infinity, the corresponding result is *pmax*. Otherwise, if the selected element is –NaN or –infinity, the corresponding result is *nmax*.

efsmin efsmin

Floating-Point Single-Precision Minimum

efsmin **rD,rA,rB**

0 5	6 8 9 10	11 15	16 20	21 31
0 0 0 1 0 0	RD	RA	RB	0 1 0 1 0 1 1 0 0 0 1

```
al ← $rA_{32:63}$
bl ← $rB_{32:63}$
if (al < bl) then temp ← al
else temp ← bl
if (isnan(al) & ~(isnan(bl))) then temp ← bl
if (isnan(bl) & ~(isnan(al))) then temp ← al
$rD_{32:63}$ ← temp
```

The low element of rA is compared against the low element of rB. The smaller element is selected and placed into the low element of rD. The minimum of +0 and –0 is –0.

Exceptions:

If the contents of rA or rB are Infinity, Denorm, or NaN, SPEFSCR[FINV] is set, and the FGH, FXH, FG and FX bits are cleared. If SPEFSCR[FINVE] is set, an interrupt is taken, and the destination 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. If one of the elements is a NaN and the other is not, the non-NaN element is selected rather than the comparison result. If the selected element is denorm, the result is a same signed zero. If the selected element is +NaN or +infinity, the corresponding result is *pmax*. Otherwise, if the selected element is –NaN or –infinity, the corresponding result is *nmax*.

efsmsub efsmsub

Floating-Point Single-Precision Multiply-Subtract

efsmsub **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 0 0 0 0 1 1

$RD_{32:63} = ((RA_{32:63} \; X_{fp} \; RB_{32:63}) -_{sp} RD_{32:63})$

The low element of **r**A is multiplied by the low element of **r**B, the low element of **r**D is subtracted from the intermediate product, and the result is stored in the low element of **r**D. If RA or RB are either zero or denormalized, the intermediate product is a properly signed zero. Otherwise, if RA or RB are either NaN or infinity, the intermediate product is either *pmax* (`sa==sb`), or *nmax* (`sa ≠sb`), and this value is used for the result and stored into RD. Otherwise, the low element of **r**D is subtracted from the intermediate product. If RD is NaN or infinity, the result is either *nmax* (`sd==0`), or *pmax* (`sd==1`). Otherwise, if an overflow occurs, then *pmax* or *nmax* (as appropriate) is stored in RD. If an underflow occurs, then +0 (for rounding modes RN, RZ, RP) or –0 (for rounding mode RM) is stored in RD.

Exceptions:

If the contents of RA or RB are Infinity, Denorm, or NaN, SPEFSCR[FINV] is set. If SPEFSCR[FINVE] is set, an exception is taken, and the destination register is not updated. Otherwise, if an overflow occurs, tSPEFSCR[FOVF] is set, and if an underflow occurs, SPEFSCR[FUNF] is set. If either underflow or overflow exceptions are enabled and the corresponding bit is set, an exception is taken. If any of these exceptions are taken, the destination register is not updated.

If the result of this instruction is inexact or if an overflow occurs but overflow exceptions are disabled, and no other exception is taken, the SPEFSCR[FINXS] is set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the exception handler, and the FGH and FXH bits are cleared.

FGH, FXH, FG and FX will be cleared if an overflow, underflow, or invalid operation/input error is signaled, regardless of enabled exceptions.

efsmul

efsmul

Floating-Point Single-Precision Multiply

efsmul rD,rA,rB

0 5	6 10	11 15	16 20	21 31
0 0 0 1 0 0	RD	RA	RB	0 1 0 1 1 0 0 1 0 0 0

$RD_{32:63} = RA_{32:63} X_{sp} RB_{32:63}$

Description:

The low element of RA is multiplied by the low element of RB and the result is stored in the low element of RD. If RA or RB are either zero or denormalized, the result is a properly signed zero. Otherwise, if RA or RB are either NaN or infinity, the result is either *pmax* (`sa == sb`), or *nmax* (`sa ≠ sb`). Otherwise, if an overflow occurs, then *pmax* or *nmax* (as appropriate) is stored in RD. If an underflow occurs, then +0 or –0 (as appropriate) is stored in RD.

Exceptions:

If the contents of RA or RB are Infinity, Denorm, or NaN, SPEFSCR[FINV] is set. If SPEFSCR[FINVE] is set, an exception is taken, and the destination register is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF] is set, and if an underflow occurs, the SPEFSCR[FUNF] is set. If either underflow or overflow exceptions are enabled and the corresponding bit is set, an exception is taken. If any of these exceptions are taken, the destination register is not updated.

If the result of this instruction is inexact or if an overflow occurs but overflow exceptions are disabled, and no other exception is taken, SPEFSCR[FINXS] is set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the exception handler, and the FGH and FXH bits are cleared.

FGH, FXH, FG and FX are cleared if an overflow, underflow, or invalid operation/input error is signaled, regardless of enabled exceptions.

efsnabs efsnabs

Floating-Point Single-Precision Negative Absolute Value

efsnabs **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 0 0 0 1 0 1

$RD_{32:63}$ = 0b1 || $RA_{33:63}$

Description:

The sign bit of the low element of RA is set to 1 and the result is placed into the low element of RD.

Exceptions:

If the low element of RA is Infinity, Denorm, or NaN, SPEFSCR[FINV] is set, and FG and FX are cleared. FGH and FXH are cleared as well. If Floating-point Invalid Input exceptions are enabled then an exception is taken, and the destination register is not updated.

efsneg efsneg

Floating-Point Single-Precision Negate

efsneg **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 0 0 0 1 1 0

$RD_{32:63} = \neg RA_{32} \mid\mid RA_{33:63}$

Description:

The sign bit of the low element of RA is complemented and the result is placed into the low element of RD.

Exceptions:

If the low element of RA is Infinity, Denorm, or NaN, SPEFSCR[FINV] is set, and FG and FX are cleared. FGH and FXH are cleared as well. If Floating-point Invalid Input exceptions are enabled then an exception is taken, and the destination register is not updated.

efsnmadd efsnmadd

Floating-Point Single-Precision Negative Multiply-Add

efsnmadd rD,**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 0 0 1 0 1 0

$\mathtt{RD}_{32:63} = -((\mathrm{RA}_{32:63}\ \mathrm{X}_{fp}\ \mathrm{RB}_{32:63})\ +_{sp}\ \mathrm{RD}_{32:63})$

The low element of **r**A is multiplied by the low element of **r**B, the intermediate product is added to the low element of **r**D, and the negated result is stored in the low element of **r**D. If RA or RB are either zero or denormalized, the intermediate product is a properly signed zero. Otherwise, if RA or RB are either NaN or infinity, the intermediate product is either *pmax* (`sa==sb`), or *nmax* (`sa ≠ sb`), and this value is used for the result and stored into RD. Otherwise, the intermediate product is added to the corresponding element of RD, and the final result is negated. If RD is NaN or infinity, the result is either *nmax* (`sd==0`), or *pmax* (`sd==1`). Otherwise, if an overflow occurs, then *pmax* or *nmax* (as appropriate) is stored in RD. If an underflow occurs, then –0 (for rounding modes RN, RZ, RP) or +0 (for rounding mode RM) is stored in RD.

Exceptions:

If the contents of RA or RB are Infinity, Denorm, or NaN, SPEFSCR[FINV] is set. If SPEFSCR[FINVE] is set, an exception is taken, and the destination register is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF] is set, and if an underflow occurs, SPEFSCR[FUNF] is set. If either underflow or overflow exceptions are enabled and the corresponding bit is set, an exception is taken. If any of these exceptions are taken, the destination register is not updated.

If the result of this instruction is inexact or if an overflow occurs but overflow exceptions are disabled, and no other exception is taken, SPEFSCR[FINXS] is set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the exception handler, and the FGH and FXH bits are cleared.

FGH, FXH, FG and FX will be cleared if an overflow, underflow, or invalid operation/input error is signaled, regardless of enabled exceptions.

efsnmsub

efsnmsub

Floating-Point Single-Precision Negative Multiply-Subtract

efsnmsub **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 0 0 1 0 1 1

$\mathrm{RD}_{32:63} = -((\mathrm{RA}_{32:63}\ X_{fp}\ \mathrm{RB}_{32:63}) -_{sp} \mathrm{RD}_{32:63})$

The low element of element of **r**A is multiplied by the low element of **r**B, the low element of **r**D is subtracted from the intermediate product, and the negated result is stored in the low element of **r**D. If RA or RB are either zero or denormalized, the intermediate product is a properly signed zero. Otherwise, if RA or RB are either NaN or infinity, the intermediate product is either *pmax* (`sa==sb`), or *nmax* (`sa ≠sb`), and this value is negated to obtain the result and is stored into RD. Otherwise, the low element of **r**D is subtracted from the intermediate product, and the final result is negated. If RD is NaN or infinity, the final result is either *pmax* (`sd==0`), or *nmax* (`sd==1`). Otherwise, if an overflow occurs, then *pmax* or *nmax* (as appropriate) is stored in RD. If an underflow occurs, then –0 (for rounding modes RN, RZ, RP) or +0 (for rounding mode RM) is stored in RD.

Exceptions:

If the contents of RA or RB are Infinity, Denorm, or NaN, SPEFSCR[FINV] is set. If SPEFSCR[FINVE] is set, an exception is taken, and the destination register is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF] is set, and if an underflow occurs, SPEFSCR[FUNF] is set. If either underflow or overflow exceptions are enabled and the corresponding bit is set, an exception is taken. If any of these exceptions are taken, the destination register is not updated.

If the result of this instruction is inexact or if an overflow occurs but overflow exceptions are disabled, and no other exception is taken, SPEFSCR[FINXS] is set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the exception handler, and the FGH and FXH bits are cleared.

FGH, FXH, FG and FX are cleared if an overflow, underflow, or invalid operation/input error is signaled, regardless of enabled exceptions.

efssqrt efssqrt

Floating-Point Single-Precision Square Root

efssqrt **rD,rA**

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 0 0 0 1 1 1

$rD_{32:63} \leftarrow SQRT(rA_{32:63})$

The square root of the low element of rA is calculated, and the results is stored in the low element of rD. If the low element of rA is zero or denorm, the result is a same signed zero. If the low element of rA is +NaN or +infinity, the corresponding result is *pmax*. Otherwise, if the low element of rA is non-zero and has a negative sign, including –NaN or –infinity, the corresponding result is –0. Otherwise, if an underflow occurs, +0 (for rounding modes RN, RZ, RP) or –0 (for rounding mode RM) is stored in the low element of rD.

Exceptions:

If the low element of rA is non-zero and has a negative sign, or is Infinity, Denorm, or NaN, SPEFSCR[FINV] is set, and SPEFSCR[FGH, FXH, FG, FX] are cleared. If SPEFSCR[FINVE] is set, an interrupt is taken and the destination register is not updated. Otherwise, if an underflow occurs, SPEFSCR[FUNF] is set. If underflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

If the result element of this instruction is inexact, or underflows but underflow 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, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler, and the FGH and FXH bits are cleared.

FG, FX, FGH, and FXH are cleared if an underflow or an invalid operation/input error is signaled for the low element, regardless of enabled exceptions.

efssub efssub

Floating-Point Single-Precision Subtract

efssub **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 0 0 0 0 0 1

$$RD_{32:63} = RA_{32:63} -_{sp} RB_{32:63}$$

Description:

The low element of RB is subtracted from the low element of RA and the result is stored in the low element of RD. If RA is NaN or infinity, the result is either *pmax* (`sa==0`), or *nmax* (`sa==1`). Otherwise, If RB is NaN or infinity, the result is either *nmax* (`sb==0`), or *pmax* (`sb==1`). Otherwise, if an overflow occurs, then *pmax* or *nmax* (as appropriate) is stored in RD. If an underflow occurs, then +0 (for rounding modes RN, RZ, RP) or –0 (for rounding mode RM) is stored in RD.

Exceptions:

If the contents of RA or RB are Infinity, Denorm, or NaN, SPEFSCR[FINV] is set. If SPEFSCR[FINVE] is set, an exception is taken, and the destination register is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF] is set, and if an underflow occurs, SPEFSCR[FUNF] is set. If either underflow or overflow exceptions are enabled and the corresponding bit is set, an exception is taken. If any of these exceptions are taken, the destination register is not updated.

If the result of this instruction is inexact or if an overflow occurs but overflow exceptions are disabled, and no other exception is taken, SPEFSCR[FINXS] is set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result, the FG and FX bits are properly updated to allow rounding to be performed in the exception handler, and the FGH and FXH bits are cleared.

FGH, FXH, FG and FX are cleared if an overflow, underflow, or invalid operation/input error is signaled, regardless of enabled exceptions.

efststeq

efststeq

Floating-Point Single-Precision Test Equal

efststeq **crf**D,**r**A,**r**B

0 … 5	6 … 8	9 10	11 … 15	16 … 20	21 … 31
0 0 0 1 0 0	crfD	0 0	RA	RB	0 1 0 1 1 0 1 1 1 1 0

Description:

```
al = $RA_{32:63}$
bl = $RB_{32:63}$
if (al == bl) then cl = 1
else cl = 0
$CR_{4*crfD:4*crfD+3}$ = undefined || cl || undefined || undefined
```

The low element of RA is compared against the low element of RB. If RA is equal to RB, then the bit in the crfD 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 **efststeq** instruction. If strict conformity to IEEE 754 standard is required, the program should use the **efscmpeq** instruction.

Implementation note: In an implementation, the execution of **efststeq** is likely to be faster than the execution of **efscmpeq** instruction.

efststgt

efststgt

Floating-Point Single-Precision Test Greater Than

efststgt **crf**D,**r**A,**r**B

0 5	6 8	9 10	11 15	16 20	21 31
0 0 0 1 0 0	crfD	0 0	RA	RB	0 1 0 1 1 0 1 1 1 0 0

Description:

```
al = $RA_{32:63}$
bl = $RB_{32:63}$
if (al > bl) then cl = 1
else cl = 0
$CR_{4*crfD:4*crfD+3}$ = undefined || cl || undefined || undefined
```

The low element of RA is compared against the low element of RB. If RA is greater than RB, then the bit in the crfD 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 **efststgt** instruction. If strict conformity to IEEE 754 standard is required, the program should use the **efscmpgt** instruction.

Implementation note: In an implementation, the execution of **efststgt** is likely to be faster than the execution of **efscmpgt** instruction.

efststlt efststlt

Floating-Point Single-Precision Test Less Than

efststlt **crf**D,**r**A,**r**B

0–5	6–8	9–10	11–15	16–20	21–31
0 0 0 1 0 0	crfD	0 0	RA	RB	0 1 0 1 1 0 1 1 1 0 1

Description:

```
al = $RA_{32:63}$
bl = $RB_{32:63}$
if (al < bl) then cl = 1
else cl = 0
$CR_{4*crfD:4*crfD+3}$ = undefined || cl || undefined || undefined
```

The low element of RA is compared against the low element of RB. If RA is less than RB, then the bit in the crfD 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 **efststlt** instruction. If strict conformity to IEEE 754 standard is required, the program should use the **efscmplt** instruction.

Implementation note: In an implementation, the execution of **efststlt** is likely to be faster than the execution of **efscmplt** instruction.

5.3.5 EFPU Vector Single-precision Embedded Floating-Point Instructions

The instruction descriptions in this section use the following conventions:

- sa = the sign of operand A
- ea = the biased exponent value of operand A
- sb = the sign of operand B
- eb = the biased exponent value of operand B
- ei = an intermediate exponent value
- r = a result value.

evfsabs evfsabs

Vector Floating-Point Single-Precision Absolute Value

evfsabs **r**D,**r**A

0 5	6 10	11 15	16 20	21 31
4	RD	RA	0 0 0 0 0	0 1 0 1 0 0 0 0 1 0 0

```
$RD_{0:31}$ = 0b0 || $RA_{1:31}$
$RD_{32:63}$ = 0b0 || $RA_{33:63}$
```

Description:

The sign bit of each element in RA is set to 0 and the results are placed into RD.

Exceptions:

If the contents of either element of RA are Infinity, Denorm, or NaN, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If Floating-point Invalid Input exceptions are enabled, an exception is taken and the destination register is not updated.

evfsadd

evfsadd

Vector Floating-Point Single-Precision Add

evfsadd rD,rA,rB

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	0 1 0 1 0 0 0 0 0 0 0

$$RD_{0:31} = RA_{0:31} +_{sp} RB_{0:31}$$
$$RD_{32:63} = RA_{32:63} +_{sp} RB_{32:63}$$

Description:

Each single-precision floating-point element of RA is added to the corresponding element of RB and the results are stored in RD. If RA is NaN or infinity, the result is either *pmax* (`sa==0`), or *nmax* (`sa==1`). Otherwise, If RB is NaN or infinity, the result is either *pmax* (`sb==0`), or *nmax* (`sb==1`). Otherwise, if an overflow occurs, then *pmax* or *nmax* (as appropriate) is stored in RD. If an underflow occurs, then +0 (for rounding modes RN, RZ, RP) or –0 (for rounding mode RM) is stored in RD.

Exceptions:

If the contents of either element of RA or RB are Infinity, Denorm, or NaN, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an exception is taken and the destination register is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF, FOVFH] are set appropriately, and if an underflow occurs, SPEFSCR[FUNF, FUNFH] are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an exception is taken. If any of these exceptions are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other exception is taken, or underflows but underflow exceptions are disabled, and no other exception is taken, SPEFSCR[FINXS] is set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the exception handler.

FG and FX (FGH and FXH) will be cleared if an overflow or underflow exception is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfsaddsub evfsaddsub

Vector Floating-Point Single-Precision Add / Subtract

evfsaddsub **rD,rA,rB**

0 5	6 10	11 15	16 20	21 31
0 0 0 1 0 0	RD	RA	RB	0 1 0 1 0 1 0 0 0 1 0

```
$rD_{0:31} \leftarrow rA_{0:31} +_{sp} rB_{0:31}$
$rD_{32:63} \leftarrow rA_{32:63} -_{sp} rB_{32:63}$
```

The high order single-precision floating-point element of rA is added to the corresponding element of rB, the low order single-precision floating-point element of rB is subtracted from the corresponding element of rA, and the results are stored in rD. If an element of rA is NaN or infinity, the corresponding result is either *pmax* (`sa==0`)or *nmax* (`sa==1`). Otherwise, if an element of rB is NaN or infinity, the corresponding result is either *pmax* (`sb==0`) or *nmax* (`sb==1`). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of rD. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or –0 (for rounding mode RM) is stored in the corresponding element of rD.

Exceptions:

If the contents of either element of rA or rB are Infinity, Denorm, or NaN, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an interrupt is taken and the destination register is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF, FOVFH] are set appropriately, and if an underflow occurs, SPEFSCR[FUNF, FUNFH] are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, SPEFSCR[FINXS, FINXSH] is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfsaddsubx evfsaddsubx

Vector Floating-Point Single-Precision Add / Subtract Exchanged

evfsaddsubx **rD,rA,rB**

0–5	6–10	11–15	16–20	21–31
0 0 0 1 0 0	RD	RA	RB	0 1 0 1 0 1 0 1 0 1 0

```
$rD_{0:31} \leftarrow rA_{32:63} +_{sp} rB_{0:31}$
$rD_{32:63} \leftarrow rA_{0:31} -_{sp} rB_{32:63}$
```

The high-order single-precision floating-point element of rB is added to the low-order element of rA, the low-order single-precision floating-point element of rB is subtracted from the high-order element of rA, and the results are stored in rD. If an element of rA is NaN or infinity, the corresponding result is either *pmax* (`sa==0`) or *nmax* (`sa==1`). Otherwise, if an element of rB is NaN or infinity, the corresponding result is either *pmax* (`sb==0`) or *nmax* (`sb==1`). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of rD. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or –0 (for rounding mode RM) is stored in the corresponding element of rD.

Exceptions:

If the contents of either element of rA or rB are Infinity, Denorm, or NaN, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an interrupt is taken and the destination register is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF, FOVFH] are set appropriately, and if an underflow occurs, SPEFSCR[FUNF, FUNFH] are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, SPEFSCR[FINXS, FINXSH] is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfsaddx evfsaddx

Vector Floating-Point Single-Precision Add Exchanged

evfsaddx **rD,rA,rB**

0 5	6 10	11 15	16 20	21 31
0 0 0 1 0 0	RD	RA	RB	0 1 0 1 0 1 0 1 0 0 0

$\mathrm{rD}_{0:31} \leftarrow \mathrm{rA}_{32:63} +_{sp} \mathrm{rB}_{0:31}$
$\mathrm{rD}_{32:63} \leftarrow \mathrm{rA}_{0:31} +_{sp} \mathrm{rB}_{32:63}$

The high-order single-precision floating-point element of rB is added to the low-order element of rA, the low-order single-precision floating-point element of rB is added to the high-order element of rA, and the results are stored in rD. If an element of rA is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if an element of rB is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of rD. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or –0 (for rounding mode RM) is stored in the corresponding element of rD.

Exceptions:

If the contents of either element of rA or rB are Infinity, Denorm, or NaN, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an interrupt is taken and the destination register is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF, FOVFH] are set appropriately, and if an underflow occurs, SPEFSCR[FUNF, FUNFH] are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, SPEFSCR[FINXS, FINXSH] is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfscfh evfscfh

Vector Convert Floating-Point Single-Precision from Half-Precision

evfscfh **rD,rB**

0 5	6 10	11 15	16 20	21 31
0 0 0 1 0 0	RD	0 0 1 0 0	RB	0 1 0 1 0 0 1 0 0 0 1

```
FP16format f;
FP32format result;

fh ← $rB_{24:31}$
fl ← $rB_{48:63}$

if ($fh_{exp}$ = 0) & ($fh_{frac}$ = 0)) then
    resulth ← $fh_{sign}$ || $^{31}0$   // signed zero value
else if Isa16NaNorInfinity(fh) then
    $SPEFSCR_{FINVH}$ ← 1
    resulth ← $fh_{sign}$ || 0b11111110 || $^{23}1$   // max value
else if Isa16Denorm(fh) then
    $SPEFSCR_{FINVH}$ ← 1
    resulth ← $fh_{sign}$ || $^{31}0$
else
    $resulth_{sign}$ ← $fh_{sign}$
    $resulth_{exp}$ ← $fh_{exp}$ - 15 + 127
    $resulth_{frac}$ ← $fh_{frac}$ || $^{13}0$

if ($fl_{exp}$ = 0) & ($fl_{frac}$ = 0)) then
    resultl ← $fl_{sign}$ || $^{31}0$   // signed zero value
else if Isa16NaNorInfinity(fl) then
    $SPEFSCR_{FINV}$ ← 1
    resultl ← $fl_{sign}$ || 0b11111110 || $^{23}1$   // max value
else if Isa16Denorm(fl) then
    $SPEFSCR_{FINV}$ ← 1
    resultl ← $fl_{sign}$ || $^{31}0$
else
    $resultl_{sign}$ ← $fl_{sign}$
    $resultl_{exp}$ ← $fl_{exp}$ - 15 + 127
    $resultl_{frac}$ ← $fl_{frac}$ || $^{13}0$

$rD_{0:31}$ = resulth; $rD_{32:63}$ = resultl
```

The half-precision FP number in each element in RB is converted to a single-precision floating-point value and the result is placed into the corresponding element of RD. The rounding mode is not used since this conversion is always exact.

Exceptions:

If either element of RB is Infinity, Denorm, or NaN,SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared. If SPEFSCR[FINVE] is set, an exception is taken; the destination register is not updated; and no other status bits are set.

evfscfsf evfscfsf

Vector Convert Floating-Point Single-Precision from Signed Fraction

evfscfsf **r**D,**r**B

0–5	6–10	11–15	16–20	21–31
4	RD	0 0 0 0 0	RB	0 1 0 1 0 0 1 0 0 1 1

Description:

```
$RD_{0:31}$ = CnvtSF32ToFP32($RB_{0:31}$)
$RD_{32:63}$ = CnvtSF32ToFP32($RB_{32:63}$)
```

Each signed fractional element of **r**B is converted to a single-precision floating-point value using the current rounding mode and the results are placed into the corresponding elements of **r**D.

Exceptions:

This instruction can signal an inexact status and set SPEFSCR[FINXS] if the conversions are not exact. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result(s). The FGH, FXH, FG, and FX bits are properly updated to allow rounding to be performed in the exception handler.

evfscfsi evfscfsi

Vector Convert Floating-Point Single-Precision from Signed Integer

evfscfsi **r**D,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	0 0 0 0 0	RB	0 1 0 1 0 0 1 0 0 0 1

Description:

```
RD$_{0:31}$ = CnvtSI32ToFP32(RB$_{0:31}$)
RD$_{32:63}$ = CnvtSI32ToFP32(RB$_{32:63}$)
```

Each signed integer element of **r**B is converted to the nearest single-precision floating-point value using the current rounding mode and the results are placed into the corresponding element of **r**D.

Exceptions:

This instruction can signal an inexact status and set SPEFSCR[FINXS] if the conversions are not exact. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result(s). The FGH, FXH, FG, and FX bits are properly updated to allow rounding to be performed in the exception handler.

evfscfuf

evfscfuf

Vector Convert Floating-Point Single-Precision from Unsigned Fraction

evfscfuf **r**D,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	0 0 0 0 0	RB	0 1 0 1 0 0 1 0 0 1 0

```
$RD_{0:31}$ = CnvtUF32ToFP32($RB_{0:31}$)
$RD_{32:63}$ = CnvtUF32ToFP32($RB_{32:63}$)
```

Each unsigned fractional element of **r**B is converted to a single-precision floating-point value using the current rounding mode and the results are placed into the corresponding elements of **r**D.

Exceptions:

This instruction can signal an inexact status and set SPEFSCR[FINXS] if the conversions are not exact. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result(s). The FGH, FXH, FG, and FX bits are properly updated to allow rounding to be performed in the exception handler.

evfscfui

evfscfui

Vector Convert Floating-Point Single-Precision from Unsigned Integer

evfscfui **r**D,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	0 0 0 0 0	RB	0 1 0 1 0 0 1 0 0 0 0

Description:

```
$RD_{0:31}$ = CnvtUI32ToFP32($RB_{0:31}$)
$RD_{32:63}$ = CnvtUI32ToFP32($RB_{32:63}$)
```

Each unsigned integer element of **r**B is converted to the nearest single-precision floating-point value using the current rounding mode and the results are placed into the corresponding elements of **r**D.

Exceptions:

This instruction can signal an inexact status and set SPEFSCR[FINXS] if the conversions are not exact. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result(s). The FGH, FXH, FG, and FX bits are properly updated to allow rounding to be performed in the exception handler.

evfscmpeq

evfscmpeq

Vector Floating-Point Single-Precision Compare Equal

evfscmpeq **crf**D,**r**A,**r**B

0 5	6 8	9 10	11 15	16 20	21 31
4	crfD	0 0	RA	RB	0 1 0 1 0 0 0 1 1 1 0

Description:

```
ah = $RA_{0:31}$
al = $RA_{32:63}$
bh = $RB_{0:31}$
bl = $RB_{32:63}$
if (ah == bh) then ch = 1
else ch = 0
if (al == bl) then cl = 1
else cl = 0
$CR_{4*crfD:4*crfD+3}$ = ch || cl || (ch | cl) || (ch & cl)
```

Each element of **r**A is compared against the corresponding element of **r**B. If **r**A equals RB, the **crf**D bit is set, otherwise it is cleared. Comparison ignores the sign of 0 (+0 = –0).

Exceptions:

If the contents of either element of RA or RB are Infinity, Denorm, or NaN, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If Floating-point Invalid Input exceptions are enabled then an exception 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.

evfscmpgt evfscmpgt

Vector Floating-Point Single-Precision Compare Greater Than

evfscmpgt **crf**D,**r**A,**r**B

0 5	6 8	9 10	11 15	16 20	21 31
4	crfD	0 0	RA	RB	0 1 0 1 0 0 0 1 1 0 0

Description:

```
ah = $RA_{0:31}$
al = $RA_{32:63}$
bh = $RB_{0:31}$
bl = $RB_{32:63}$
if (ah > bh) then ch = 1
else ch = 0
if (al > bl) then cl = 1
else cl = 0
$CR_{4*crfD:4*crfD+3}$ = ch || cl || (ch | cl) || (ch & cl)
```

Each element of **r**A is compared against the corresponding element of **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 either element of RA or RB are Infinity, Denorm, or NaN, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If Floating-point Invalid Input exceptions are enabled then an exception 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.

evfscmplt

evfscmplt

Vector Floating-Point Single-Precision Compare Less Than

evfscmplt **crf**D,**r**A,**r**B

0 5	6 8	9 10	11 15	16 20	21 31
4	crfD	0 0	RA	RB	0 1 0 1 0 0 0 1 1 0 1

Description:

```
ah = $RA_{0:31}$
al = $RA_{32:63}$
bh = $RB_{0:31}$
bl = $RB_{32:63}$
if (ah < bh) then ch = 1
else ch = 0
if (al < bl) then cl = 1
else cl = 0
$CR_{4*crfD:4*crfD+3}$ = ch || cl || (ch | cl) || (ch & cl)
```

Each element of **r**A is compared against the corresponding element of **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 either element of RA or RB are Infinity, Denorm, or NaN, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If Floating-point Invalid Input exceptions are enabled then an exception 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.

evfscth evfscth

Vector Convert Floating-Point Single-Precision to Half-Precision

evfscth **rD,rB**

0					5	6	10	11				15	16	20	21										31
0	0	0	1	0	0	RD		0	0	1	0	0	RB		0	1	0	1	0	0	1	0	1	0	1

```
FP32format fh, fl;
FP16format resulth, resultl;

fh ← $rB_{0:31;}$ fl ← $rB_{32:63}$

if ($fh_{exp}$ = 0) & ($fh_{frac}$ = 0)) then
    resulth ← $fh_{sign}$ || $^{15}0$   // signed zero value
else if Isa32NaNorInfinity(fh) then
    $SPEFSCR_{FINVH}$ ← 1
    result ← $fh_{sign}$ || 0b11110 || $^{10}1$   // max value
else if Isa32Denorm(fh) then
    $SPEFSCR_{FINVH}$ ← 1
    resulth ← $f_{sign}$ || $^{15}0$
else
    unbias ← $fh_{exp}$ - 127
    if unbias > 15 then
        resulth ← $fh_{sign}$ || 0b11110 || $^{10}0$   // max value
        $SPEFSCR_{FOVFH}$ ← 1
    else if unbias < -14 && (result would not round up to bmin) then
        resulth ← $fh_{sign}$ || $^{15}0$    // like-signed zero value
        $SPEFSCR_{FUNFH}$ ← 1
    else
        $resulth_{sign}$ ← $fh_{sign}$; $resulth_{exp}$ ← unbias + 15; $resulth_{frac}$ ← $fh_{frac[0:9]}$
        guard ← $fh_{frac[10]}$; sticky ← ($fh_{frac[11:22]}$ ≠ 0)
        resulth ← Round16(resulth, LOWER, guard, sticky)
        $SPEFSCR_{FGH}$ ← guard; $SPEFSCR_{FXH}$ ← sticky
        if guard | sticky then $SPEFSCR_{FINXS}$ ← 1

if ($fl_{exp}$ = 0) & ($fl_{frac}$ = 0)) then
    resultl ← $fl_{sign}$ || $^{15}0$   // signed zero value
else if Isa32NaNorInfinity(fl) then
    $SPEFSCR_{FINV}$ ← 1
    resultl ← $fl_{sign}$ || 0b11110 || $^{10}1$   // max value
else if Isa32Denorm(fl) then
    $SPEFSCR_{FINV}$ ← 1
    resultl ← $fl_{sign}$ || $^{15}0$
else
    unbias ← $fl_{exp}$ - 127
    if unbias > 15 then
        resultl ← $fl_{sign}$ || 0b11110 || $^{10}1$    // max value
        $SPEFSCR_{FOVF}$ ← 1
    else if unbias < -14 && (result would not round up to bmin) then
        resultl ← $fl_{sign}$ || $^{15}0$   // like-signed zero value
        $SPEFSCR_{FUNF}$ ← 1
    else
        $resultl_{sign}$ ← $fl_{sign}$; $resultl_{exp}$ ← unbias + 15; $resultl_{frac}$ ← $fl_{frac[0:9]}$
        guard ← $fl_{frac[10]}$; sticky ← ($fl_{frac[11:22]}$ ≠ 0)
        resultl ← Round16(resultl, LOWER, guard, sticky)
        $SPEFSCR_{FG}$ ← guard; $SPEFSCR_{FX}$ ← sticky
        if guard | sticky then $SPEFSCR_{FINXS}$ ← 1
```

```
rD$_{0:31}$ = $^{16}$0 || resulth; rD$_{32:63}$ = $^{16}$0 || resultl
```

The single-precision FP number in each element in RB is converted to a half-precision floating-point value using the current rounding mode. The result is then prepended with 16 zeros, and placed into the corresponding element of RD.

Exceptions:

If the contents of either element of rB is Infinity, Denorm, or NaN, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an interrupt is taken and the destination register is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF, FOVFH] are set appropriately, and if an underflow occurs, SPEFSCR[FUNF, FUNFH] are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, SPEFSCR[FINXS, FINXSH] is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result(s). The FGH, FXH, FG, and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfsctsf evfsctsf

Vector Convert Floating-Point Single-Precision to Signed Fraction

evfsctsf rD,rB

0–5	6–10	11–15	16–20	21–31
4	RD	0 0 0 0 0	RB	0 1 0 1 0 0 1 0 1 1 1

Description:

```
ah = RB$_{0:31}$
if (ah == Denorm) then
    RD$_{0:31}$ = 0
else if ((al == +0) || (al == -0)) // zero cases
    RD$_{0:31}$ = 0
else if (eah < 127) then
    RD$_{0:31}$ = CnvtFP32ToSF32Sat(ah)
else if ((eah == 127) && (sah == 1) && (fah==0)) then
    RD$_{0:31}$ = 0x80000000 // max negative, no overflow
else if (ah == NAN) then RD$_{0:31}$ = 0
else // Overflow
    if (sah == 0) then // Positive
        RD$_{0:31}$ = 0x7FFFFFFF
    else
        RD$_{0:31}$ = 0x80000000


al = RB$_{32:63}$
if (al == Denorm) then
    RD$_{32:63}$ = 0
else if ((al == +0) || (al == -0)) // zero cases
    RD$_{32:63}$ = 0
else if (eal < 127) then
    RD$_{32:63}$ = CnvtFP32ToSF32Sat(al)
else if ((eal == 127) && (sal == 1) && (fal==0)) then
    RD$_{32:63}$ = 0x80000000 // max negative, no overflow
else if (al == NAN) then RD$_{32:63}$ = 0
else // Overflow
    if (sal == 0) then // Positive
        RD$_{32:63}$ = 0x7FFFFFFF
    else
        RD$_{32:63}$ = 0x80000000
```

Each single-precision floating-point element in RB 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 signed fraction. NaNs are converted as though they were zero.

Exceptions:

If either element of RB is Infinity, Denorm, or NaN or if an overflow occurs, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an exception is taken; the destination register is not updated; and no other status bits are set.

If either result element of this instruction is inexact and no other exception is taken, SPEFSCR[FINXS] is set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point

Round exception vector. In this case, the destination register is updated with the truncated result. The FGH, FXH, FG, and FX bits are properly updated to allow rounding to be performed in the exception handler.

evfsctsi

evfsctsi

Vector Convert Floating-Point Single-Precision to Signed Integer

evfsctsi rD,rB

0 5	6 10	11 15	16 20	21 31
4	RD	0 0 0 0 0	RB	0 1 0 1 0 0 1 0 1 0 1

Description:

```
ah = $RB_{0:31}$
if (ah == Denorm) then
    $RD_{0:31}$ = 0
else if (eah < 158) then
    $RD_{0:31}$ = CnvtFP32ToSI32Sat(ah)
else if ((eah == 158) && (sah == 1) && (fah==0)) then
    $RD_{0:31}$ = 0x80000000 // max negative, no overflow
else if (ah == NAN) then $RD_{0:31}$ = 0
else // Overflow
    if (sah == 0) then // Positive
        $RD_{0:31}$ = 0x7FFFFFFF
    else
        $RD_{0:31}$ = 0x80000000

al = $RB_{32:63}$
if (al == Denorm) then
    $RD_{32:63}$ = 0
else if (eal < 158) then
    $RD_{32:63}$ = CnvtFP32ToSI32Sat(al)
else if ((eal == 158) && (sal == 1) && (fal==0)) then
    $RD_{32:63}$ = 0x80000000 // max negative, no overflow
else if (al == NAN) then $RD_{32:63}$ = 0
else // Overflow
    if (sal == 0) then // Positive
        $RD_{32:63}$ = 0x7FFFFFFF
    else
        $RD_{32:63}$ = 0x80000000
```

Each single-precision floating-point element in RB 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 the contents of either element of RB are Infinity, Denorm, or NaN or if an overflow occurs on conversion, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an exception is taken, the destination register is not updated, and no other status bits are set.

If either result element of this instruction is inexact and no other exception is taken, SPEFSCR[FINXS] is set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point

Round exception vector. In this case, the destination register is updated with the truncated result. The FGH, FXH, FG and FX bits are properly updated to allow rounding to be performed in the exception handler.

evfsctsiz

evfsctsiz

Vector Convert Floating-Point Single-Precision to Signed Integer with Round toward Zero

evfsctsiz rD,rB

0 5	6 10	11 15	16 20	21 31
4	RD	0 0 0 0 0	RB	0 1 0 1 0 0 1 1 0 1 0

Description:

```
ah = $RB_{0:31}$
if (ah == Denorm) then
    $RD_{0:31}$ = 0
else if (eah < 158) then
    $RD_{0:31}$ = CnvtFP32ToSI32Sat(ah)
else if ((eah == 158) && (sah == 1) && (fah==0)) then
    $RD_{0:31}$ = 0x80000000 // max negative, no overflow
else if (ah == NAN) then $RD_{0:31}$ = 0
else // Overflow
    if (sah == 0) then // Positive
        $RD_{0:31}$ = 0x7FFFFFFF
    else
        $RD_{0:31}$ = 0x80000000

al = $RB_{32:63}$
if (al == Denorm) then
    $RD_{32:63}$ = 0
else if (eal < 158) then
    $RD_{32:63}$ = CnvtFP32ToSI32Sat(al)
else if ((eal == 158) && (sal == 1) && (fal==0)) then
    $RD_{32:63}$ = 0x80000000 // max negative, no overflow
else if (al == NAN) then $RD_{32:63}$ = 0
else // Overflow
    if (sal == 0) then // Positive
        $RD_{32:63}$ = 0x7FFFFFFF
    else
        $RD_{32:63}$ = 0x80000000
```

Each single-precision floating-point element in RB 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 either element of RB is Infinity, Denorm, or NaN or if an overflow occurs, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an exception is taken, the destination register is not updated, and no other status bits are set.

If either result element of this instruction is inexact and no other exception is taken, SPEFSCR[FINXS] is set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result. The FGH, FXH, FG and FX bits are properly updated to allow rounding to be performed in the exception handler.

evfsctuf evfsctuf

Vector Convert Floating-Point Single-Precision to Unsigned Fraction

evfsctuf **r**D,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	0 0 0 0 0	RB	0 1 0 1 0 0 1 0 1 1 0

Description:

```
ah = RB$_{0:31}$
if (ah == Denorm) then // force denorm to zero
    RD$_{0:31}$ = 0
else if ((ah == +0) || (ah == -0)) // zero cases
    RD$_{0:31}$ = 0
else if (sah == 1) // Negative
    RD$_{0:31}$ = 0
else if (eah < 127)
    RD$_{0:31}$ = CnvtFP32ToUF32Sat(ah)
else if (ah == NAN) then RD$_{0:31}$ = 0
else // Overflow
    RD$_{0:31}$ = 0xFFFFFFFF

al = RB$_{32:63}$
if (al == Denorm) then
    RD$_{32:63}$ = 0
else if ((al == +0) || (al == -0)) // zero cases
    RD$_{32:63}$ = 0
else if (sal == 1) // Negative
    RD$_{32:63}$ = 0
else if (eal < 127)
    RD$_{32:63}$ = CnvtFP32ToUF32Sat(al)
else if (al == NAN) then RD$_{32:63}$ = 0
else // Overflow
    RD$_{32:63}$ = 0xFFFFFFFF
```

Each single-precision floating-point element in RB 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 fraction. NaNs are converted as though they were zero.

Exceptions:

If either element of RB is Infinity, Denorm, or NaN, or if an overflow occurs, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an exception is taken; the destination register is not updated; and no other status bits are set.

If either result element of this instruction is inexact and no other exception is taken, SPEFSCR[FINXS] is set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result. The FGH, FXH, FG and FX bits are properly updated to allow rounding to be performed in the exception handler.

evfsctui evfsctui

Vector Convert Floating-Point Single-Precision to Unsigned Integer

evfsctui rD,rB

0 5	6 10	11 15	16 20	21 31
4	RD	0 0 0 0 0	RB	0 1 0 1 0 0 1 0 1 0 0

Description:

```
ah = $RB_{0:31}$
if (ah == Denorm) then // force denorm to zero
    $RD_{0:31}$ = 0
else if ((ah == +0) || (ah == -0)) // zero cases
    $RD_{0:31}$ = 0
else if (sah == 1) // Negative
    $RD_{0:31}$ = 0
else if (eah <= 158)
    $RD_{0:31}$ = CnvtFP32ToUI32Sat(ah)
else if (ah == NAN) then $RD_{0:31}$ = 0
else // Overflow
    $RD_{0:31}$ = 0xFFFFFFFF

al = $RB_{32:63}$
if (al == Denorm) then
    $RD_{32:63}$ = 0
else if ((al == +0) || (al == -0)) // zero cases
    $RD_{32:63}$ = 0
else if (sal == 1) // Negative
    $RD_{32:63}$ = 0
else if (eal <= 158)
    $RD_{32:63}$ = CnvtFP32ToUI32Sat(al)
else if (al == NAN) then $RD_{32:63}$ = 0
else // Overflow
    $RD_{32:63}$ = 0xFFFFFFFF
```

Each single-precision floating-point element in RB 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 either element of RB is Infinity, Denorm, or NaN, or if an overflow occurs, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an exception is taken; the destination register is not updated; and no other status bits are set.

If either result element of this instruction is inexact and no other exception is taken, SPEFSCR[FINXS] is set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result. The FGH, FXH, FG and FX bits are properly updated to allow rounding to be performed in the exception handler.

evfsctuiz evfsctuiz

Vector Convert Floating-Point Single-Precision to Unsigned Integer with Round toward Zero

evfsctui rD,rB

0 5	6 10	11 15	16 20	21 31
4	RD	0 0 0 0 0	RB	0 1 0 1 0 0 1 1 0 0 0

Description:

```
ah = $RB_{0:31}$
if (ah == Denorm) then // force denorm to zero
    $RD_{0:31}$ = 0
else if ((ah == +0) || (ah == -0)) // zero cases
    $RD_{0:31}$ = 0
else if (sah == 1) // Negative
    $RD_{0:31}$ = 0
else if (eah <= 158)
    $RD_{0:31}$ = CnvtFP32ToUI32Sat(ah)
else if (ah == NAN) then $RD_{0:31}$ = 0
else // Overflow
    $RD_{0:31}$ = 0xFFFFFFFF

al = $RB_{32:63}$
if (al == Denorm) then
    $RD_{32:63}$ = 0
else if ((al == +0) || (al == -0)) // zero cases
    $RD_{32:63}$ = 0
else if (sal == 1) // Negative
    $RD_{32:63}$ = 0
else if (eal <= 158)
    $RD_{32:63}$ = CnvtFP32ToUI32Sat(al)
else if (al == NAN) then $RD_{32:63}$ = 0
else // Overflow
    $RD_{32:63}$ = 0xFFFFFFFF
```

Each single-precision floating-point element in RB 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 either element of RB is Infinity, Denorm, or NaN, or if an overflow occurs, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] cleared appropriately. If SPEFSCR[FINVE] is set, an exception is taken, the destination register is not updated, and no other status bits are set.

If either result element of this instruction is inexact and no other exception is taken, SPEFSCR[FINXS] is set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result. The FGH, FXH, FG, and FX bits are properly updated to allow rounding to be performed in the exception handler.

evfsdiff evfsdiff

Vector Floating-Point Single-Precision Differences

evfsdiff **rD,rA,rB**

0 5	6 10	11 15	16 20	21 31
0 0 0 1 0 0	RD	RA	RB	0 1 0 1 0 1 0 0 1 0 1

$rD_{0:31} \leftarrow rA_{0:31} -_{sp} rA_{32:63}$
$rD_{32:63} \leftarrow rB_{0:31} -_{sp} rB_{32:63}$

The low-order single-precision floating-point element of rA is subtracted from the high-order element of rA, the low-order single-precision floating-point element of rB is subtracted from the high-order element of rB, and the results are stored in rD. If the high-order element of rA or rB is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if the low order element of rA or rB is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of rD. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or –0 (for rounding mode RM) is stored in the corresponding element of rD.

Exceptions:

If the contents of either element of rA or rB are Infinity, Denorm, or NaN, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an interrupt is taken and the destination register is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF, FOVFH] are set appropriately, and if an underflow occurs, SPEFSCR[FUNF, FUNFH] are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, SPEFSCR[FINXS, FINXSH] is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfsdiffsum evfsdiffsum

Vector Floating-Point Single-Precision Difference / Sum

evfsdiffsum **rD,rA,rB**

0 5	6 10	11 15	16 20	21 31
0 0 0 1 0 0	RD	RA	RB	0 1 0 1 0 1 0 0 1 1 1

```
$rD_{0:31} \leftarrow rA_{0:31} -_{sp} rA_{32:63}$
$rD_{32:63} \leftarrow rB_{0:31} +_{sp} rB_{32:63}$
```

The low-order single-precision floating-point element of rA is subtracted from the high-order element of rA, the low-order single-precision floating-point element of rB is added to the high-order element of rB, and the results are stored in rD. If the high-order element of rA or rB is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if the low order element of rA or rB is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of rD. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or –0 (for rounding mode RM) is stored in the corresponding element of rD.

Exceptions:

If the contents of either element of rA or rB are Infinity, Denorm, or NaN, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an interrupt is taken and the destination register is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF, FOVFH] are set appropriately, and if an underflow occurs, SPEFSCR[FUNF, FUNFH] are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, SPEFSCR[FINXS, FINXSH] is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfsdiv evfsdiv

Vector Floating-Point Single-Precision Divide

evfsdiv rD,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	0 1 0 1 0 0 0 1 0 0 1

$RD_{0:31} = RA_{0:31} \div_{sp} RB_{0:31}$
$RD_{32:63} = RA_{32:63} \div_{sp} RB_{32:63}$

Each single-precision floating-point element of **r**A is divided by the corresponding element of **r**B and the result is stored in **r**D. If RB is a NaN or infinity, the result is a properly signed zero. Otherwise, if RB is a denormalized number or a zero, or if RA is either NaN or infinity, the result is either *pmax* (`sa==sb`), or *nmax* ($sa \neq sb$). Otherwise, if an overflow occurs, then *pmax* or *nmax* (as appropriate) is stored in RD. If an underflow occurs, then +0 or –0 (as appropriate) is stored in RD.

Exceptions:

If the contents of RA or RB are Infinity, Denorm, or NaN, or if both RA and RB are ±0, the SPEFSCR[FINV, FINVH] are set appropriately, and the SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an exception is taken and the destination register is not updated. Otherwise, if the content of RB is ±0 and the content of RA is a finite normalized non-zero number, the SPEFSCR[FDBZ, FDBZH] are set appropriately. If Floating-point Divide by Zero exceptions are enabled, an exception is then taken. Otherwise, if an overflow occurs, SPEFSCR[FOVF, FOVFH] are set appropriately, and if an underflow occurs, SPEFSCR[FUNF, FUNFH] are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding bit is set, an exception is taken. If any of these exceptions are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other exception is taken, or underflows but underflow exceptions are disabled, and no other exception is taken, SPEFSCR[FINXS] is set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the exception handler.

FG and FX (FGH and FXH) will be cleared if an overflow or underflow exception is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfsmadd evfsmadd

Vector Floating-Point Single-Precision Multiply-Add

evfsmadd rD,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	0 1 0 1 0 0 0 0 0 1 0

$$RD_{0:31} = ((RA_{0:31} \; X_{fp} \; RB_{0:31}) +_{sp} RD_{0:31})$$
$$RD_{32:63} = ((RA_{32:63} \; X_{fp} \; RB_{32:63}) +_{sp} RD_{32:63})$$

Each single-precision floating-point element of **r**A is multiplied with the corresponding element of **r**B, the intermediate product is added to the corresponding element of **r**D, and the result is stored in **r**D. If RA or RB are either zero or denormalized, the intermediate product is a properly signed zero. Otherwise, if RA or RB are either NaN or infinity, the intermediate product is either *pmax* (`sa==sb`), or *nmax* (`sa ≠ sb`), and this value is used for the result and stored into RD. Otherwise, the intermediate product is added to the corresponding element of RD. If RD is NaN or infinity, the result is either *pmax* (`sd==0`), or *nmax* (`sd==1`). Otherwise, if an overflow occurs, then *pmax* or *nmax* (as appropriate) is stored in RD. If an underflow occurs, then +0 (for rounding modes RN, RZ, RP) or –0 (for rounding mode RM) is stored in RD.

Exceptions:

If the contents of either element of RA, RB, or RD are Infinity, Denorm, or NaN, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an exception is taken and the destination register is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF, FOVFH] are set appropriately, and if an underflow occurs, SPEFSCR[FUNF, FUNFH] are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an exception is taken. If any of these exceptions are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other exception is taken, or underflows but underflow exceptions are disabled, and no other exception is taken, SPEFSCR[FINXS] is set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the exception handler.

FG and FX (FGH and FXH) will be cleared if an overflow or underflow exception is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfsmax evfsmax

Vector Floating-Point Single-Precision Maximum

evfsmax **rD,rA,rB**

0 5	6 10	11 15	16 20	21 31
0 0 0 1 0 0	RD	RA	RB	0 1 0 1 0 1 0 0 0 0 0

```
ah ← $rA_{0:31}$
bh ← $rB_{0:31}$
if (ah < bh) then temph ← bh
else temph ← ah
if (isnan(ah) & ~(isnan(bh))) then temph ← bh
if (isnan(bh) & ~(isnan(ah))) then temph ← ah
$rD_{0:31}$ ← temph

al ← $rA_{32:63}$
bl ← $rB_{32:63}$
if (al < bl) then templ ← bl
else templ ← al
if (isnan(al) & ~(isnan(bl))) then templ ← bl
if (isnan(bl) & ~(isnan(al))) then templ ← al
$rD_{32:63}$ ← templ
```

Each single-precision floating-point element of rA is compared against the corresponding elements of rB. The larger element is selected and placed into the corresponding element of rD. The maximum of +0 and –0 is +0.

Exceptions:

If the contents of either element of rA or rB are Infinity, Denorm, or NaN, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an interrupt is taken, and the destination 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. If one of the elements is a NaN and the other is not, the non-NaN element is selected rather than the comparison result. If the selected element is denorm, the result is a same signed zero. If the selected element is +NaN or +infinity, the corresponding result is *pmax*. Otherwise, if the selected element is –NaN or –infinity, the corresponding result is *nmax*.

evfsmin evfsmin

Vector Floating-Point Single-Precision Minimum

evfsmin **rD,rA,rB**

0 5	6 10	11 15	16 20	21 31
0 0 0 1 0 0	RD	RA	RB	0 1 0 1 0 1 0 0 0 0 1

```
ah ← $rA_{0:31}$
bh ← $rB_{0:31}$
if (ah < bh) then temph ← ah
else temph ← bh
if (isnan(ah) & ~(isnan(bh))) then temph ← bh
if (isnan(bh) & ~(isnan(ah))) then temph ← ah
$rD_{0:31}$ ← temph

al ← $rA_{32:63}$
bl ← $rB_{32:63}$
if (al < bl) then templ ← al
else templ ← bl
if (isnan(al) & ~(isnan(bl))) then templ ← bl
if (isnan(bl) & ~(isnan(al))) then templ ← al
$rD_{32:63}$ ← templ
```

Each single-precision floating-point element of rA is compared against the corresponding elements of rB. The smaller element is selected and placed into the corresponding element of rD. The minimum of +0 and –0 is –0.

Exceptions:

If the contents of either element of rA or rB are Infinity, Denorm, or NaN, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an interrupt is taken, and the destination 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. If one of the elements is a NaN and the other is not, the non-NaN element is selected rather than the comparison result. If the selected element is denorm, the result is a same signed zero. If the selected element is +NaN or +infinity, the corresponding result is *pmax*. Otherwise, if the selected element is –NaN or –infinity, the corresponding result is *nmax*.

evfsmsub evfsmsub

Vector Floating-Point Single-Precision Multiply-Subtract

evfsmsub **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	0 1 0 1 0 0 0 0 0 1 1

$RD_{0:31} = ((RA_{0:31} \ X_{fp} \ RB_{0:31}) -_{sp} RD_{0:31})$
$RD_{32:63} = ((RA_{32:63} \ X_{fp} \ RB_{32:63}) -_{sp} RD_{32:63})$

Each single-precision floating-point element of **r**A is multiplied with the corresponding element of **r**B, the corresponding element of **r**D is subtracted from the intermediate product, and the result is stored in **r**D. If RA or RB are either zero or denormalized, the intermediate product is a properly signed zero. Otherwise, if RA or RB are either NaN or infinity, the intermediate product is either *pmax* (`sa==sb`), or *nmax* (`sa` ≠ `sb`), and this value is used for the result and stored into RD. Otherwise, the corresponding element of **r**D is subtracted from the intermediate product. If RD is NaN or infinity, the result is either *nmax* (`sd==0`), or *pmax* (`sd==1`). Otherwise, if an overflow occurs, then *pmax* or *nmax* (as appropriate) is stored in RD. If an underflow occurs, then +0 (for rounding modes RN, RZ, RP) or –0 (for rounding mode RM) is stored in RD.

Exceptions:

If the contents of either element of RA, RB, or RD are Infinity, Denorm, or NaN, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an exception is taken and the destination register is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF, FOVFH] are set appropriately, and if an underflow occurs, SPEFSCR[FUNF, FUNFH] are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an exception is taken. If any of these exceptions are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other exception is taken, or underflows but underflow exceptions are disabled, and no other exception is taken, SPEFSCR[FINXS] is set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the exception handler.

FG and FX (FGH and FXH) will be cleared if an overflow or underflow exception is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfsmul

evfsmul

Vector Floating-Point Single-Precision Multiply

evfsmul rD,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	0 1 0 1 0 0 0 1 0 0 0

$RD_{0:31} = RA_{0:31} \times_{sp} RB_{0:31}$
$RD_{32:63} = RA_{32:63} \times_{sp} RB_{32:63}$

Each single-precision floating-point element of **r**A is multiplied with the corresponding element of **r**B and the result is stored in **r**D. If RA or RB are either zero or denormalized, the result is a properly signed zero. Otherwise, if RA or RB are either NaN or infinity, the result is either *pmax* (`sa==sb`), or *nmax* (`sa!=sb`). Otherwise, if an overflow occurs, then *pmax* or *nmax* (as appropriate) is stored in RD. If an underflow occurs, then +0 or –0 (as appropriate) is stored in RD.

Exceptions:

If the contents of either element of RA or RB are Infinity, Denorm, or NaN, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an exception is taken and the destination register is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF, FOVFH] are set appropriately, and if an underflow occurs, SPEFSCR[FUNF, FUNFH] are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an exception is taken. If any of these exceptions are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other exception is taken, or underflows but underflow exceptions are disabled, and no other exception is taken, SPEFSCR[FINXS] is set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the exception handler.

FG and FX (FGH and FXH) will be cleared if an overflow or underflow exception is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfsmule evfsmule

Vector Floating-Point Single-Precision Multiply By Even Element

evfsmule **rD,rA,rB**

0 5	6 10	11 15	16 20	21 31
0 0 0 1 0 0	RD	RA	RB	0 1 0 1 0 1 0 1 1 1 0

$rD_{0:31} \leftarrow rA_{0:31} \times_{sp} rB_{0:31}$
$rD_{32:63} \leftarrow rA_{0:31} \times_{sp} rB_{32:63}$

The single-precision floating-point elements of rB are multiplied by the even (high-order) element of rA, and the results are stored in rD. If an element of rB or the even element of rA is either zero denormalized, the corresponding result is a properly signed zero. Otherwise, if an element of rB or the even element of rA is either NaN or infinity, the corresponding result is either *pmax* (a_{sign}==b_{sign}), or *nmax* ($a_{sign} \neq b_{sign}$). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of rD. If an underflow occurs, +0 or –0 (as appropriate) is stored in the corresponding element of rD.

Exceptions:

If the contents of either element of rB or the even element of rA is Infinity, Denorm, or NaN, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an interrupt is taken and the destination register is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF, FOVFH] are set appropriately, or if an underflow occurs, SPEFSCR[FUNF, FUNFH] are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow 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, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow exception is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfsmulo evfsmulo

Vector Floating-Point Single-Precision Multiply By Odd Element

evfsmulo **rD,rA,rB**

0					5	6 10	11 15	16 20	21										31
0	0	0	1	0	0	RD	RA	RB	0	1	0	1	0	1	0	1	1	1	1

```
$rD_{0:31} \leftarrow rA_{32:63} \times_{sp} rB_{0:31}$
$rD_{32:63} \leftarrow rA_{32:63} \times_{sp} rB_{32:63}$
```

The single-precision floating-point elements of rB are multiplied by the odd (low-order) element of rA, and the results are stored in rD. If an element of rB or the odd element of rA is either zero or denormalized, the corresponding result is a properly signed zero. Otherwise, if an element of rB or the odd element of rA is either NaN or infinity, the corresponding result is either *pmax* ($a_{sign}==b_{sign}$), or *nmax* ($a_{sign} \neq b_{sign}$). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of rD. If an underflow occurs, +0 or –0 (as appropriate) is stored in the corresponding element of rD.

Exceptions:

If the contents of either element of rB or the odd element of rA is Infinity, Denorm, or NaN, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an interrupt is taken and the destination register is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF, FOVFH] are set appropriately, and if an underflow occurs, SPEFSCR[FUNF, FUNFH] are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow 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, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow exception is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfsmulx

evfsmulx

Vector Floating-Point Single-Precision Multiply Exchanged

evfsmulx **rD,rA,rB**

0 5	6 10	11 15	16 20	21 31
0 0 0 1 0 0	RD	RA	RB	0 1 0 1 0 1 0 1 1 0 0

```
$rD_{0:31} \leftarrow rA_{32:63} \times_{sp} rB_{0:31}$
$rD_{32:63} \leftarrow rA_{0:31} \times_{sp} rB_{32:63}$
```

The high-order single-precision floating-point element of rB is multiplied by the low-order element of rA, the low-order single-precision floating-point element of rB is multiplied by the high-order element of rA, and the results are stored in rD. If an element of rA or rB is either zero or denormalized, the corresponding result is a properly signed zero. Otherwise, if an element of rA or rB are either NaN or infinity, the corresponding result is either *pmax* ($a_{sign}==b_{sign}$), or *nmax* ($a_{sign} \neq b_{sign}$). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of rD. If an underflow occurs, +0 or –0 (as appropriate) is stored in the corresponding element of rD.

Exceptions:

If the contents of either element of rA or rB are Infinity, Denorm, or NaN, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an interrupt is taken and the destination register is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF, FOVFH] are set appropriately, and if an underflow occurs, SPEFSCR[FUNF, FUNFH] are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow 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, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow exception is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfsnabs

evfsnabs

Vector Floating-Point Single-Precision Negative Absolute Value

evfsnabs **r**D,**r**A

0–5	6–10	11–15	16–20	21–31
4	RD	RA	0 0 0 0 0	0 1 0 1 0 0 0 0 1 0 1

```
RD$_{0:31}$ = 0b1 || RA$_{1:31}$
RD$_{32:63}$ = 0b1 || RA$_{33:63}$
```

Description:

The sign bit of each element in RA is set to 1 and the results are placed into RD.

Exceptions:

If the contents of either element of RA are Infinity, Denorm, or NaN, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If Floating-point Invalid Input exceptions are enabled then an exception is taken, and the destination register is not updated.

evfsneg evfsneg

Vector Floating-Point Single-Precision Negate

evfsneg **r**D,**r**A

0–5	6–10	11–15	16–20	21–31
4	RD	RA	0 0 0 0 0	0 1 0 1 0 0 0 0 1 1 0

```
$RD_{0:31} = \neg RA_0 \;||\; RA_{1:31}$
$RD_{32:63} = \neg RA_{32} \;||\; RA_{33:63}$
```

Description:

The sign bit of each element in RA is complemented and the results are placed into RD.

Exceptions:

If the contents of either element of RA are Infinity, Denorm, or NaN, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If Floating-point Invalid Input exceptions are enabled then an exception is taken, and the destination register is not updated.

evfsnmadd

evfsnmadd

Vector Floating-Point Single-Precision Negative Multiply-Add

evfsnmadd rD,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	0 1 0 1 0 0 0 1 0 1 0

$$RD_{0:31} = -((RA_{0:31} \, X_{fp} \, RB_{0:31}) +_{sp} RD_{0:31})$$
$$RD_{32:63} = -((RA_{32:63} \, X_{fp} \, RB_{32:63}) +_{sp} RD_{32:63})$$

Each single-precision floating-point element of **r**A is multiplied with the corresponding element of **r**B, the intermediate product is added to the corresponding element of **r**D, and the negated result is stored in **r**D. If RA or RB are either zero or denormalized, the intermediate product is a properly signed zero. Otherwise, if RA or RB are either NaN or infinity, the intermediate product is either *pmax* (`sa==sb`), or *nmax* (`sa!=sb`), and this value is used for the result and stored into RD. Otherwise, the intermediate product is added to the corresponding element of RD, and the final result is negated. If RD is NaN or infinity, the result is either *nmax* (`sd==0`), or *pmax* (`sd==1`). Otherwise, if an overflow occurs, then *pmax* or *nmax* (as appropriate) is stored in RD. If an underflow occurs, then –0 (for rounding modes RN, RZ, RP) or +0 (for rounding mode RM) is stored in RD.

Exceptions:

If the contents of either element of RA, RB, or RD are Infinity, Denorm, or NaN, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an exception is taken and the destination register is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF, FOVFH] are set appropriately, and if an underflow occurs, SPEFSCR[FUNF, FUNFH] are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an exception is taken. If any of these exceptions are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other exception is taken, or underflows but underflow exceptions are disabled, and no other exception is taken, SPEFSCR[FINXS] is set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the exception handler.

FG and FX (FGH and FXH) will be cleared if an overflow or underflow exception is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfsnmsub evfsnmsub

Vector Floating-Point Single-Precision Negative Multiply-Subtract

evfsnmsub rD,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	0 1 0 1 0 0 0 1 0 1 1

$$RD_{0:31} = -((RA_{0:31}\ X_{fp}\ RB_{0:31})\ -_{sp}\ RD_{0:31})$$
$$RD_{32:63} = -((RA_{32:63}\ X_{fp}\ RB_{32:63})\ -_{sp}\ RD_{32:63})$$

Each single-precision floating-point element of **r**A is multiplied with the corresponding element of **r**B, the corresponding element of **r**D is subtracted from the intermediate product, and the negated result is stored in **r**D. If RA or RB are either zero or denormalized, the intermediate product is a properly signed zero. Otherwise, if RA or RB are either NaN or infinity, the intermediate product is either *pmax* (`sa==sb`), or *nmax* (`sa!=sb`), and this value is negated to obtain the result and is stored into RD. Otherwise, the corresponding element of **r**D is subtracted from the intermediate product, and the final result is negated. If RD is NaN or infinity, the final result is either *pmax* (`sd==0`), or *nmax* (`sd==1`). Otherwise, if an overflow occurs, then *pmax* or *nmax* (as appropriate) is stored in RD. If an underflow occurs, then –0 (for rounding modes RN, RZ, RP) or +0 (for rounding mode RM) is stored in RD.

Exceptions:

If the contents of either element of RA, RB, or RD are Infinity, Denorm, or NaN, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an exception is taken and the destination register is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF, FOVFH] are set appropriately, and if an underflow occurs, SPEFSCR[FUNF, FUNFH] are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an exception is taken. If any of these exceptions are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other exception is taken, or underflows but underflow exceptions are disabled, and no other exception is taken, SPEFSCR[FINXS] is set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the exception handler.

FG and FX (FGH and FXH) will be cleared if an overflow or underflow exception is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfssqrt evfssqrt

Vector Floating-Point Single-Precision Square Root

evfssqrt **rD,rA**

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 0 0 0 0 1 1 1

```
$rD_{0:31} \leftarrow SQRT(rA_{0:31})$
$rD_{32:63} \leftarrow SQRT(rA_{32:63})$
```

The square root of each single-precision floating-point element of rA is calculated, and the results are stored in rD. If an element of rA is zero or denorm, the result is a same signed zero. If an element of rA is +NaN or +infinity, the corresponding result is *pmax*. Otherwise, if an element of rA is non-zero and has a negative sign, including –NaN or –infinity, the corresponding result is –0. Otherwise, if an underflow occurs, +0 (for rounding modes RN, RZ, RP) or –0 (for rounding mode RM) is stored in the corresponding element of rD.

Exceptions:

If the contents of either element of rA are non-zero and have a negative sign, or are Infinity, Denorm, or NaN, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an interrupt is taken and the destination register is not updated. Otherwise, if an underflow occurs, SPEFSCR[FUNF, FUNFH] are set appropriately. If underflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

If either result element of this instruction is inexact, or underflows but underflow exceptions are disabled, and no other interrupt is taken, SPEFSCR[FINXS, FINXSH] is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfssub evfssub

Vector Floating-Point Single-Precision Subtract

evfssub **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	0 1 0 1 0 0 0 0 0 0 1

$RD_{0:31} = RA_{0:31} -_{sp} RB_{0:31}$
$RD_{32:63} = RA_{32:63} -_{sp} RB_{32:63}$

Description:

Each single-precision floating-point element of RB is subtracted from the corresponding element of RA and the results are stored in RD. If RA is NaN or infinity, the result is either *pmax* (`sa==0`), or *nmax* (`sa==1`). Otherwise, If RB is NaN or infinity, the result is either *nmax* (`sb==0`), or *pmax* (`sb==1`). Otherwise, if an overflow occurs, then *pmax* or *nmax* (as appropriate) is stored in RD. If an underflow occurs, then +0 (for rounding modes RN, RZ, RP) or –0 (for rounding mode RM) is stored in RD.

Exceptions:

If the contents of either element of RA or RB are Infinity, Denorm, or NaN, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an exception is taken and the destination register is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF, FOVFH] are set appropriately, and if an underflow occurs, SPEFSCR[FUNF, FUNFH] are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an exception is taken. If any of these exceptions are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other exception is taken, or underflows but underflow exceptions are disabled, and no other exception is taken, SPEFSCR[FINXS] is set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the exception handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow exception is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfssubadd evfssubadd

Vector Floating-Point Single-Precision Subtract/Add

evfssubadd **rD,rA,rB**

0 5	6 10	11 15	16 20	21 31
0 0 0 1 0 0	RD	RA	RB	0 1 0 1 0 1 0 0 0 1 1

```
$rD_{0:31} \leftarrow rA_{0:31} -_{sp} rB_{0:31}$
$rD_{32:63} \leftarrow rA_{32:63} +_{sp} rB_{32:63}$
```

The high-order single-precision floating-point element of rB is subtracted from the corresponding element of rA, the low-order single-precision floating-point element of rB is subtracted from the corresponding element of rA, and the results are stored in rD. If an element of rA is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if an element of rB is NaN or infinity, the corresponding result is either *nmax* or *pmax* (as appropriate). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of rD. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or –0 (for rounding mode RM) is stored in the corresponding element of rD.

Exceptions:

If the contents of either element of rA or rB are Infinity, Denorm, or NaN, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an interrupt is taken and the destination register is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF, FOVFH] are set appropriately, or if an underflow occurs, SPEFSCR[FUNF, FUNFH] are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow 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, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfssubaddx evfssubaddx

Vector Floating-Point Single-Precision Subtract / Add Exchanged

evfssubaddx **rD,rA,rB**

0 5	6 10	11 15	16 20	21 31
0 0 0 1 0 0	RD	RA	RB	0 1 0 1 0 1 0 1 0 1 1

$rD_{0:31} \leftarrow rA_{32:63} -_{sp} rB_{0:31}$
$rD_{32:63} \leftarrow rA_{0:31} +_{sp} rB_{32:63}$

The high-order single-precision floating-point element of rB is subtracted from the low-order element of rA, the low-order single-precision floating-point element of rB is added to the high-order from the corresponding element of rA, and the results are stored in rD. If an element of rA is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if an element of rB is NaN or infinity, the corresponding result is either *nmax* or *pmax* (as appropriate). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of rD. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or –0 (for rounding mode RM) is stored in the corresponding element of rD.

Exceptions:

If the contents of either element of rA or rB are Infinity, Denorm, or NaN, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an interrupt is taken and the destination register is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF, FOVFH] are set appropriately, and if an underflow occurs, SPEFSCR[FUNF, FUNFH] are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow 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, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfssubx evfssubx

Vector Floating-Point Single-Precision Subtract Exchanged

evfssubx **rD,rA,rB**

0 5	6 10	11 15	16 20	21 31
0 0 0 1 0 0	RD	RA	RB	0 1 0 1 0 1 0 1 0 0 1

```
$rD_{0:31} \leftarrow rA_{32:63} -_{sp} rB_{0:31}$
$rD_{32:63} \leftarrow rA_{0:31} -_{sp} rB_{32:63}$
```

The high-order single-precision floating-point element of rB is subtracted from the low-order element of rA, the low-order single-precision floating-point element of rB is subtracted from the high-order from the corresponding element of rA, and the results are stored in rD. If an element of rA is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if an element of rB is NaN or infinity, the corresponding result is either *nmax* or *pmax* (as appropriate). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of rD. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or –0 (for rounding mode RM) is stored in the corresponding element of rD.

Exceptions:

If the contents of either element of rA or rB are Infinity, Denorm, or NaN, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an interrupt is taken and the destination register is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF, FOVFH] are set appropriately, and if an underflow occurs, SPEFSCR[FUNF, FUNFH] are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

If either result element of this instruction is inexact, overflows but overflow exceptions are disabled and no other interrupt is taken, or underflows but underflow 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, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfssum evfssum

Vector Floating-Point Single-Precision Sums

evfssum **rD,rA,rB**

0 5	6 10	11 15	16 20	21 31
0 0 0 1 0 0	RD	RA	RB	0 1 0 1 0 1 0 0 1 0 0

$rD_{0:31} \leftarrow rA_{0:31} +_{sp} rA_{32:63}$
$rD_{32:63} \leftarrow rB_{0:31} +_{sp} rB_{32:63}$

The high-order single-precision floating-point element of rA is added to the low-order element of rA, the high-order single-precision floating-point element of rB is added to the low-order element of rB, and the results are stored in rD. If the high-order element of rA or rB is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if the low order element of rA or rB is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of rD. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or –0 (for rounding mode RM) is stored in the corresponding element of rD.

Exceptions:

If the contents of either element of rA or rB are Infinity, Denorm, or NaN, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an interrupt is taken and the destination register is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF, FOVFH] are set appropriately, and if an underflow occurs, SPEFSCR[FUNF, FUNFH] are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

If either result element of this instruction is inexact, overflows but overflow exceptions are disabled and no other interrupt is taken, or underflows but underflow exceptions are disabled and no other interrupt is taken, SPEFSCR[FINXS, FINXSH] is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfssumdiff evfssumdiff

Vector Floating-Point Single-Precision Sum / Difference

evfssumdiff **rD,rA,rB**

0 5	6 10	11 15	16 20	21 31
0 0 0 1 0 0	RD	RA	RB	0 1 0 1 0 1 0 0 1 1 0

$rD_{0:31} \leftarrow rA_{0:31} +_{sp} rA_{32:63}$
$rD_{32:63} \leftarrow rB_{0:31} -_{sp} rB_{32:63}$

The high-order single-precision floating-point element of rA is added to the low-order element of rA, the low-order single-precision floating-point element of rB is subtracted from the high-order element of rB, and the results are stored in rD. If the high-order element of rA or rB is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if the low order element of rA or rB is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of rD. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or –0 (for rounding mode RM) is stored in the corresponding element of rD.

Exceptions:

If the contents of either element of rA or rB are Infinity, Denorm, or NaN, SPEFSCR[FINV, FINVH] are set appropriately, and SPEFSCR[FGH, FXH, FG, FX] are cleared appropriately. If SPEFSCR[FINVE] is set, an interrupt is taken and the destination register is not updated. Otherwise, if an overflow occurs, SPEFSCR[FOVF, FOVFH] are set appropriately, and if an underflow occurs, SPEFSCR[FUNF, FUNFH] are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, SPEFSCR[FINXS, FINXSH] is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfststeq

evfststeq

Vector Floating-Point Single-Precision Test Equal

evfststeq **crf**D,**r**A,**r**B

0 5	6 8	9 10	11 15	16 20	21 31
4	crfD	0 0	RA	RB	0 1 0 1 0 0 1 1 1 1 0

Description:

```
ah = $RA_{0:31}$
al = $RA_{32:63}$
bh = $RB_{0:31}$
bl = $RB_{32:63}$
if (ah == bh) then ch = 1
else ch = 0
if (al == bl) then cl = 1
else cl = 0
$CR_{4*crfD:4*crfD+3}$ = ch || cl || (ch | cl) || (ch & cl)
```

Each element of **r**A is compared against the corresponding element of **r**B. If **r**A equals RB, the bit in **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 taken during the execution of **evfststeq**. If strict conformity to IEEE 754 standard is required, the program should use **evfscmpeq**.

Implementation note: In an implementation, the execution of **evfststeq** is likely to be faster than the execution of **evfscmpeq**.

evfststgt evfststgt

Vector Floating-Point Single-Precision Test Greater Than

evfststgt **crf**D,**r**A,**r**B

0 5	6 8	9 10	11 15	16 20	21 31
4	crfD	0 0	RA	RB	0 1 0 1 0 0 1 1 1 0 0

Description:

```
ah = $RA_{0:31}$
al = $RA_{32:63}$
bh = $RB_{0:31}$
bl = $RB_{32:63}$
if (ah > bh) then ch = 1
else ch = 0
if (al > bl) then cl = 1
else cl = 0
$CR_{4*crfD:4*crfD+3}$ = ch || cl || (ch | cl) || (ch & cl)
```

Each element of **r**A is compared against the corresponding element of **r**B. If **r**A is greater than **r**B, the bit in **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 taken during the execution of **evfststgt**. If strict conformity to IEEE 754 standard is required, the program should use **evfscmpgt**.

Implementation note: In an implementation, the execution of **evfststgt** is likely to be faster than the execution of **evfscmpgt**.

evfststlt evfststlt

Vector Floating-Point Single-Precision Test Less Than

evfststlt **crf**D,**r**A,**r**B

0 5	6 8	9 10	11 15	16 20	21 31
4	crfD	0 0	RA	RB	0 1 0 1 0 0 1 1 1 0 1

Description:

```
ah = $RA_{0:31}$
al = $RA_{32:63}$
bh = $RB_{0:31}$
bl = $RB_{32:63}$
if (ah < bh) then ch = 1
else ch = 0
if (al < bl) then cl = 1
else cl = 0
$CR_{4*crfD:4*crfD+3}$ = ch || cl || (ch | cl) || (ch & cl)
```

Each element of **r**A is compared with the corresponding element of **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 taken during the execution of **evfststlt**. If strict conformity to IEEE 754 standard is required, the program should use **evfscmplt**.

Implementation note: In an implementation, the execution of **evfststlt** is likely to be faster than the execution of **evfscmplt**.

5.4 Embedded Floating-point Results Summary

Table 5-2 summarizes the results of floating-point operations on for add, sub, mul, and div. Flag settings are performed on appropriate element flags.

Table 5-2. Floating-Point Results Summary—Add, Sub, Mul, Div

Operation	Operand A	Operand B	Result	F INV	FOVF	FUNF	FDBZ	F INX
Add								
Add	∞	∞	amax	1	0	0	0	0
Add	∞	NaN	amax	1	0	0	0	0
Add	∞	denorm	amax	1	0	0	0	0
Add	∞	zero	amax	1	0	0	0	0
Add	∞	Norm	amax	1	0	0	0	0
Add	NaN	∞	amax	1	0	0	0	0
Add	NaN	NaN	amax	1	0	0	0	0

Table 5-2. Floating-Point Results Summary—Add, Sub, Mul, Div (continued)

Operation	Operand A	Operand B	Result	F INV	FOVF	FUNF	FDBZ	F INX
Add	NaN	denorm	amax	1	0	0	0	0
Add	NaN	zero	amax	1	0	0	0	0
Add	NaN	norm	amax	1	0	0	0	0
Add	denorm	∞	bmax	1	0	0	0	0
Add	denorm	NaN	bmax	1	0	0	0	0
Add	denorm	denorm	zero[1]	1	0	0	0	0
Add	denorm	zero	zero[1]	1	0	0	0	0
Add	denorm	norm	operand_b	1	0	0	0	0
Add	zero	∞	bmax	1	0	0	0	0
Add	zero	NaN	bmax	1	0	0	0	0
Add	zero	denorm	zero[1]	1	0	0	0	0
Add	zero	zero	zero[1]	0	0	0	0	0
Add	zero	norm	operand_b	0	0	0	0	0
Add	norm	∞	bmax	1	0	0	0	0
Add	norm	NaN	bmax	1	0	0	0	0
Add	norm	denorm	operand_a	1	0	0	0	0
Add	norm	zero	operand_a	0	0	0	0	0
Add	norm	norm	_Calc_	0	*	*	0	*
Subtract								
Sub	∞	∞	amax	1	0	0	0	0
Sub	∞	NaN	amax	1	0	0	0	0
Sub	∞	denorm	amax	1	0	0	0	0
Sub	∞	zero	amax	1	0	0	0	0
Sub	∞	Norm	amax	1	0	0	0	0
Sub	NaN	∞	amax	1	0	0	0	0
Sub	NaN	NaN	amax	1	0	0	0	0
Sub	NaN	denorm	amax	1	0	0	0	0
Sub	NaN	zero	amax	1	0	0	0	0
Sub	NaN	norm	amax	1	0	0	0	0
Sub	denorm	∞	–bmax	1	0	0	0	0
Sub	denorm	NaN	–bmax	1	0	0	0	0
Sub	denorm	denorm	zero[2]	1	0	0	0	0

Table 5-2. Floating-Point Results Summary—Add, Sub, Mul, Div (continued)

Operation	Operand A	Operand B	Result	F INV	FOVF	FUNF	FDBZ	F INX
Sub	denorm	zero	zero[2]	1	0	0	0	0
Sub	denorm	norm	–operand_b	1	0	0	0	0
Sub	zero	∞	–bmax	1	0	0	0	0
Sub	zero	NaN	–bmax	1	0	0	0	0
Sub	zero	denorm	zero[2]	1	0	0	0	0
Sub	zero	zero	zero[2]	0	0	0	0	0
Sub	zero	norm	–operand_b	0	0	0	0	0
Sub	norm	∞	–bmax	1	0	0	0	0
Sub	norm	NaN	–bmax	1	0	0	0	0
Sub	norm	denorm	operand_a	1	0	0	0	0
Sub	norm	zero	operand_a	0	0	0	0	0
Sub	norm	norm	_Calc_	0	*	*	0	*
Multiply[3]								
Mul	∞	∞	max	1	0	0	0	0
Mul	∞	NaN	max	1	0	0	0	0
Mul	∞	denorm	zero	1	0	0	0	0
Mul	∞	zero	zero	1	0	0	0	0
Mul	∞	Norm	max	1	0	0	0	0
Mul	NaN	∞	max	1	0	0	0	0
Mul	NaN	NaN	max	1	0	0	0	0
Mul	NaN	denorm	zero	1	0	0	0	0
Mul	NaN	zero	zero	1	0	0	0	0
Mul	NaN	norm	max	1	0	0	0	0
Mul	denorm	∞	zero	1	0	0	0	0
Mul	denorm	NaN	zero	1	0	0	0	0
Mul	denorm	denorm	zero	1	0	0	0	0
Mul	denorm	zero	zero	1	0	0	0	0
Mul	denorm	norm	zero	1	0	0	0	0
Mul	zero	∞	zero	1	0	0	0	0
Mul	zero	NaN	zero	1	0	0	0	0
Mul	zero	denorm	zero	1	0	0	0	0
Mul	zero	zero	zero	0	0	0	0	0

Table 5-2. Floating-Point Results Summary—Add, Sub, Mul, Div (continued)

Operation	Operand A	Operand B	Result	F INV	FOVF	FUNF	FDBZ	F INX
Mul	zero	norm	zero	0	0	0	0	0
Mul	norm	∞	max	1	0	0	0	0
Mul	norm	NaN	max	1	0	0	0	0
Mul	norm	denorm	zero	1	0	0	0	0
Mul	norm	zero	zero	0	0	0	0	0
Mul	norm	norm	_Calc_	0	*	*	0	*
Divide[3]								
Div	∞	∞	zero	1	0	0	0	0
Div	∞	NaN	zero	1	0	0	0	0
Div	∞	denorm	max	1	0	0	0	0
Div	∞	zero	max	1	0	0	0	0
Div	∞	Norm	max	1	0	0	0	0
Div	NaN	∞	zero	1	0	0	0	0
Div	NaN	NaN	zero	1	0	0	0	0
Div	NaN	denorm	max	1	0	0	0	0
Div	NaN	zero	max	1	0	0	0	0
Div	NaN	norm	max	1	0	0	0	0
Div	denorm	∞	zero	1	0	0	0	0
Div	denorm	NaN	zero	1	0	0	0	0
Div	denorm	denorm	max	1	0	0	0	0
Div	denorm	zero	max	1	0	0	0	0
Div	denorm	norm	zero	1	0	0	0	0
Div	zero	∞	zero	1	0	0	0	0
Div	zero	NaN	zero	1	0	0	0	0
Div	zero	denorm	max	1	0	0	0	0
Div	zero	zero	max	1	0	0	0	0
Div	zero	norm	zero	0	0	0	0	0
Div	norm	∞	zero	1	0	0	0	0
Div	norm	NaN	zero	1	0	0	0	0
Div	norm	denorm	max	1	0	0	0	0

Table 5-2. Floating-Point Results Summary—Add, Sub, Mul, Div (continued)

Operation	Operand A	Operand B	Result	F INV	FOVF	FUNF	FDBZ	F INX
Div	norm	zero	max	0	0	0	1	0
Div	norm	norm	_Calc_	0	*	*	0	*

Notes:

The following definitions apply:

1. - sign of result is positive when sign_a and sign_b are different for all rounding modes except round to minus infinity, where it is negative.
2. - sign of result is positive when sign_a and sign_b are the same for all rounding modes except round to minus infinity, where it is negative.
3. - sign of result is always (sign_a XOR sign_b)
   * - updated according to results of calculation
   Calc - result is updated with the results of calculation
   max - max normalized number with sign of (sign_a XOR sign_b)
   amax - max normalized number with sign of sign_a

bmax - max normalized number with sign of sign_b

nmax - max negative normalized number
pmax - max positive normalized number

Table 5-3 summarizes the results of floating-point operations on for madd, msub, nmadd, and nmsub.

Table 5-3. Floating-Point Results Summary—madd, msub, nmadd, nmsub

Operation	Operand A	Operand B	Operand D	Result	F INV	FOVF	FUNF	FDBZ	F INX
madd									
madd	∞ , NaN	∞ , NaN, Norm	∞ , NaN, denorm, zero, Norm	abmax	1	0	0	0	0
madd	∞ , NaN	denorm, zero	∞ , NaN	dmax	1	0	0	0	0
madd	∞ , NaN	denorm, zero	denorm, zero	zero[1]	1	0	0	0	0
madd	∞ , NaN	denorm, zero	Norm	operand_d	1	0	0	0	0
madd	denorm	∞ , NaN, denorm, zero, Norm	∞ , NaN	dmax	1	0	0	0	0
madd	denorm	∞ , NaN, denorm, zero, Norm	denorm, zero	zero[1]	1	0	0	0	0
madd	denorm	∞ , NaN, denorm, zero, Norm	Norm	operand_d	1	0	0	0	0
madd	zero	∞ , NaN, denorm,	∞ , NaN	dmax	1	0	0	0	0
madd	zero	∞ , NaN, denorm	denorm, zero	zero[1]	1	0	0	0	0
madd	zero	∞ , NaN, denorm	Norm	operand_d	1	0	0	0	0
madd	zero	zero, Norm	∞ , NaN	dmax	1	0	0	0	0
madd	zero	zero, Norm	denorm	zero[1]	1	0	0	0	0
madd	zero	zero, Norm	zero	zero[1]	0	0	0	0	0
madd	zero	zero, Norm	Norm	operand_d	0	0	0	0	0

Table 5-3. Floating-Point Results Summary—madd, msub, nmadd, nmsub (continued)

Operation	Operand A	Operand B	Operand D	Result	F INV	FOVF	FUNF	FDBZ	F INX
madd	norm	∞ , NaN	∞ , NaN, denorm, zero, Norm	abmax	1	0	0	0	0
madd	norm	denorm	∞ , NaN	dmax	1	0	0	0	0
madd	norm	denorm	denorm, zero	zero[1]	1	0	0	0	0
madd	norm	denorm	norm	operand_d	1	0	0	0	0
madd	norm	zero	∞ , NaN	dmax	1	0	0	0	0
madd	norm	zero	denorm	zero[1]	1	0	0	0	0
madd	norm	zero	zero	zero[1]	0	0	0	0	0
madd	norm	zero	norm	operand_d	0	0	0	0	0
madd	norm	norm	∞ , NaN	dmax	1	0	0	0	0
madd	norm	norm	denorm	ab_Calc	1	*	*	0	*
madd	norm	norm	zero	ab_Calc	0	*	*	0	*
madd	norm	norm	norm	_Calc_	0	*	*	0	*
nmadd									
nmadd	∞ , NaN	∞ , NaN, Norm	∞ , NaN, denorm, zero, Norm	–abmax	1	0	0	0	0
nmadd	∞ , NaN	denorm, zero	∞ , NaN	–dmax	1	0	0	0	0
nmadd	∞ , NaN	denorm, zero	denorm, zero	zero[3]	1	0	0	0	0
nmadd	∞ , NaN	denorm, zero	Norm	–operand_d	1	0	0	0	0
nmadd	denorm	∞ , NaN, denorm, zero, Norm	∞ , NaN	–dmax	1	0	0	0	0
nmadd	denorm	∞ , NaN, denorm, zero, Norm	denorm, zero	zero[3]	1	0	0	0	0
nmadd	denorm	∞ , NaN, denorm, zero, Norm	Norm	–operand_d	1	0	0	0	0
nmadd	zero	∞ , NaN, denorm,	∞ , NaN	–dmax	1	0	0	0	0
nmadd	zero	∞ , NaN, denorm	denorm, zero	zero[3]	1	0	0	0	0
nmadd	zero	∞ , NaN, denorm	Norm	–operand_d	1	0	0	0	0
nmadd	zero	zero, Norm	∞ , NaN	–dmax	1	0	0	0	0
nmadd	zero	zero, Norm	denorm	zero[3]	1	0	0	0	0
nmadd	zero	zero, Norm	zero	zero[3]	0	0	0	0	0
nmadd	zero	zero, Norm	Norm	–operand_d	0	0	0	0	0
nmadd	norm	∞ , NaN	∞ , NaN, denorm, zero, Norm	–abmax	1	0	0	0	0
nmadd	norm	denorm	∞ , NaN	–dmax	1	0	0	0	0

Table 5-3. Floating-Point Results Summary—madd, msub, nmadd, nmsub (continued)

Operation	Operand A	Operand B	Operand D	Result	F INV	FOVF	FUNF	FDBZ	F INX
nmadd	norm	denorm	denorm, zero	zero[3]	1	0	0	0	0
nmadd	norm	denorm	norm	–operand_d	1	0	0	0	0
nmadd	norm	zero	∞ , NaN	–dmax	1	0	0	0	0
nmadd	norm	zero	denorm	zero[3]	1	0	0	0	0
nmadd	norm	zero	zero	zero[3]	0	0	0	0	0
nmadd	norm	zero	norm	–operand_d	0	0	0	0	0
nmadd	norm	norm	∞ , NaN	–dmax	1	0	0	0	0
nmadd	norm	norm	denorm	–ab_Calc	1	*	*	0	*
nmadd	norm	norm	zero	–ab_Calc	0	*	*	0	*
nmadd	norm	norm	norm	–(_Calc_)	0	*	*	0	*
msub									
msub	∞ , NaN	∞ , NaN, Norm	∞ , NaN, denorm, zero, Norm	abmax	1	0	0	0	0
msub	∞ , NaN	denorm, zero	∞ , NaN	–dmax	1	0	0	0	0
msub	∞ , NaN	denorm, zero	denorm, zero	zero[2]	1	0	0	0	0
msub	∞ , NaN	denorm, zero	Norm	–operand_d	1	0	0	0	0
msub	denorm	∞ , NaN, denorm, zero, Norm	∞ , NaN	–dmax	1	0	0	0	0
msub	denorm	∞ , NaN, denorm, zero, Norm	denorm, zero	zero[2]	1	0	0	0	0
msub	denorm	∞ , NaN, denorm, zero, Norm	Norm	–operand_d	1	0	0	0	0
msub	zero	∞ , NaN, denorm,	∞ , NaN	–dmax	1	0	0	0	0
msub	zero	∞ , NaN, denorm	denorm, zero	zero[2]	1	0	0	0	0
msub	zero	∞ , NaN, denorm	Norm	–operand_d	1	0	0	0	0
msub	zero	zero, Norm	∞ , NaN	–dmax	1	0	0	0	0
msub	zero	zero, Norm	denorm	zero[2]	1	0	0	0	0
msub	zero	zero, Norm	zero	zero[2]	0	0	0	0	0
msub	zero	zero, Norm	Norm	–operand_d	0	0	0	0	0
msub	norm	∞ , NaN	∞ , NaN, denorm, zero, Norm	abmax	1	0	0	0	0
msub	norm	denorm	∞ , NaN	–dmax	1	0	0	0	0
msub	norm	denorm	denorm, zero	zero[2]	1	0	0	0	0
msub	norm	denorm	norm	–operand_d	1	0	0	0	0

Table 5-3. Floating-Point Results Summary—madd, msub, nmadd, nmsub (continued)

Operation	Operand A	Operand B	Operand D	Result	F INV	FOVF	FUNF	FDBZ	F INX
msub	norm	zero	∞ , NaN	–dmax	1	0	0	0	0
msub	norm	zero	denorm	zero[2]	1	0	0	0	0
msub	norm	zero	zero	zero[2]	0	0	0	0	0
msub	norm	zero	norm	–operand_d	0	0	0	0	0
msub	norm	norm	∞ , NaN	–dmax	1	0	0	0	0
msub	norm	norm	denorm	ab_Calc	1	*	*	0	*
msub	norm	norm	zero	ab_Calc	0	*	*	0	*
msub	norm	norm	norm	_Calc_	0	*	*	0	*
nmsub									
nmsub	∞ , NaN	∞ , NaN, Norm	∞ , NaN, denorm, zero, Norm	–abmax	1	0	0	0	0
nmsub	∞ , NaN	denorm, zero	∞ , NaN	dmax	1	0	0	0	0
nmsub	∞ , NaN	denorm, zero	denorm, zero	zero[4]	1	0	0	0	0
nmsub	∞ , NaN	denorm, zero	Norm	operand_d	1	0	0	0	0
nmsub	denorm	∞ , NaN, denorm, zero, Norm	∞ , NaN	dmax	1	0	0	0	0
nmsub	denorm	∞ , NaN, denorm, zero, Norm	denorm, zero	zero[4]	1	0	0	0	0
nmsub	denorm	∞ , NaN, denorm, zero, Norm	Norm	operand_d	1	0	0	0	0
nmsub	zero	∞ , NaN, denorm,	∞ , NaN	dmax	1	0	0	0	0
nmsub	zero	∞ , NaN, denorm	denorm, zero	zero[4]	1	0	0	0	0
nmsub	zero	∞ , NaN, denorm	Norm	operand_d	1	0	0	0	0
nmsub	zero	zero, Norm	∞ , NaN	dmax	1	0	0	0	0
nmsub	zero	zero, Norm	denorm	zero[4]	1	0	0	0	0
nmsub	zero	zero, Norm	zero	zero[4]	0	0	0	0	0
nmsub	zero	zero, Norm	Norm	–operand_d	0	0	0	0	0
nmsub	norm	∞ , NaN	∞ , NaN, denorm, zero, Norm	–abmax	1	0	0	0	0
nmsub	norm	denorm	∞ , NaN	dmax	1	0	0	0	0
nmsub	norm	denorm	denorm, zero	zero[4]	1	0	0	0	0
nmsub	norm	denorm	norm	operand_d	1	0	0	0	0
nmsub	norm	zero	∞ , NaN	dmax	1	0	0	0	0
nmsub	norm	zero	denorm	zero[4]	1	0	0	0	0

Table 5-3. Floating-Point Results Summary—madd, msub, nmadd, nmsub (continued)

Operation	Operand A	Operand B	Operand D	Result	F INV	FOVF	FUNF	FDBZ	F INX
nmsub	norm	zero	zero	zero[4]	0	0	0	0	0
nmsub	norm	zero	norm	operand_d	0	0	0	0	0
nmsub	norm	norm	∞ , NaN	dmax	1	0	0	0	0
nmsub	norm	norm	denorm	–ab_Calc	1	*	*	0	*
nmsub	norm	norm	zero	–ab_Calc	0	*	*	0	*
nmsub	norm	norm	norm	–(_Calc_)	0	*	*	0	*

Notes:

The following definitions apply:

1. – sign of result is positive when (sign_a XOR sign_b) and sign_d are different for all rounding modes except round to minus infinity, where it is negative.
2. – sign of result is positive when (sign_a XOR sign_b) and sign_d are the same for all rounding modes except round to minus infinity, where it is negative.
3. – sign of result is negative when (sign_a XOR sign_b) and sign_d are different for all rounding modes except round to minus infinity, where it is positive.
4. – sign of result is negative when (sign_a XOR sign_b) and sign_d are the same for all rounding modes except round to minus infinity, where it is positive.

* – updated according to results of calculation
ab_Calc – result is updated with the results of intermediate product calculation, rounded
Calc – result is updated with the results of calculation, rounded
abmax – max normalized number with sign of (sign_a XOR sign_b)
dmax – max normalized number with sign of sign_d
nmax – max negative normalized number
pmax – max positive normalized number

Table 5-4 summarizes the results of floating-point operations for sqrt.

Table 5-4. Floating-Point Results Summary—sqrt

Operand A	Result	F INV	FOVF	FUNF	FDBZ	F INX
+∞	pmax	1	0	0	0	0
–∞	–0	1	0	0	0	0
+NaN	pmax	1	0	0	0	0
–NaN	–0	1	0	0	0	0
+denorm	+zero	1	0	0	0	0
–denorm	–zero	1	0	0	0	0
+zero	+zero	0	0	0	0	0
–zero	–zero	0	0	0	0	0
+norm	_Calc_	0	*	*	0	*
–norm	–0	1	0	0	0	0

Table 5-5 shows the floating-point results summary for min and max.

Table 5-5. Floating–Point Results Summary—Min, Max

Operand A	Operand B	Result	FINV	FOVF	FUNF	FDBZ	FINX
Max							
+∞	+∞	pmax	1	0	0	0	0
+∞	–∞	pmax	1	0	0	0	0
+∞	+NaN	pmax	1	0	0	0	0
+∞	–NaN	pmax	1	0	0	0	0
+∞	denorm	pmax	1	0	0	0	0
+∞	zero	pmax	1	0	0	0	0
+∞	Norm	pmax	1	0	0	0	0
–∞	+∞	pmax	1	0	0	0	0
–∞	–∞	nmax	1	0	0	0	0
–∞	+NaN	nmax	1	0	0	0	0
–∞	–NaN	nmax	1	0	0	0	0
–∞	denorm	bzero	1	0	0	0	0
–∞	zero	bzero	1	0	0	0	0
–∞	Norm	operand_b	1	0	0	0	0
+NaN	+∞	pmax	1	0	0	0	0
+NaN	–∞	nmax	1	0	0	0	0
+NaN	+NaN	pmax	1	0	0	0	0
+NaN	–NaN	pmax	1	0	0	0	0
+NaN	denorm	bzero	1	0	0	0	0
+NaN	zero	bzero	1	0	0	0	0
+NaN	Norm	operand_b	1	0	0	0	0
–NaN	+∞	pmax	1	0	0	0	0
–NaN	–∞	nmax	1	0	0	0	0
–NaN	+NaN	pmax	1	0	0	0	0
–NaN	–NaN	nmax	1	0	0	0	0
–NaN	denorm	bzero	1	0	0	0	0
–NaN	zero	bzero	1	0	0	0	0
–NaN	Norm	operand_b	1	0	0	0	0
+denorm	+∞	pmax	1	0	0	0	0

Table 5-5. Floating–Point Results Summary—Min, Max (continued)

Operand A	Operand B	Result	FINV	FOVF	FUNF	FDBZ	FINX
+denorm	–∞	azero	1	0	0	0	0
+denorm	+NaN	azero	1	0	0	0	0
+denorm	–NaN	azero	1	0	0	0	0
+denorm	denorm	azero	1	0	0	0	0
+denorm	zero	azero	1	0	0	0	0
+denorm	+Norm	operand_b	1	0	0	0	0
+denorm	–Norm	azero	1	0	0	0	0
–denorm	+∞	pmax	1	0	0	0	0
–denorm	–∞	azero	1	0	0	0	0
–denorm	+NaN	azero	1	0	0	0	0
–denorm	–NaN	azero	1	0	0	0	0
–denorm	denorm	bzero	1	0	0	0	0
–denorm	zero	bzero	1	0	0	0	0
–denorm	+Norm	operand_b	1	0	0	0	0
–denorm	–Norm	azero	1	0	0	0	0
+zero	+∞	pmax	1	0	0	0	0
+zero	–∞	azero	1	0	0	0	0
+zero	+NaN	azero	1	0	0	0	0
+zero	–NaN	azero	1	0	0	0	0
+zero	denorm	azero	1	0	0	0	0
+zero	zero	azero	0	0	0	0	0
+zero	+Norm	operand_b	0	0	0	0	0
+zero	–Norm	azero	0	0	0	0	0
–zero	+∞	pmax	1	0	0	0	0
–zero	–∞	azero	1	0	0	0	0
–zero	+NaN	azero	1	0	0	0	0
–zero	–NaN	azero	1	0	0	0	0
–zero	denorm	bzero	1	0	0	0	0
–zero	zero	bzero	0	0	0	0	0
–zero	+Norm	operand_b	0	0	0	0	0
–zero	–Norm	azero	0	0	0	0	0

Table 5-5. Floating–Point Results Summary—Min, Max (continued)

Operand A	Operand B	Result	FINV	FOVF	FUNF	FDBZ	FINX
+Norm	+∞	pmax	1	0	0	0	0
+Norm	–∞	operand_a	1	0	0	0	0
+Norm	+NaN	operand_a	1	0	0	0	0
+Norm	–NaN	operand_a	1	0	0	0	0
+Norm	denorm	operand_a	1	0	0	0	0
+Norm	zero	operand_a	0	0	0	0	0
+Norm	Norm	_Calc_	0	0	0	0	0
–Norm	+∞	pmax	1	0	0	0	0
–Norm	–∞	operand_a	1	0	0	0	0
–Norm	+NaN	operand_a	1	0	0	0	0
–Norm	–NaN	operand_a	1	0	0	0	0
–Norm	denorm	bzero	1	0	0	0	0
–Norm	zero	bzero	0	0	0	0	0
–Norm	Norm	_Calc_	0	0	0	0	0
Min							
+∞	+∞	pmax	1	0	0	0	0
+∞	–∞	nmax	1	0	0	0	0
+∞	+NaN	pmax	1	0	0	0	0
+∞	–NaN	pmax	1	0	0	0	0
+∞	denorm	bzero	1	0	0	0	0
+∞	zero	bzero	1	0	0	0	0
+∞	Norm	operand_b	1	0	0	0	0
–∞	+∞	nmax	1	0	0	0	0
–∞	–∞	nmax	1	0	0	0	0
–∞	+NaN	nmax	1	0	0	0	0
–∞	–NaN	nmax	1	0	0	0	0
–∞	denorm	nmax	1	0	0	0	0
–∞	zero	nmax	1	0	0	0	0
–∞	Norm	nmax	1	0	0	0	0
+NaN	+∞	pmax	1	0	0	0	0
+NaN	–∞	nmax	1	0	0	0	0

Table 5-5. Floating–Point Results Summary—Min, Max (continued)

Operand A	Operand B	Result	FINV	FOVF	FUNF	FDBZ	FINX
+NaN	+NaN	pmax	1	0	0	0	0
+NaN	–NaN	nmax	1	0	0	0	0
+NaN	denorm	bzero	1	0	0	0	0
+NaN	zero	bzero	1	0	0	0	0
+NaN	Norm	operand_b	1	0	0	0	0
–NaN	+∞	pmax	1	0	0	0	0
–NaN	–∞	nmax	1	0	0	0	0
–NaN	+NaN	nmax	1	0	0	0	0
–NaN	–NaN	nmax	1	0	0	0	0
–NaN	denorm	bzero	1	0	0	0	0
–NaN	zero	bzero	1	0	0	0	0
–NaN	Norm	operand_b	1	0	0	0	0
+denorm	+∞	azero	1	0	0	0	0
+denorm	–∞	nmax	1	0	0	0	0
+denorm	+NaN	azero	1	0	0	0	0
+denorm	–NaN	azero	1	0	0	0	0
+denorm	denorm	bzero	1	0	0	0	0
+denorm	zero	bzero	1	0	0	0	0
+denorm	+Norm	azero	1	0	0	0	0
+denorm	–Norm	operand_b	1	0	0	0	0
–denorm	+∞	azero	1	0	0	0	0
–denorm	–∞	nmax	1	0	0	0	0
–denorm	+NaN	azero	1	0	0	0	0
–denorm	–NaN	azero	1	0	0	0	0
–denorm	denorm	azero	1	0	0	0	0
–denorm	zero	azero	1	0	0	0	0
–denorm	+Norm	azero	1	0	0	0	0
–denorm	–Norm	operand_b	1	0	0	0	0
+zero	+∞	azero	1	0	0	0	0
+zero	–∞	nmax	1	0	0	0	0
+zero	+NaN	azero	1	0	0	0	0

Table 5-5. Floating–Point Results Summary—Min, Max (continued)

Operand A	Operand B	Result	FINV	FOVF	FUNF	FDBZ	FINX
+zero	–NaN	azero	1	0	0	0	0
+zero	denorm	bzero	1	0	0	0	0
+zero	zero	bzero	0	0	0	0	0
+zero	+Norm	azero	0	0	0	0	0
+zero	–Norm	operand_b	0	0	0	0	0
–zero	+∞	azero	1	0	0	0	0
–zero	–∞	nmax	1	0	0	0	0
–zero	+NaN	azero	1	0	0	0	0
–zero	–NaN	azero	1	0	0	0	0
–zero	denorm	azero	1	0	0	0	0
–zero	zero	azero	0	0	0	0	0
–zero	+Norm	azero	0	0	0	0	0
–zero	–Norm	operand_b	0	0	0	0	0
+Norm	+∞	operand_a	1	0	0	0	0
+Norm	–∞	nmax	1	0	0	0	0
+Norm	+NaN	operand_a	1	0	0	0	0
+Norm	–NaN	operand_a	1	0	0	0	0
+Norm	denorm	bzero	1	0	0	0	0
+Norm	zero	bzero	0	0	0	0	0
+Norm	Norm	_Calc_	0	0	0	0	0
–Norm	+∞	operand_a	1	0	0	0	0
–Norm	–∞	nmax	1	0	0	0	0
–Norm	+NaN	operand_a	1	0	0	0	0
–Norm	–NaN	operand_a	1	0	0	0	0
–Norm	denorm	operand_a	1	0	0	0	0
–Norm	zero	operand_a	0	0	0	0	0
–Norm	Norm	_Calc_	0	0	0	0	0

Table 5-6 shows the floating-points results summary for convert to unsigned.

Table 5-6. Floating-Point Results Summary—Convert to Unsigned

Operand B	Integer Result efsctui[z]	Fractional Result efsctuf	F INV	FOVF	FUNF	FDBZ	F INX
+ ∞	0xFFFF_FFFF	0xFFFF_FFFF	1	0	0	0	0
- ∞	zero	zero	1	0	0	0	0
+NaN	zero	zero	1	0	0	0	0
–NaN	zero	zero	1	0	0	0	0
denorm	zero	zero	1	0	0	0	0
zero	zero	zero	0	0	0	0	0
+norm	_Calc_	_Calc_	*	0	0	0	*
–norm	zero	zero	0	0	0	0	0

Table 5-7 shows the floating-points results summary for convert to signed.

Table 5-7. Floating-Point Results Summary—Convert to Signed

Operand B	Integer Result efsctsWi[z]	Fractional Result efsctsf	F INV	FOVF	FUNF	FDBZ	F INX
+ ∞	0x7FFF_FFFF	0x7FFF_FFFF	1	0	0	0	0
– ∞	0x8000_0000	0x8000_0000	1	0	0	0	0
+NaN	zero	zero	1	0	0	0	0
–NaN	zero	zero	1	0	0	0	0
denorm	zero	zero	1	0	0	0	0
zero	zero	zero	0	0	0	0	0
+norm	_Calc_	_Calc_	*	0	0	0	*
–norm	_Calc_	_Calc_	*	0	0	0	*

Table 5-8 shows the floating-points results summary for convert from unsigned.

Table 5-8. Floating-Point Results Summary—Convert from Unsigned

Operand B	Integer Source efscfui	Fractional Source efscfu	F INV	FOVF	FUNF	FDBZ	F INX
zero	zero	zero	0	0	0	0	0
norm	_Calc_	_Calc_	0	0	0	0	*

Table 5-9 shows the floating-points results summary for convert from signed.

Table 5-9. Floating-Point Results Summary—Convert from Signed

Operand B	Integer Source efscfsi	Fractional Source efscfsf	F INV	FOVF	FUNF	FDBZ	F INX
zero	zero	zero	0	0	0	0	0
norm	_Calc_	_Calc_	0	0	0	0	*

Table 5-10 shows the floating-points results summary for fabs, fnabs, fneg.

Table 5-10. Floating-Point Results Summary—fabs, fnabs, fneg

Operand A	fabs	fnabs	fneg	F INV	FOVF	FUNF	FDBZ	F INX
∞	+ ∞	- ∞	–A	1	0	0	0	0
NaN	Sign bit cleared	Sign bit set	–A	1	0	0	0	0
denorm	Sign bit cleared	Sign bit set	–A	1	0	0	0	0
zero	zero	zero	zero	0	0	0	0	0
norm	norm	norm	norm	0	0	0	0	0

Table 5-11 shows the floating-point results summary for convert from half-precision.

Table 5-11. Floating-point Results Summary—Convert from half-precision

Operand B	e[v]fscfh	F INV	FOVF	FUNF	FDBZ	F INX
∞	bmax	1	0	0	0	0
NaN	bmax	1	0	0	0	0
denorm	bzero	1	0	0	0	0
zero	bzero	0	0	0	0	0
+norm	_Calc_	0	0	0	0	*
–norm	_Calc_	0	0	0	0	*

Table 5-12 shows the floating-point results summary for convert from half-precision.

Table 5-12. Floating-point Results Summary—Convert to half-precision

Operand B	e[v]fscth	F INV	FOVF	FUNF	FDBZ	F INX
∞	bmax_{hp}	1	0	0	0	0
NaN	bmax_{hp}	1	0	0	0	0
denorm	bzero	1	0	0	0	0
zero	bzero	0	0	0	0	0
+norm	_Calc_	0	*	*	0	*
–norm	_Calc_	0	*	*	0	*

5.5 EFPU Instruction Timing

Instruction timing in number of processor clock cycles for EFPU instructions are shown in Table 5-13, and Table 5-14. Pipelined instructions are shown with cycles of total latency and throughput cycles. Divide instructions are not pipelined and block other instructions from executing during divide execution.

Instruction pipelining in the CPU is affected by the possibility of a floating-point instruction generating an exception. A load or store class instruction that follows an EFPU instruction stalls until it can be ensured that no previous instruction can generate a floating-point exception. This determination is based on which floating-point exception enable bits are set (FINVE, FOVFE, FUNFE, FDBZE, and FINXE) and at what point in the FPU pipeline an exception can be guaranteed to not occur. Invalid input operands are detected in the first stage of the pipeline, while underflow, overflow, and inexactness are determined later in the pipeline. Best overall performance occurs when either floating-point exceptions are disabled, or when load and store class instructions are scheduled such that previous floating-point instructions have already resolved the possibility of exceptional results.

5.5.1 EFPU Single-Precision Vector Floating-Point Instruction Timing

Instruction timing for EFPU vector floating-point instructions is shown in Table 5-13. The table is sorted by opcode. The number of stall cycles for **evfsdiv** and **evfssqrt** is (latency) cycles.

Table 5-13. EFPU Vector Floating-Point Instruction Timing

Instruction	Latency	Throughput	Comments
evfsabs	4	1	—
evfsadd	4	1	—
evfsaddx	4	1	—
evfsaddsub	4	1	—
evfsaddsubx	4	1	—
evfscfh	4	1	—
evfscfsf	4	1	—
evfscfsi	4	1	—
evfscfuf	4	1	—
evfscfui	4	1	—
evfscmpeq	4	1	—
evfscmpgt	4	1	—
evfscmplt	4	1	—
evfscth	4	1	—
evfsctsf	4	1	—
evfsctsi	4	1	—
evfsctsiz	4	1	—
evfsctuf	4	1	—
evfsctui	4	1	—
evfsctuiz	4	1	—
evfsdiff	4	1	
evfsdiffsum	4	1	
evfsdiv	13	13	blocking, no overlap with next inst.
evfsmax	4	1	

Table 5-13. EFPU Vector Floating-Point Instruction Timing (continued)

Instruction	Latency	Throughput	Comments
evfsmin	4	1	
evfsmadd	4	1[1]	dest also used as source
evfsmsub	4	1[1]	dest also used as source
evfsmul	4	1	—
evfsmule	4	1	—
evfsmulo	4	1	—
evfsmulx	4	1	—
evfsnabs	4	1	—
evfsneg	4	1	—
evfsnmadd	4	1[1]	dest also used as source
evfsnmsub	4	1[1]	dest also used as source
evfssqrt	15	15	blocking, no overlap with next inst.
evfssub	4	1	—
evfssubx	4	1	—
evfssubadd	4	1	—
evfssubaddx	4	1	—
evfssum	4	1	—
evfssumdiff	4	1	—
evfststeq	4	1	—
evfststgt	4	1	—
evfststlt	4	1	—

[1] Destination register is also a source register, so for full throughput, back-to-back operations must use a different dest reg.

5.5.2 EFPU Single-precision Scalar Floating-Point Instruction Timing

Instruction timing for EFPU single-precision scalar floating-point instructions is shown in Table 5-14. The table is sorted by opcode.

Table 5-14. EFPU Single-precision Scalar Floating-Point Instruction Timing

Instruction	Latency	Throughput	Comments
efsabs	4	1	—
efsadd	4	1	—
efscfh	4	1	—
efscfsf	4	1	—

Table 5-14. EFPU Single-precision Scalar Floating-Point Instruction Timing (continued)

Instruction	Latency	Throughput	Comments
efscfsi	4	1	—
efscfuf	4	1	—
efscfui	4	1	—
efscmpeq	4	1	—
efscmpgt	4	1	—
efscmplt	4	1	—
efscth	4	1	—
efsctsf	4	1	—
efsctsi	4	1	—
efsctsiz	4	1	—
efsctuf	4	1	—
efsctui	4	1	—
efsctuiz	4	1	—
efsdiv	13	13	blocking, no execution overlap with next instruction
efsmadd	4	1[1]	dest also used as source
efsmsub	4	1[1]	dest also used as source
efsmax	4	1	
efsmin	4	1	
efsmul	4	1	—
efsnabs	4	1	—
efsneg	4	1	—
efsnmadd	4	1[1]	dest also used as source
efsnmsub	4	1[1]	dest also used as source
efssqrt	15	15	blocking, no overlap with next inst.
efssub	4	1	—
efststeq	4	1	—
efststgt	4	1	—
efststlt	4	1	—

Note:

[1] Destination register is also a source register, so for full throughput, back-to-back operations must use a different dest reg.

5.6 Instruction Forms and Opcodes

Table 5-15 gives the division of the opcode space for the EFPU instructions. This is the architectural assignment; not all instructions are implemented in all versions of the CPU.

Table 5-15. Opcode Space Division

Opcode Bits		Instruction Class
0–5	**21–28**	
4	0101 00*xx*	Embedded vector floating-point instructions
4	0101 010*x*	Embedded vector floating-point instructions
4	0101 0110	Embedded scalar floating-point single-precision instructions
4	0101 0111	Reserved (Embedded scalar floating-point double-precision instructions)[1]
4	0101 10*xx*	Embedded scalar floating-point single-precision instructions
4	0101 11*xx*	Reserved (Embedded scalar floating-point double-precision instructions)[1]

[1] Attempted execution of a defined EFP double-precision instruction results in an unimplemented instruction execution if MSR[SPE] = 1 or an EFPU unavailable except if MSR[SPE] = 0.

Table 5-16 shows the embedded vector floating-point instruction opcodes.

Table 5-16. Embedded Vector Floating-Point Instruction Opcodes

Instruction	Opcode Bits						Comments
	0–5	**6–10**	**11–15**	**16–20**	**21–24**	**25–31**	
evfsadd	4	rD	rA	rB	0101	0000000	—
evfssub	4	rD	rA	rB	0101	0000001	rA – rB
evfsmadd	4	rD	rA	rB	0101	0000010	—
evfsmsub	4	rD	rA	rB	0101	0000011	—
evfsabs	4	rD	rA	00000	0101	0000100	—
evfsnabs	4	rD	rA	00000	0101	0000101	—
evfsneg	4	rD	rA	00000	0101	0000110	—
evfssqrt	4	rD	rA	00000	0101	0000111	—
evfsmul	4	rD	rA	rB	0101	0001000	—
evfsdiv	4	rD	rA	rB	0101	0001001	—
evfsnmadd	4	rD	rA	rB	0101	0001010	—
evfsnmsub	4	rD	rA	rB	0101	0001011	—
evfscmpgt	4	crfD 00	rA	rB	0101	0001100	—
evfscmplt	4	crfD 00	rA	rB	0101	0001101	—
evfscmpeq	4	crfD 00	rA	rB	0101	0001110	—
—	4	—	—	—	0101	0001111	—

Table 5-16. Embedded Vector Floating-Point Instruction Opcodes (continued)

Instruction	Opcode Bits						Comments
	0–5	6–10	11–15	16–20	21–24	25–31	
evfscfui	4	rD	00000	rB	0101	0010000	—
evfscfsi	4	rD	00000	rB	0101	0010001	—
evfscfh	4	rD	00100	rB	0101	0010001	—
evfscfuf	4	rD	00000	rB	0101	0010010	—
evfscfsf	4	rD	00000	rB	0101	0010011	—
evfsctui	4	rD	00000	rB	0101	0010100	—
evfsctsi	4	rD	00000	rB	0101	0010101	—
evfscth	4	rD	00100	rB	0101	0010101	—
evfsctuf	4	rD	00000	rB	0101	0010110	—
evfsctsf	4	rD	00000	rB	0101	0010111	—
evfsctuiz	4	rD	00000	rB	0101	0011000	—
—	4	—	—	—	0101	0011001	—
evfsctsiz	4	rD	00000	rB	0101	0011010	—
—	4	—	—	—	0101	0011011	—
evfststgt	4	crfD 00	rA	rB	0101	0011100	—
evfststlt	4	crfD 00	rA	rB	0101	0011101	—
evfststeq	4	crfD 00	rA	rB	0101	0011110	—
—	4	—	—	—	0101	0011111	—
evfsmax	4	rD	rA	rB	0101	0100000	—
evfsmin	4	rD	rA	rB	0101	0100001	—
evfsaddsub	4	rD	rA	rB	0101	0100010	—
evfssubadd	4	rD	rA	rB	0101	0100011	rA – rB; rA + rB
evfssum	4	rD	rA	rB	0101	0100100	—
evfsdiff	4	rD	rA	rB	0101	0100101	—
evfssumdiff	4	rD	rA	rB	0101	0100110	—
evfsdiffsum	4	rD	rA	rB	0101	0100111	—
evfsaddx	4	rD	rA	rB	0101	0101000	—
evfssubx	4	rD	rA	rB	0101	0101001	—
evfsaddsubx	4	rD	rA	rB	0101	0101010	—
evfssubaddx	4	rD	rA	rB	0101	0101011	rA – rB; rA + rB
evfsmulx	4	rD	rA	rB	0101	0101100	—

Table 5-16. Embedded Vector Floating-Point Instruction Opcodes (continued)

Instruction	Opcode Bits						Comments
	0–5	6–10	11–15	16–20	21–24	25–31	
—	4	rD	rA	rB	0101	0101101	—
evfsmule	4	rD	rA	rB	0101	0101110	—
evfsmulo	4	rD	rA	rB	0101	0101111	—

Table 5-17 shows the embedded vector floating-point instruction opcodes.

Table 5-17. Embedded Scalar Single-Precision Floating-Point Instruction Opcodes

Instruction	Opcode Bits						Comments
	0–5	6–10	11–15	16–20	21–24	25–31	
efsmax	4	rD	rA	rB	0101	0110000	—
efsmin	4	rD	rA	rB	0101	0110001	—
efsadd	4	rD	rA	rB	0101	1000000	—
efssub	4	rD	rA	rB	0101	1000001	rA – rB
efsmadd	4	rD	rA	rB	0101	1000010	—
efsmsub	4	rD	rA	rB	0101	1000011	—
efsabs	4	rD	rA	00000	0101	1000100	—
efsnabs	4	rD	rA	00000	0101	1000101	—
efsneg	4	rD	rA	00000	0101	1000110	—
efssqrt	4	rD	rA	00000	0101	1000111	—
efsmul	4	rD	rA	rB	0101	1001000	—
efsdiv	4	rD	rA	rB	0101	1001001	—
efsnmadd	4	rD	rA	rB	0101	1001010	—
efsnmsub	4	rD	rA	rB	0101	1001011	—
efscmpgt	4	crfD 00	rA	rB	0101	1001100	—
efscmplt	4	crfD 00	rA	rB	0101	1001101	—
efscmpeq	4	crfD 00	rA	rB	0101	1001110	—
efscfd	4	rD	00000	rB	0101	1001111	optional, not implemented
efscfui	4	rD	00000	rB	0101	1010000	—
efscfsi	4	rD	00000	rB	0101	1010001	—
efscfh	4	rD	00100	rB	0101	1010001	—
efscfuf	4	rD	00000	rB	0101	1010010	—
efscfsf	4	rD	00000	rB	0101	1010011	—

Table 5-17. Embedded Scalar Single-Precision Floating-Point Instruction Opcodes (continued)

Instruction	Opcode Bits						Comments
	0–5	6–10	11–15	16–20	21–24	25–31	
efsctui	4	rD	00000	rB	0101	1010100	—
efsctsi	4	rD	00000	rB	0101	1010101	—
efscth	4	rD	00100	rB	0101	1010101	—
efsctuf	4	rD	00000	rB	0101	1010110	—
efsctsf	4	rD	00000	rB	0101	1010111	—
efsctuiz	4	rD	00000	rB	0101	1011000	—
—	4	—	—	—	0101	1011001	—
efsctsiz	4	rD	00000	rB	0101	1011010	—
—	4	—	—	—	0101	1011011	—
efststgt	4	crfD 00	rA	rB	0101	1011100	—
efststlt	4	crfD 00	rA	rB	0101	1011101	—
efststeq	4	crfD 00	rA	rB	0101	1011110	—
—	4	—	—	—	0101	1011111	—

Chapter 6
Signal Processing Extension (SPE)

This chapter provides an overview of the signal processing engine, version 2.1, which is designed to accelerate signal processing applications normally suited to DSP operation. This is accomplished using short vectors (two, four, or eight elements) within 64-bit GPRs and using single instruction multiple data (SIMD) operations to perform the requisite computations. SPE2.1 also architects an accumulator register to allow for certain back to back operations without loop unrolling. SPE2.1 is fully backward compatible with the original SPE. The remainder of this document uses the term SPE to refer to version 2.1 unless otherwise noted.

6.1 Nomenclature and Conventions

Several conventions regarding nomenclature are used in this chapter:

- Due to historical precedent, the terms SPE and SIMD are sometimes used interchangeably.
- The signal processing engine is abbreviated as SPE.
- All register bit numbering is 64-bit with bit 0 being the most significant bit. Registers that are only 32-bit define bit 32 as the most significant bit. For both 32-bit and 64-bit registers, bit 63 is the least significant bit.
- Bits 0–31 of a 64-bit register are referenced as word 0, upper word, even word, or high word element of the register. Bits 32–63 are referred to as word 1, lower word, odd word, or low word element of the register. Each word is an element of a 64-bit GPR.
- Bits 0–15 of a 64-bit register are referenced as half word 0. Bits 16–31 are referred to as half word 1. Bits 32–47 are referenced as half word 2. Bits 48–63 are referred to as half word 3. Each half word is an element of a 64-bit GPR.
- Bits 0–7 of a 64-bit register are referenced as byte 0. Bits 8–15 are referred to as byte 1. Bits 16–23 are referenced as byte 2. Bits 24–31 are referred to as byte 3. Bits 32–39 are referred to as byte 4. Bits 40–47 are referenced as byte 5. Bits 48–55 are referred to as byte 6. Bits 56–63 are referenced as byte 7. Each byte is an element of a 64-bit GPR.
- Bits 0–15 and bits 32–47 are referenced as even half words. Bits 16–31 and bits 48–63 are referenced as odd half words. Bits 0–15 and bits 16–31 are referenced as upper half words. Bits 32–47 and bits 48–63 are referenced as lower half words.
- Mnemonics for SPE instructions generally begin with the letters ‘ev’ (embedded vector).

Table 6-1 shows RTL conventions that are used in this chapter.

Table 6-1. RTL Notation

Notation	Meaning
$\times_{sf}$	Signed fractional multiplication. Result of multiplying 2 quantities having bit lengths x and y taking the least significant x + y – 1 bits of the product and concatenating a 0 to the least significant bit forming a signed fractional result of x + y bits.
$\times_{si}$	Signed integer multiplication
$\times_{su}$, $\times_{sui}$	Signed by Unsigned multiplication (same for int and frac)
$\times_{ui}$	Unsigned integer multiplication
<<	Logical shift left. x << y shifts value x left by y bits, leaving zeros in the vacated bits.
>>	Logical shift right. x >> y shifts value x right by y bits, leaving zeros in the vacated bits.

6.2 SPE Programming Model

The e200z760n3 core provides a register file with thirty-two 64-bit registers. The embedded category in the Power ISA instructions operate on the lower (least significant) 32 bits of the 64-bit register. SPE instructions generally take elements from each source register and operate on them with the corresponding elements of a second source register (and/or the accumulator) to produce results. Results are placed in the destination register and/or the accumulator. Vector instructions (i.e. produce results of more than one element) provide results for each element that are independent of the computation of the other elements. These instructions can also be used to perform scalar DSP operations by ignoring the results of the upper 32-bit half of the register file.

SPE compare instructions and set instructions with record store the comparison result into the condition register (CR). The meaning of the CR bits are now overloaded for SPE operations. SPE compare instructions specify a CR field, two source registers, and the type of compare: greater than, less than, or equal. Two bits of the CR field are written with the result of the vector compare: one for each of the high and low 32-bits of the result. The remaining two bits reflect the ANDing and ORing of the vector compare results. An additional set of compare instructions (**evset**$_{\mathbf{xx}}$[.]) return a set of Boolean values into a destination register, allowing for subsequent predicated computational operations, such as a select operation to be performed.

A partially visible accumulator register is architected for the SPE integer and fractional multiply accumulate forms of instructions. Its usage is described in Section 6.2.2, “Accumulator Register.”

6.2.1 GPR Registers

The SPE requires a GPR register file with thirty-two 64-bit registers. For 32-bit implementations, the embedded category of the Power ISA instructions that normally operate on a 32-bit register, access and change only the least significant 32-bits of the GPRs. They leave the most significant 32-bits unchanged. SPE instructions view the 64-bit register as being composed of a vector of elements, each of which is 32 bits, 16 bits, or 8 bits wide.

Nomenclature is as follows:

- The most significant 32 bits are called word 0 (W0), the upper word, high word or even word.
- The least significant 32 bits are called word 1 (W1), the lower word, low word or odd word.
- Half word elements are called half word 0, 1, 2, or 3, from most significant to least significant.
- Byte elements are called byte 0, 1, 2, 3, 4, 5, 6, or 7, from most significant to least significant.

Unless otherwise specified, SPE instructions write all 64 bits of the destination register.

Figure 6-1 shows vector storage in GPRs.

<table>
<tr><th></th><th>0 7</th><th>8 15</th><th>16 23</th><th>24 31</th><th>32 39</th><th>40 47</th><th>48 55</th><th>56 63</th></tr>
<tr><td>GPR</td><td colspan="8">Double word</td></tr>
<tr><td>GPR</td><td colspan="4">Upper word (word 0)</td><td colspan="4">Lower word (word 1)</td></tr>
<tr><td>GPR</td><td colspan="2">half word 0</td><td colspan="2">half word 1</td><td colspan="2">half word 2</td><td colspan="2">half word 3</td></tr>
<tr><td>GPR</td><td>byte 0</td><td>byte 1</td><td>byte 2</td><td>byte 3</td><td>byte 4</td><td>byte 5</td><td>byte 6</td><td>byte 7</td></tr>
</table>

Figure 6-1. Vector Storage in GPRs

6.2.2 Accumulator Register

The accumulator is a 64-bit register that allows the back-to-back execution of dependent MAC and dot product instructions, something that is found in the inner loops of DSP code such as FIR and FFT 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 that is specified as part of the instruction. However, the accumulator has to be explicitly initialized when starting a new accumulation loop.

The accumulator is for used the following kinds of instructions:

- Certain integer/fractional accumulation
- Multiply accumulate (MAC)
- Dot product
- Summation forms

Based upon the type of instruction, the accumulator can hold either a single 64-bit value or a vector of two 32-bit elements, a vector of four 16-bit elements, or vector of eight 8-bit elements. In addition, for certain instructions, the accumulator can be updated along with the destination register.

Figure 6-2 shows accumulator storage.

<table>
<tr><th></th><th colspan="2">0 15</th><th colspan="2">16 31</th><th colspan="2">32 47</th><th colspan="2">48 63</th></tr>
<tr><td>ACC</td><td colspan="8">Double word</td></tr>
<tr><td>ACC</td><td colspan="4">Upper word</td><td colspan="4">Lower word</td></tr>
<tr><td>ACC</td><td colspan="2">half word 0</td><td colspan="2">half word 1</td><td colspan="2">half word 2</td><td colspan="2">half word 3</td></tr>
<tr><td>ACC</td><td>byte 0</td><td>byte 1</td><td>byte 2</td><td>byte 3</td><td>byte 4</td><td>byte 5</td><td>byte 6</td><td>byte 7</td></tr>
</table>

Figure 6-2. Accumulator Storage

An example of a MAC instruction is **evmhossfaaw r**D,**r**A,**r**B. In this instruction, the least significant 16 bits of **r**A and **r**B are multiplied for both elements of the vector; the result is shifted left one bit and added to the accumulator; and the result is possibly saturated to 32 bits in case of overflow. The final result is placed both in the accumulator and also in **r**D. Therefore, the result of this instruction can be used by accessing **r**D.

To read the accumulator contents into a **register**, the **evmar** instruction is used. To initialize the accumulator, the **evmra** instruction or another instruction targeting the accumulator such as **evsplati**$_{xx}$**a** is used.

6.2.3 SPE Status and Control Register (SPEFSCR)

The e200 z760n3 core implements the SPEFSCR register for status reporting and control of SPE instructions. This register is also used by the embedded floating-point units. Status and control bits are shared for floating-point operations and SPE operations. The SPEFSCR register is implemented as special purpose register (SPR) number 512 and is read and written by the **mfspr** and **mtspr** instructions in both user and supervisor mode. SPE instructions affect both the high element (bits 0–1) and low element status flags (bits 16–17).

Figure 6-3 shows the SPEFSCR.

SPR 512 Access: Read/Write

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R W	SOVH	OVH	FGH	FXH	FINVH	FDBZH	FUNFH	FOVFH	—	RM	FINXS	FINVS	FDBZS	FUNFS	FOVFS	MODE
Reset	All zeros															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R W	SOV	OV	FG	FX	FINV	FDBZ	FUNF	FOVF	—	FINXE	FINVE	FDBZE	FUNFE	FOVFE	FRMC	
Reset	All zeros															

Figure 6-3. SPE/EFPU Status and Control Register (SPEFSCR)

Table 6-2 describes the SPEFSCR bits.

Table 6-2. SPE Status and Control Register

Bits	Name	Description
0 (32)	SOVH	Summary Integer Overflow High The SOVH bit is set to 1 whenever an instruction sets OVH. The SOVH bit remains set until it is cleared by a **mtspr** instruction specifying the SPEFSCR register.
1 (33)	OVH	Integer Overflow High The OVH bit is set to 1 whenever an integer or fractional SPE instruction signals an overflow in the upper half of the result.
2 (34)	FGH	Embedded Floating-Point Guard bit High Defined by Embedded Floating-Point APUs.
3 (35)	FXH	Embedded Floating-Point Inexact bit High Defined by Embedded Floating-Point APUs.

Table 6-2. SPE Status and Control Register (continued)

Bits	Name	Description
4 (36)	FINVH	Embedded Floating-Point Invalid Operation/Input error High Defined by Embedded Floating-Point APUs.
5 (37)	FDBZH	Embedded Floating-Point Divide by Zero High Defined by Embedded Floating-Point APUs.
6 (38)	FUNFH	Embedded Floating-Point Underflow High Defined by Embedded Floating-Point APUs.
7 (39)	FOVFH	Embedded Floating-Point Overflow High Defined by Embedded Floating-Point APUs.
8 (40)	—	Reserved
9 (41)	RM	Rounding Mode - Fixed Point 0 Normal Rounding (Biased-rounding), rounding performed by adding 1/2 lsb 1 Round to Nearest Even Rounding (convergent rounding), round to nearest even value
10 (42)	FINXS	Embedded Floating-Point Inexact Sticky Flag Defined by Embedded Floating-Point APUs.
11 (43)	FINVS	Embedded Floating-Point Invalid Operation Sticky Flag Defined by Embedded Floating-Point APUs.
12 (44)	FDBZS	Embedded Floating-Point Divide by Zero Sticky Flag Defined by Embedded Floating-Point APUs.
13 (45)	FUNFS	Embedded Floating-Point Underflow Sticky Flag Defined by Embedded Floating-Point APUs.
14 (46)	FOVFS	Embedded Floating-Point Overflow Sticky Flag Defined by Embedded Floating-Point APUs.
15 (47)	MODE	Embedded Floating-Point Operating Mode Defined by Embedded Floating-Point APUs.
16 (48)	SOV	Summary Integer Overflow The SOV bit is set to 1 whenever an instruction sets OV. The SOV bit remains set until it is cleared by an **mtspr** instruction specifying the SPEFSCR register.
17 (49)	OV	Integer Overflow The OV bit is set to 1 whenever an integer or fractional SPE instruction signals an overflow in the low element result.
18 (50)	FG	Embedded Floating-Point Guard bit (low/scalar) Defined by Embedded Floating-Point APUs.
19 (51)	FX	Embedded Floating-Point Inexact bit (low/scalar) Defined by Embedded Floating-Point APUs.
20 (52)	FINV	Embedded Floating-Point Invalid Operation / Input error (low/scalar) Defined by Embedded Floating-Point APUs.
21 (53)	FDBZ	Embedded Floating-Point Divide by Zero (low/scalar) Defined by Embedded Floating-Point APUs.
22 (54)	FUNF	Embedded Floating-Point Underflow (low/scalar) Defined by Embedded Floating-Point APUs.

Table 6-2. SPE Status and Control Register (continued)

Bits	Name	Description
23 (55)	FOVF	Embedded Floating-Point Overflow (low/scalar) Defined by Embedded Floating-Point APUs.
24 (56)	—	Reserved
25 (57)	FINXE	Embedded Floating-Point Round (Inexact) Exception Enable Defined by Embedded Floating-Point APUs.
26 (58)	FINVE	Embedded Floating-Point Invalid Operation / Input Error Exception Enable Defined by Embedded Floating-Point APUs.
27 (59)	FDBZE	Embedded Floating-Point Divide by Zero Exception Enable Defined by Embedded Floating-Point APUs.
28 (60)	FUNFE	Embedded Floating-Point Underflow Exception Enable Defined by Embedded Floating-Point APUs.
29 (61)	FOVFE	Embedded Floating-Point Overflow Exception Enable Defined by Embedded Floating-Point APUs.
30–31 (62–63)	FRMC	Embedded Floating-Point Rounding Mode Control Defined by Embedded Floating-Point APUs.

6.2.3.1 Context Switch

When a context switch occurs, the OS process must explicitly save the accumulator as part of the context of the swapped-out task and then explicitly load the accumulator from the context of the new task that is being swapped in. When the old task is restarted, its accumulator must be restored before restarting the task.

6.2.4 GPRs and Power ISA Instructions

The e200 z760n3 core implements the 32-bit forms of the embedded category instructions in the Power ISA. All 32-bit Power ISA instructions operate upon the lower half of the 64-bit GPR. These instructions do not affect the upper half of a GPR.

6.2.5 SPE Available Bit in MSR

MSR[SPE] is defined as the SPE available bit. If this bit is clear and software attempts to execute any of the SPE instructions other than the **brinc** instruction (which does not affect the upper 32-bits of a GPR), the SPE unavailable exception is taken. If this bit is set, software can execute any of the SPE instructions.

6.2.6 SPE Exception Bit in ESR

ESR[SPE] is defined as the SPE exception bit. This bit is set whenever the processor takes an exception related to the execution of the SPE instructions.

6.2.7 Data Formats

The SPE provides two different data formats, integer and fractional. Integer data formats can be treated as signed or unsigned quantities. Fractional data formats are usually treated as signed quantities

6.2.7.1 Integer Format

Integer data format is the same as what is conventionally used in computing.

Unsigned integers consist of 8, 16, 32, or 64-bit binary integer values. The largest representable value is $2^n - 1$ where n represents the number of bits in the value. The smallest representable value is 0. Certain computations that produce values larger than $2^n - 1$ or smaller than 0 set OV or OVH in the SPEFSCR.

Signed integers consist of 8, 16, 32, or 64-bit binary values in twos-complement form. The largest representable value is $2^{n-1} - 1$ where n represents the number of bits in the value. The smallest representable value is -2^{n-1}. Certain computations that produce values larger than $2^{n-1} - 1$ or smaller than -2^{n-1} set OV or OVH in the SPEFSCR.

6.2.7.2 Fractional Format

Fractional data format is the same that is conventionally used for DSP fractional arithmetic. Fractional data is useful for representing data converted from analog devices.

Unsigned fractions consist of 16, 32, or 64-bit binary fractional values that range from 0 to less than 1. Unsigned fractions place the decimal point immediately to the left of the most significant bit. The most significant bit of the value represents the value 2^{-1}, the next most significant bit represents the value 2^{-2} and so on. The largest representable value is $1 - 2^{-n}$ where n represents the number of bits in the value. The smallest representable value is 0. Certain computations that produce values larger than $1 - 2^{-n}$ or smaller than 0 set OV or OVH in the SPEFSCR. SPE does not contain explicit instructions that manipulate unsigned fractional data. Unsigned integer forms produce the same bit exact results as unsigned fractional values would, therefore unsigned fractional instruction forms are not defined for SPE.

Signed fractions consist of 16, 32, or 64-bit binary fractional values in twos complement form that range from –1 to less than 1. Signed fractions in 1.31 or 1.63 format place the decimal point immediately to the right of the most significant bit. The largest representable value is $1 - 2^{-(n-1)}$ where n represents the number of bits in the value. The smallest representable value is –1. Certain computations that produce values larger than $1 - 2^{-(n-1)}$or smaller than –1 set OV or OVH in the SPEFSCR. Multiplication of two signed fractional values causes the result to be shifted left one bit to remove the resultant redundant sign bit in the product. In this case, a 0 bit is concatenated as the least significant bit of the shifted result.

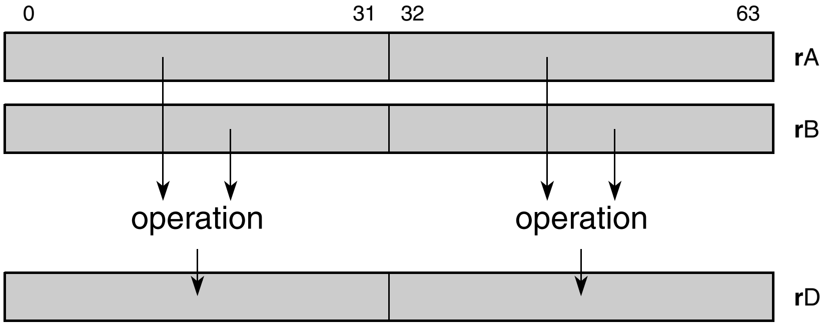
Guarded fractional representations are also available in 33.31 format and in 17.47 format for a subset of operations, providing for significant guarding capabilities.

6.2.8 Computational Operations

SPE supports several different computational capabilities. Both modulo and saturation results can be performed. Modulo results produce truncation of the overflow bits in a calculation. Saturation provides a maximum or minimum representable value (for the data type) for overflow or underflow respectively.

Instructions are provided for a wide range of computational capability. The operation types can be divided into several basic categories:

- Simple vector instructions. These instructions use the corresponding elements of the operands to produce a vector result that is placed in the destination register, the accumulator, or both.



  - arithmetic, logical, shift, and rotate of vector elements
  - Averaging, summation, rounding, min, max, sum of absolute differences, absolute differences, saturation, operations
  - vector permutation, packing, unpacking, merge, swap, extraction, interleave, de-interleave operations
- Multiply and accumulate instructions. These instructions perform multiply operations, optionally add the result to the accumulator and place the result into the destination register and optionally into the accumulator. These instructions are composed of different multiply forms, data formats and data accumulate options.
- Dot product instructions. These instructions perform multiple multiply operations, optionally add the results to the accumulator, and place the result into the destination register and optionally into the Accumulator. These instructions are composed of different forms, data formats and data accumulate options.
- Load and store instructions. These instructions provide load and store capabilities for moving data to and from memory. A variety of forms are provided that position data for efficient computation.
- Compare instructions and set instructions.
- Miscellaneous instructions. These instructions perform miscellaneous functions such as field manipulation, bit-reversed and circular incrementing, count leading, and more.

6.2.8.1 Simple Vector Arithmetic Instructions

Simple vector arithmetic instructions are outlined in Table 6-3.

Table 6-3. Simple Vector Arithmetic Instructions

Basic Operation	Variants	Description	ACC?
Absolute Value	evabsb, evabsh, evabs, evabsd	absolute value byte, half word, word, double word elements	—
	evabsbs, evabshs, evabss, evabsds	abs b, h, w, d with saturation	—

Table 6-3. Simple Vector Arithmetic Instructions

Basic Operation	Variants	Description	ACC?
Absolute Difference	evabsdifsb, evabsdifsh, evabsdifsw, evabsdifub, evabsdifuh, evabsdifuw	absolute difference signed/unsigned byte, half word, word elements	—
Add	evaddb, evaddh, evaddw, evaddd	add byte, half word, word, double word elements	—
	evaddbss, evaddhss, evaddwss, evadddss evaddbus, evaddhus, evaddwus, evadddus	add byte, half word, word, double word elements with signed or unsigned saturation	—
	evaddhx, evaddhxss, evaddhxus	add exchanged half word elements with optional signed or unsigned saturation. The even and odd half word elements of operand rA are pairwise exchanged before adding	—
	evaddwx, evaddwxss, evaddwxus	add exchanged word elements with optional signed or unsigned saturation. The high and low word elements of operand rA are exchanged before adding	—
	evaddib, evaddih, evaddiw	add unsigned imm value UIMM to all elements	—
	evaddsmiaaw, evaddssiaaw, evaddumiaaw, evaddusiaaw	add word elements from rA and Accumulator using signed/unsigned modulo/saturation operations, results into rD and Accumulator	Y
	evaddsmiaa, evaddssiaa, evaddusiaa	add 64-bit value in rA and Accumulator with optional signed/unsigned saturation, result into rD and Accumulator	Y
AddSubf	evadd2subf2h, evadd2subf2hss	add for upper 2 half word elements, subf for lower 2 elements, with optional signed saturation.	—
	evaddsubfh, evaddsubfhss	add for even half word elements, subf for odd elements, with optional signed saturation.	—
	evaddsubfhx, evaddsubfhxss	The even and odd half word elements of operand rA are pairwise exchanged and then the resulting even elements are added and the odd elements are subtracted to/from elements in rB, with optional signed saturation.	—
	evaddsubfw, evaddsubfwss	The high word element of rA is added and the low word element of rA is subtracted to/from the corresponding element of rB, with optional signed saturation.	—
	evaddsubfwx, evaddsubfwxss	The word elements of rA are exchanged and then the resulting high word element is added and low word elements is subtracted to/from word elements of rB, with optional signed saturation.	—
Average	evavgbs, evavghs, evavgws, evavgds, evavgbsr, evavghsr, evavgwsr, evavgdsr evavgbu, evavghu, evavgwu, evavgdu evavgbur, evavghur, evavgwur, evavgdur	compute the average of corresponding elements in rA and rB, signed/unsigned with optional rounding	—
Count Leading	evcntlsh, evcntlzh evcntlsw, evcntlzw	count leading sign/zero bits in each half word count leading sign/zero bits in each word	—

Table 6-3. Simple Vector Arithmetic Instructions

Basic Operation	Variants	Description	ACC?
Divide	evdivws, evdivwu, evdivwsf, evdivwuf evdivs, evdivu	32 / 32 → 32 signed, unsigned integer 32 / 32 → 32 signed, unsigned fractional 64 / 64 → 64, signed, unsigned	—
Extend	evextsb, evextzb	the low byte of each word element in rA is sign/zero extended to a word and placed into rD	—
	evextsbh	the odd bytes of rA are sign extended to half words and placed into rD	—
	evextsh, *evextzh* (use evclrh)	the odd half word elements in rA are sign/zero extended to a word and placed into rD	—
	evextsw	the low word element in rA is sign extended to 64-bits and placed into rD	—
Maximum	evmaxbs, evmaxhs, evmaxws, evmaxds evmaxbu, evmaxhu, evmaxwu, evmaxdu	maximum of elements in rA signed; b, h, w, d maximum of elements in rA unsigned; b, h, w, d	—
	evmaxbpsh, evmaxbpuh	pairwise maximum of bytes in rA extended to half word, signed/unsigned	—
	evmaxhpsw, evmaxhpuw	pairwise maximum of half words in rA extended to word, signed/unsigned	—
	evmaxwpsd, evmaxwpud	pairwise maximum of words in rA extended to double word, signed/unsigned	—
Maximum Magnitude	evmaxmagws	pairwise maximum of magnitude values of signed words in rA	—
Minimum	evminbs, evminhs, evminws, evminds evminbu, evminhu, evminwu, evmindu	minimum of elements in rA signed; b, h, w, d minimum of elements in rA unsigned; b, h, w, d	—
	evminbpsh, evminbpuh	pairwise minimum of bytes in rA extended to half word, signed/unsigned	—
	evminhpsw, evminhpuw	pairwise minimum of half words in rA extended to word, signed/unsigned	—
	evminwpsd, evminwpud	pairwise minimum of words in rA extended to double word, signed/unsigned	—
Negate	evnegb, evnegh, evneg, evnegd	negate signed elements in rA; b,h,w,d	—
	evnegbs, evneghs, evnegs, evnegds	negate signed elements in rA with saturation; b,h,w,d	—
	evnegbo, evnegho, evnegwo	negate signed odd elements in rA; b,h,w	—
	evnegbos, evneghos, evnegwos	negate signed odd elements in rA with saturation; b,h,w	—

Table 6-3. Simple Vector Arithmetic Instructions

Basic Operation	Variants	Description	ACC?
Round	evrndhb, evrndhbss, evrndhbus	The four half word elements of rA are rounded into 8-bits and placed into the even bytes of rD with optional signed or unsigned saturation	—
	evrndhnb, evrndhnbss, evrndhnbus	The four half word elements of rA are rounded into 8-bits using round to nearest even rounding and placed into the even bytes of rD with optional signed or unsigned saturation	—
	evrndwh, evrndwhss, evrndwhus	The two word elements of rA are rounded into 16-bits and placed into the even half words of rD with optional signed or unsigned saturation	—
	evrndwnh, evrndwnhss, evrndwnhus	The two word elements of rA are rounded into 16-bits using round to nearest even rounding and placed into the even half words of rD with optional signed or unsigned saturation	—
	evrnddw, evrnddwss, evrnddwus	The double word element of rA is rounded into 32-bits and placed into the high word of rD with optional signed or unsigned saturation. The low word is cleared.	—
	evrndndw, evrndndwss, evrndndwus	The double word element of rA is rounded into 32-bits using round to nearest even rounding and placed into the high word of rD with optional signed or unsigned saturation. The low word is cleared.	—
Sum of Absolute Differences	evsad2sh, evsad2sha, evsad2shaaw	Sums of pairs of absolute differences of 2 signed half words, optionally loading the Accumulator, or accumulating with the Accumulator values	opt.
	evsad2uh, evsad2uha, evsad2uhaaw	Sums of pairs of absolute differences of 2 unsigned half words, optionally loading the Accumulator, or accumulating with the Accumulator values	opt.
	evsad4sb, evsad4sba, evsad4sbaaw	Sums of four absolute differences of 2 signed bytes, optionally loading the Accumulator, or accumulating with the Accumulator values	opt.
	evsad4ub, evsad4uba, evsad4ubaaw	Sums of four absolute differences of 2 unsigned bytes, optionally loading the Accumulator, or accumulating with the Accumulator values	opt.
	evsadsw, evsadswa, evsadswaa	Sum of pair of absolute differences of 2 signed words, optionally loading the Accumulator, or accumulating with the Accumulator value	opt.
	evsaduw, evsaduwa, evsaduwaa	Sum of pair of absolute differences of 2 unsigned words, optionally loading the Accumulator, or accumulating with the Accumulator value	opt.

Table 6-3. Simple Vector Arithmetic Instructions

Basic Operation	Variants	Description	ACC?
Saturate	evsatsbub	Saturate signed byte to unsigned byte range	
	evsatubsb	Saturate unsigned byte to signed byte range	—
	evsatsdsw, evsatsduw	Saturate signed double word to signed or unsigned word range	—
	evsatuduw	Saturate unsigned double word to unsigned word range	—
	evsatshsb, evsatshub	Saturate signed half word to signed or unsigned byte range	—
	evsatshuh	Saturate signed half word to unsigned half word range	—
	evsatuhub	Saturate unsigned half word to unsigned byte range	—
	evsatuhsh	Saturate unsigned half word to signed half word range	—
	evsatswgsdf	Saturate signed word guarded (17.47) to signed double word fractional (1.63) range	—
	evsatswsh, evsatswuh	Saturate signed word to signed or unsigned half word range	—
	evsatswuw	Saturate signed word to unsigned word range	—
	evsatuwuh	Saturate unsigned word to unsigned half word range	—
	evsatuwsw	Saturate unsigned word to signed word range	—
Subf	evsubfb, evsubfh, evsubfw, evsubfd	subtract byte, half word, word, double word elements	—
	evsubfbss, evsubfhss, evsubfwss, evsubfdss evsubfbus, evsubfhus, evsubfwus, evsubfdus	subtract byte, half word, word, double word elements with signed or unsigned saturation	—
	evsubfhx, evsubfhxss, evsubfhxus	subtract exchanged half word elements with optional signed or unsigned saturation. The even and odd half word elements of operand rA are pairwise exchanged before subtracting	—
	evsubfwx, evsubfwxss, evsubfwxus	subtract exchanged word elements with optional signed or unsigned saturation. The high and low word elements of operand rA are exchanged before subtracting	—
	evsubifb, evsubifh, evsubifw	subtract unsigned imm value UIMM from all elements	—
	evsubfsmiaaw, evsubfssiaaw, evsubfumiaaw, evsubfusiaaw	subtract word elements in rA from Accumulator using signed/unsigned modulo/saturation operations, results into rD and Accumulator	Y
	evsubfsmiaa, evsubfssiaa, evsubfusiaa	subtract 64-bit value in rA from Accumulator with optional signed/unsigned saturation, result into rD and Accumulator	Y

Table 6-3. Simple Vector Arithmetic Instructions

Basic Operation	Variants	Description	ACC?
SubfAdd	evsubf2add2h, evsubf2add2hss	subtract for upper 2 half word elements, add for lower 2 elements, with optional signed saturation.	—
	evsubfaddh, evsubfaddhss	subtract for even half word elements, add for odd elements, with optional signed saturation.	—
	evsubfaddhx, evsubfaddhxss	The even and odd half word elements of operand rA are pairwise exchanged and then the resulting even elements are subtracted and the odd elements are added from/to elements in rB, with optional signed saturation.	—
	evsubfaddw, evsubfaddwss	The low word element of rA is added and the high word element of rA is subtracted to/from the corresponding element of rB, with optional signed saturation.	—
	evsubfaddwx, evsubfaddwxss	The word elements of rA are exchanged and then the resulting high word element is subtracted and low word element is added from/to word elements of rB, with optional signed saturation.	—

Table 6-3. Simple Vector Arithmetic Instructions

Basic Operation	Variants	Description	ACC?
Summation/ Diff	evsumws, evsumwu, evsumwsa, evsumwua	The signed or unsigned word elements of rA are summed together into 64 bits and placed into rD and optionally into the Accumulator	opt
	evsumwsaa, evsumwuaa	The signed or unsigned word elements of rA are summed together along with the contents of the Accumulator and placed into rD and the Accumulator	Y
	evsum2hs, evsum2hu, evsum2hsa, evsum2hua	Signed or unsigned pairs of half word elements of rA are summed together into words and placed into rD and optionally into the Accumulator	opt
	evsum2hsaaw, evsum2huaaw	Signed or unsigned pairs of half word elements of rA are summed together along with the contents of the corresponding word element of the accumulator, into words, and placed into rD and the Accumulator	Y
	evsum4bs, evsum4bu, evsum4bsa, evsum4bua	Signed or unsigned quads of byte elements of rA are summed together into words and placed into rD and optionally into the Accumulator	opt
	evsum4bsaaw, evsum4buaaw	Signed or unsigned quads of byte elements of rA are summed together along with the contents of the corresponding word element of the accumulator, into words, and placed into rD and the Accumulator	Y
	evsum2his, evsum2hisa	Signed pairs of interleaved half word elements of rA are summed together into words and placed into rD and optionally into the Accumulator	opt
	evsum2hisaaw	Signed pairs of interleaved half word elements of rA are summed together along with the contents of the corresponding word element of the accumulator, into words, and placed into rD and the Accumulator	Y
	evdiff2his, evdiff2hisa	Signed pairs of interleaved half word elements of rA are subtracted to produce a pair of word differences and placed into rD and optionally into the Accumulator	opt
	evdiff2hisaaw	Signed pairs of interleaved half word elements of rA are subtracted to produce a pair of word differences and the differences are added together with the contents of the corresponding word element of the accumulator, into words, and placed into rD and the Accumulator	Y

6.2.8.2 Vector Logical Instructions

Vector logical instructions are outlined in Table 6-4.

Table 6-4. Simple Vector Logical Instructions

Basic Operation	Variants	Description
AND	evand	AND word elements of rA and rB
ANDC	evandc	AND word elements of rA with complemented elements of rB
Clear	evclrbe, evclrbo	Clear (zero) even bytes of source value in rA using immediate mask (mask). Clear (zero) odd bytes of source value in rA using immediate mask (mask).
	evclrh	Clear (zero) half word elements of source value in rA using immediate mask (mask).
NAND	evnand	NAND word elements of rA and rB
NOR	evnor	NOR word elements of rA and rB
OR	evor	OR word elements of rA and rB
ORC	evorc	OR word elements of rA with complemented elements of rB
XNOR	eveqv	XNOR word elements of rA and rB
XOR	evxor	XOR word elements of rA and rB

6.2.8.3 Vector Shift/Rotate Instructions

Vector shift and rotate instructions are outlined in Table 6-5.

Table 6-5. Simple Vector Shift/Rotate Instructions

Basic Operation	Variants	Description
Shift Left	evslb, evslh, evslw, evsl evslbi, evslhi, evslwi, evsli	Logical shift left of the 8,16, 32 or 64-bit element(s) in rA by the amount(s) in rB or by the immediate value UIMM
	evsloi	Logical shift left of the value in rA by 0 to 7 bytes
Logical Shift Right	evsrbu, evsrhu, evsrwu, evsru evsrbiu, evsrhiu, evsrwiu, evsriu	Logical shift right of the 8,16, 32 or 64-bit element(s) in rA by the amount(s) in rB or by the immediate value UIMM
	evsroiu	Logical shift right of the value in rA by 0 to 7 bytes
Arithmetic Shift Right	evsrbs, evsrhs, evsrws, evsrs evsrbis, evsrhis, evsrwis, evsris	Arithmetic shift right of the 8,16, 32 or 64-bit element(s) in rA by the amount(s) in rB or by the immediate value UIMM
	evsrois	Arithmetic shift right of the value in rA by 0 to 7 bytes
Rotate Left	evrlb, evrlh, evrlw evrlbi, evrlhi, evrlwi	Rotate left of the 8,16, or 32-bit elements in rA by the amount(s) in rB or by the immediate value UIMM

6.2.8.4 Vector Compare and Vector Set Instructions

Vector compare and set instructions are outlined in Table 6-6 and Table 6-7. The compare operations update the condition register with the results of the comparison.

Table 6-6. Vector Compare Instructions

Basic Comparison Operation	Variants	Description
=	evcmpeq, evcmpeqd	Compare word or double word elements for equal
>	evcmpgts, evcmpgtu, evcmpgtds, evcmpgtdu	Compare word or double word elements for greater than signed/unsigned
<	evcmplts, evcmpltu, evcmpltds, evcmpltdu	Compare word or double word elements for less than signed/unsigned

Table 6-7. Vector Set Instructions

Basic Comparison Operation	Variants	Description
=	evseteqb[.], evseteqh[.], evseteqw[.]	Compare byte, half word or word elements in rA and rB for equal. For each byte, half word or word, set destination byte half word or word to all ‘1’s if condition met. Optionally set CR0 with comparison results.
>	evsetgtbs[.], evsetgtbu[.], evsetgths[.], evsetgthu[.], evsetgtws[.], evsetgtwu[.]	Compare byte, half word or word elements in rA and rB for greater than signed or unsigned. For each byte, half word or word, set destination byte half word or word to all ‘1’s if condition met. Optionally set CR0 with comparison results.
<	evsetltbs[.], evsetltbu[.], evsetlths[.], evsetlthu[.], evsetltws[.], evsetltwu[.]	Compare byte, half word or word elements in rA and rB for greater than signed or unsigned. For each byte, half word or word, set destination byte half word or word to all ‘1’s if condition met. Optionally set CR0 with comparison results.

6.2.8.5 Vector Select Instructions

Vector select instructions are outlined in Table 6-8.

Table 6-8. Vector Select Instructions

Operation	Variants	Description
Select	evsel	Select word elements from rA or rB based on crS condition register field
Select Bits	evselbit	Select bit elements from rA or rB based on select bit vector in rD, place results into rD
	evselbitm0	Insert bit elements from rB into rD based on select bit mask in rA of 0, place results into rD
	evselbitm1	Insert bit elements from rB into rD based on select bit mask in rA of 1, place results into rD

6.2.8.6 Vector Data Arrangement Instructions

Vector data arrangement instructions are outlined in Table 6-9. These instructions are used to rearrange fields of elements from one or more source vector registers.

Table 6-9. Vector Data Arrangement Instructions

Basic Operation	Variants	Description
De-interleave	evdlveb	de-interleave even bytes; the vector of even byte elements in rA and even byte elements in rB are concatenated and placed into rD
	evdlveob	de-interleave even/odd bytes; the vector of even byte elements in rA and odd byte elements in rB are concatenated and placed into rD
	evdlvob	de-interleave odd bytes; the vector of odd byte elements in rA and odd byte elements in rB are concatenated and placed into rD
	evdlvoeb	de-interleave odd/even bytes; the vector of odd byte elements in rA and even byte elements in rB are concatenated and placed into rD
	evdlveh	de-interleave even half words; the even half word elements in rA and even half word elements in rB are concatenated and placed into rD
	evdlveoh	de-interleave even/odd half words; the even half word elements in rA and odd half word elements in rB are concatenated and placed into rD
	evdlvoh	de-interleave odd half word; the odd half word elements in rA and odd half word elements in rB are concatenated and placed into rD
	evdlvoeh	de-interleave odd/even half words; the odd half word elements in rA and even half word elements in rB are concatenated and placed into rD

Table 6-9. Vector Data Arrangement Instructions (continued)

Basic Operation	Variants	Description
Interleave	evilveh	interleave even half words; the even half words from rA are placed into the even half words of rD and the even half words of rB are placed into the odd half words of rD
	evilveoh	interleave even/odd half words; the even half words from rA are placed into the even half words of rD and the odd half words of rB are placed into the odd half words of rD
	evilvhih	interleave high half words; the high half words from rA are placed into the even half words of rD and the high half words of rB are placed into the odd half words of rD
	evilvhiloh	interleave high/low half words; the high half words from rA are placed into the even half words of rD and the low half words of rB are placed into the odd half words of rD
	evilvloh	interleave low half words; the low half words from rA are placed into the even half words of rD and the low half words of rB are placed into the odd half words of rD
	evilvlohih	interleave low/high half words; the low half words from rA are placed into the even half words of rD and the high half words of rB are placed into the odd half words of rD
	evilvoeh	interleave odd/even half words; the odd half words from rA are placed into the even half words of rD and the even half words of rB are placed into the odd half words of rD
	evilvoh	interleave odd half words; the odd half words from rA are placed into the even half words of rD and the odd half words of rB are placed into the odd half words of rD
Merge	evmergehi	merge high words; the high word from rA is placed into the high word of rD and the high word of rB is placed into the low word of rD
	evmergehilo	merge high/low words; the high word from rA is placed into the high word of rD and the low word of rB is placed into the low word of rD
	evmergelo	merge low words; the low word from rA is placed into the high word of rD and the low word of rB is placed into the low word of rD
	evmergelohi	merge low/high words; the low word from rA is placed into the high word of rD and the high word of rB is placed into the low word of rD
Permute	evperm	Permute the byte elements in rB according to the permute vector in rA and place the results in rD
	evperm2	Permute the vector of concatenated byte elements from rA and rB according to the permute vector in rD and place the results in rD
	evperm3	Permute the vector of concatenated byte elements from rD and rB according to the permute vector in rA and place the results in rD

Table 6-9. Vector Data Arrangement Instructions (continued)

Basic Operation	Variants	Description
Pack	evpksdsws, evpkuduws	Pack the signed or unsigned double word elements from rA and rB into a pair of signed or unsigned word elements in rD, saturating if necessary
	evpksdswfrs	Pack the signed double word fractional elements from rA and rB into a pair of signed word elements in rD using the current rounding mode in SPEFSCR, saturating if necessary
	evpksdshefrs	Pack the signed 33.31 guarded fractional elements from rA and rB into a pair of signed half word even elements in rD using the current rounding mode in SPEFSCR, saturating if necessary
	evpkshsbs, evpkshubs, evpkuhubs	Pack the 8 signed or unsigned half word elements from rA and rB into 8 signed or unsigned byte elements in rD, saturating if necessary
	evpkswgshefrs	Pack the signed 17.47 guarded fractional elements from rA and rB into a pair of signed half word even elements in rD using the current rounding mode in SPEFSCR, saturating if necessary
	evpkswgswfrs	Pack the signed 17.47 guarded fractional elements from rA and rB into a pair of signed word elements in rD using the current rounding mode in SPEFSCR, saturating if necessary
	evpkswshs, evpkswuhs, evpkuwuhs	Pack the 4 signed or unsigned word elements from rA and rB into 4 signed or unsigned half word elements in rD, saturating if necessary
	evpkswshilvs	Pack the 4 signed word elements from rA and rB into 4 signed half word elements in rD with interleaving, saturating if necessary
	evpkswshfrs	Pack the 4 signed fractional word elements from rA and rB into 4 signed or fractional half word elements in rD using the current rounding mode in SPEFSCR, saturating if necessary
	evpkswshilvfrs	Pack the 4 signed fractional word elements from rA and rB into 4 signed or fractional half word elements in rD with interleaving using the current rounding mode in SPEFSCR, saturating if necessary

Table 6-9. Vector Data Arrangement Instructions (continued)

Basic Operation	Variants	Description
Splat	evsplatb	splat (replicate) the byte from rA selected by the immediate field into all byte elements of rD
	evsplath	splat (replicate) the half word from rA selected by the immediate field into all half word elements of rD
	evsplatfib, splatfih, splatfi	Splat the 5-bit SIMM field as a signed fraction into all byte, half word, or word elements of rD
	evsplatfiba, splatfiha, splatfia	Splat the 5-bit SIMM field as a signed fraction into all byte, half word, or word elements of rD and the accumulator
	evsplatfid	Splat the 5-bit SIMM field as a signed fraction into rD
	evsplatfida	Splat the 5-bit SIMM field as a signed fraction into rD and the accumulator
	evsplatfibo, splatfiho, splatfio	Splat the 5-bit SIMM field as a signed fraction into the odd byte, half word, or word elements of rD
	evsplatfiboa, splatfihoa, splatfioa	Splat the 5-bit SIMM field as a signed fraction into the odd byte, half word, or word elements of rD and the accumulator
	evsplatib, evsplatih, evsplati	Splat the 5-bit SIMM field as a signed integer into all byte, half word, or word elements of rD
	evsplatiba, evsplatiha, evsplatia	Splat the 5-bit SIMM field as a signed integer into all byte, half word, or word elements of rD and the accumulator
	evsplatid	Splat the 5-bit SIMM field as a signed integer into rD
	evsplatida	Splat the 5-bit SIMM field as a signed integer into rD and the accumulator
	evsplatibe, evsplatihe, evsplatie	Splat the 5-bit SIMM field as a signed integer into the even byte, half word, or word elements of rD
	evsplatibea, evsplatihea, evsplatiea	Splat the 5-bit SIMM field as a signed integer into the even byte, half word, or word elements of rD and the accumulator

Table 6-9. Vector Data Arrangement Instructions (continued)

Basic Operation	Variants	Description
Swap	evswapbhilo	bytes within the upper 2 byte pairs in rA are swapped, and concatenated with swapped bytes in the lower 2 byte pairs of rB.
	evswapblohi	bytes within the lower 2 byte pairs in rA are swapped, and concatenated with swapped bytes in the upper 2 byte pairs of rB.
	evswaphe	The even half words in rA are swapped, and merged with the odd half words of rB.
	evswaphhi	The upper 2 half words in rA are swapped, and concatenated with the lower 2 half words of rB.
	evswaphhilo	The upper 2 half words in rA are swapped, and concatenated with swapped lower 2 half words of rB.
	evswaphlo	The lower 2 half words in rA are swapped, and concatenated after the upper 2 half words of rB.
	evswaphlohi	The lower 2 half words in rA are swapped, and concatenated with swapped upper 2 half words of rB.
	evswapho	The odd half words in rA are swapped, and then merged with even half words of rB.
Unpack	evunpkhibsi, evunpkhibui, evunpklobsi, evunpklobui	Unpack the high or low 4 bytes of rA into signed or unsigned integer half words
	evunpkhihf, evunpkhihsi, evunkpkhihui, evunpklohf, evunpklohsi, evunkpklohui	Unpack the high or low 2 half words of rA into signed fractional, signed integer, or unsigned integer words
	evunpkhiwgsf, evunpklowgsf	Unpack the high or low word of rA into guarded signed fractional (17.47) format
Extract	evxtrb	A specified byte in rA is placed into a specified byte of rD, zeroing all other bytes of rD
	evxtrd	a double word is extracted from the concatenated byte elements of rA and rB and placed into rD
	evxtrh	A specified half word in rA is placed into a specified half word of rD, zeroing all other half words of rD
Insert	evinsb	A specified byte in rA is placed into a specified byte of rD; all other bytes of rD are unchanged.
	evinsh	A specified half word in rA is placed into a specified half word of rD; all other half words of rD are unchanged.

6.2.8.7 Multiply and accumulate instructions

These instructions perform multiply operations, optionally add the result to the accumulator and place the result into the destination register and optionally into the accumulator. These instructions are composed of different multiply forms, data formats and data accumulate options. The mnemonics for these instructions indicate their various characteristics. These are shown in Table 6-10.

Table 6-10. 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 accumulate result
ho	half word odd	16 × 16 → 32
hog	half word odd guarded	16 × 16 → 32, 64-bit final accumulate result
w	word	32 × 32 → 64
wehg	word even high guarded	32 × 32 → 64 in 17.47 format
wh	word high	32 × 32 → 32 (high order 32 bits of product)
wl	word low	32 × 32 → 32 (low order 32 bits of product)
wohg	word odd high guarded	32 × 32 → 64 in 17.47 format
Data Format		
smf	signed modulo fractional	modulo, no saturation or overflow
smfr	signed modulo fractional round	modulo, no saturation or overflow, rounding based on current rounding mode
smi	signed modulo integer	modulo, no saturation or overflow
ssf	signed saturate fractional	saturation on product and accumulate
ssfr	signed saturate fractional round	saturation on product and accumulate, rounding based on current rounding mode
ssi	signed saturate integer	saturation on accumulate
umi	unsigned modulo integer	modulo, no saturation or overflow
usi	unsigned saturate integer	saturation on accumulate
Accumulate Option		
a	place in Accumulator	result → rD, Accumulator
aa	add to Accumulator	Accumulator + result → rD, Accumulator
aaw	add to Accumulator as word elements	Accumulator[0:31] + result[0:31] → rD[0:31], Accumulator[0:31] Accumulator[32:63] + result[32:63] → rD[32:63], Accumulator[32:63]

Table 6-10. Mnemonic Extensions for Multiply Accumulate Instructions (continued)

Extension	Meaning	Comments
aaw3	add to rD as word elements	rD[0:31] + result[0:31] → rD[0:31], Accumulator[0:31] rD[32:63] + result[32:63] → rD[32:63], Accumulator[32:63]
an	add negated to Accumulator	Accumulator – result → rD, Accumulator
anw	add negated to Accumulator as word elements	Accumulator[0:31] – result[0:31] → rD[0:31], Accumulator[0:31] Accumulator[32:63] – result[32:63] → rD[32:63], Accumulator[32:63]
anw3	add negated to rD as word elements	rD[0:31] – result[0:31] → rD[0:31], Accumulator[0:31] rD[32:63] – result[32:63] → rD[32:63], Accumulator[32:63]

6.2.8.8 Dot product instructions

These instructions perform multiple multiply operations, optionally add the results to the accumulator, and place the result into the destination register and optionally into the accumulator. These instructions are composed of different forms, data formats and data accumulate options. The mnemonics for these instructions indicate their various characteristics. These are shown in Table 6-11.

Table 6-11. Mnemonic Extensions for Dot Product Instructions

Extension	Meaning	Comments
	Multiply Form	
b	byte	8 × 8 + 8 × 8 + 8 × 8 + 8 × 8 → 32, high and low
4h	four half words	16 × 16 + 16 × 16 + 16 × 16 + 16 × 16 → 32
4hg	four half words guarded	16 × 16 + 16 × 16 + 16 × 16 + 16 × 16 → 64
h	half word	16 × 16 op 16 × 16 → 32, high and low
hih	high half words	16 × 16 op 16 × 16 → 32, high half words, used for complex mul
loh	low half words	16 × 16 op 16 × 16 → 32, low half words, used for complex mul
4hxga	four half words exchanged guarded add	(16 × 16 + 16 × 16) + (16 × 16 + 16 × 16) → 64, even and odd rA half words changed
4hxgs	four half words exchanged guarded subtract	(16 × 16 - 16 × 16) + (16 × 16 - 16 × 16) → 64, even and odd rA half words changed
w	word	32 × 32 op 32 × 32 → 64
wg	word guarded	32 × 32 op 32 × 32 → 64 in 17.47 fractional format
wxga	word exchanged guarded add	32 × 32 + 32 × 32 → 64 in 17.47 fractional format, words in rA are changed
wxgs	word exchanged guarded subtract	32 × 32 - 32 × 32 → 64 in 17.47 fractional format, words in rA are changed

Table 6-11. Mnemonic Extensions for Dot Product Instructions (continued)

Extension	Meaning	Comments
Operation		
a	add	addition of intermediate products
s	subtract	subtraction of intermediate products
c	complex	complex format arithmetic
Data Format		
smf	add signed modulo fractional	modulo, no saturation or overflow
smi	signed modulo integer	modulo, no saturation or overflow
ssf	signed saturate fractional	saturation on product and accumulate
ssfr	signed saturate fractional round	saturation on product and accumulate, rounding based on current rounding mode
ssi	signed saturate integer	saturation on product and accumulate
umi	unsigned modulo integer	modulo, no saturation or overflow
usi	unsigned saturate integer	saturation on product and accumulate
Accumulate Option		
a	place in Accumulator	result → rD, Accumulator
aa	add to Accumulator	Accumulator + result → rD, Accumulator
aa3	add to Accumulator, 3op	rD + result → rD, Accumulator
aaw	add to Accumulator as word elements	Accumulator[0:31] + result[0:31] → rD[0:31], Accumulator[0:31] Accumulator[32:63] + result[32:63] → rD[32:63], Accumulator[32:63]
aaw3	add to Accumulator as word elements, 3 op	rD[0:31] + result[0:31] → rD[0:31], Accumulator[0:31] rD[32:63] + result[32:63] → rD[32:63], Accumulator[32:63]

6.2.8.9 Miscellaneous Vector Instructions

Miscellaneous vector instructions are outlined in Table 6-4.

Table 6-12. Misc. Vector Instructions

Operation	Variants	Description
load vector for shift	evlvsl	load vector for shift left; place a vector of constant values for a vector permute for left shift
	evlvsr	load vector for shift right; place a vector of constant values for a vector permute for right shift

Table 6-12. Misc. Vector Instructions

Operation	Variants	Description
store Accumulator	evmar	move Accumulator to register rA
load Accumulator	evmra	move register rA to Accumulator
Bit reversed increment	brinc	Compute a bit-reversed increment for a memory offset for bit-reversed addressing
Circular Increment	circinc	Computes a modulo increment for supporting circular buffer index pointer modification

6.2.9 Load and Store Instructions

SPE provides a number of load and store instructions. These instructions provide load and store capabilities for moving data elements between the GPRs and memory. Data elements of 8, 16, 32, and 64 bits are supported. A variety of forms are provided that position data for efficient computation.

6.2.9.1 Addressing Modes—Non-Update forms

Base + index and base + scaled immediate addressing modes are provided. Base registers hold 64-bit pointer values (32-bit pointers in a 32-bit implementation of the architecture), while registers used as index values provide 32-bit index values. Scaled immediate values are unsigned and are scaled by the size of the access.

6.2.9.1.1 Base + Scaled Immediate Addressing—Non-Update Form

In the base + scaled immediate addressing mode, register rA holds a 32-bit pointer value or a value of zero (if rA = 0), and an immediate field in the instruction word provides a 5-bit unsigned immediate value which is zero-extended and scaled (shifted left) by 1, 2, or 3, depending on the size (half word, word, or double word) of the access. The sum of the value in rA and the zero-extended scaled immediate form the effective address:

```
if (rA = 0) then b ← 0
else b ← $(rA_{32:63})$
SCL ← {1,2,3} // half word, word, or double word
EA ← b + EXTZ(UIMM*SCL)
```

6.2.9.1.2 Base + Index Addressing

In the Base + Index addressing mode, register rA holds a 32-bit pointer value or a value of zero (if rA = 0), while register rB provides a 32-bit index. The sum forms the effective address:

```
if (rA = 0) then b ← 0
else b ← $(rA_{32:63})$
EA ← b + (rB)
```

6.2.9.2 Addressing Modes—Update forms

The base + scaled immediate addressing mode is also provided with an update form. As in the non-update form, base register rA holds 32-bit pointer values. For the update form of the base + scaled immediate addressing mode, the same effective address calculation is used as defined in Section 6.2.9.1.1, “Base + Scaled Immediate Addressing—Non-Update Form,” and the calculated effective address is placed into rA by the instruction.

For the base + scaled immediate with update addressing mode, scaled immediate values of 0 are reserved for future definition and are treated as illegal. Instruction encodings with rA = 0 are also reserved for future definition and treated as illegal instructions.

6.2.9.3 Addressing Modes—Modify forms

The base + index addressing mode is also provided with a set of modify forms. In the modify forms, register rB holds 32-bit pointer values, while register rA is used to provide an index value as well as to provide specialized control information for performing a post-modification to the lower 32 bits of rA.

Modify forms are provided to allow for parallel address computations to occur, which are useful for sequential accessing of arrays, lists, circular buffers, and other complex data structures. Modify forms of load and store instructions cause a calculated update value to be placed in the lower portion of register rA. Support for specialized addressing modes are available when using base + index modify forms.

For the base + index modify forms, the modify calculation mode selection is based on a **mode** field in register rA (rA[0:3]). Modify forms modify the original value in rA based on an addressing calculation performed in parallel with the load or store instruction, which may or may not be the value of the effective address of the load or store instruction, depending on the actual calculation mode. This is in contrast to normal update forms of the Power Arch load and store instructions since the new value placed into rA need not correspond to the effective address of the load or store.

The following three modify calculation modes are currently defined and selected by the value in rA[0:3]:

- Linear addressing: mode = 0000
- Circular addressing: mode = 1000
- Bit-reversed addressing: mode = 1010

All other mode encodings are reserved, and either result in an unimplemented instruction exception, or a boundedly undefined result depending on the implementation.

Instruction encodings with rA = 0 are reserved for future definition and are treated as illegal instructions.

6.2.9.3.1 Linear Addressing Update Mode

Linear addressing update calculation mode causes the sum of rA[32:63] and rB[32:63] to be placed into rA[32:63]:

```
if(mode=0000) then
    $rA_{32:63} \leftarrow rA_{32:63} + rB_{32:63}$
```

6.2.9.3.2 Circular Addressing Modify Mode

Circular addressing modify mode is provided to support addressing of circular buffers. Circular addressing mode causes a circular increment to be performed on a portion of rA[32:63] (the circular buffer index portion of rA) after the EA calculation, using the offset and length specifiers in rA and the result is placed into rA[32–63]. rA[0–31] is left unchanged. rA[32–63] must be $\geq_{si}$ 0 and $\leq_{ui}$ Length, and the magnitude of Offset must be ≤ Length + 1, or the resulting value is boundedly undefined. rB must point to a double-word boundary in memory, and Length + 1 must be a multiple of eight bytes or an alignment error will be generated.

Figure 6-4 shows how rA is used in forming the update value for mode 1000 (circinc).

0....3	4 5	6....13	14 15	16 31	32 ... 63	
Mode (1000)	-	**Offset** (signed)	-	Length (unsigned)	**Index** (must always be positive and <=$_{ui}$ **Length**)	rA

```
$Offset_{0:7}$ ← $rA_{8:15}$; // signed byte offset, must be <= Length+1
$Length_{0:15}$ ← $rA_{16:31}$; // unsigned buffer length-1 in bytes. Length is byte index of
                   // last byte in buffer.
                   // buffer must be aligned on a doubleword boundary, and be a
                   // multiple of 8 bytes, i.e. $Length_{13:15}$ =3'b111.
$Index_{0:31}$ ← $rA_{32:63}$; // index into buffer, must be <=$_{ui}$ $Length_{0:15}$, (so always >=$_{si}$ 0).

if (($Offset_0$ = 0) & (($EXTS_{32}$($Offset_{0:7}$) + $Index_{0:31}$) >$_{ui}$ $EXTZ_{32}$($Length_{0:15}$))) then
    $rA_{32:63}$ ← $Index_{0:31}$ + $EXTS_{32}$($Offset_{0:7}$) - $EXTZ_{32}$($Length_{0:15}$) - 1; // wrap at end

elseif ($Offset_0$ = 1) & (($EXTS_{32}$($Offset_{0:7}$) + $Index_{0:31}$) <$_{si}$ 0)) then
    $rA_{32:63}$ ← $Index_{0:31}$ + $EXTS_{32}$($Offset_{0:7}$) + $EXTZ_{32}$($Length_{0:15}$) + 1; // wrap at start

else $rA_{32:63}$ ← $Index_{0:31}$ + $EXTS_{32}$($Offset_{0:7}$);
```

Figure 6-4. rA Used to Form Update Value for Mode 1000

Note that **misalignment** may cause the operand fetched to span the virtual boundary between the last byte of the buffer at byte Buffer[Length] and the first byte of the Buffer at byte Buffer[0].

6.2.9.3.3 Bit-Reversed Addressing Modify Mode

Bit-reversed addressing modify calculation mode is provided to support addressing of buffers and arrays used in FFT calculations.

When using bit-reversed addressing modify mode, a bit-reversed increment is performed on rA[32:63] after the EA calculation, using a mask specifier in rA. The mask specifier is also used to indicate the bits of rA[32:63] which are updated.

Figure 6-5 shows how rA is used in forming the update value for bit-reversed addressing update mode. Note that the computation is similar to the **brinc** instruction computation, but the mask is applied to updating only those bits of rA indicated by a ‘1’ in the mask value, unlike in the **brinc** instruction, in which all low order bits of rD corresponding to the maximum mask size are updated.

0 3	4 15	16 31	32 ... 63	
Mode (1010)	0....0	Mask	**Index**	rA

```
Mask ← $rA_{16:31}$
// mask value is $^{log2(\#points)}1$, zero extended, then left shifted $log_2$(element size
// in bytes). e.g., a 16 point FFT on half words has a mask of 16‘b0000000000011110


a ← $rA_{48:63}$ // up to 64KB in a single FFT
d ← bitreverse(1 + bitreverse(a | ~Mask))

$rA_{32:63}$ ← $rA_{32:47}$ || ((r$A_{48:63}$ & ~Mask) |(d & Mask)) // different than brinc. allows main
pointer sharing to multiple buffers less than 64KB in size.
```

Figure 6-5. rA Used to Form Update Value for Bit-Reversed Addressing Update Mode

6.2.9.4 Vector Load and Store Instruction Summary

Vector load and store instructions are provided to load and store various size vectors of byte, half-word, word or double-word size. These instructions allow for endian-neutral code to be written. In addition, update forms of the non-indexed instructions are provided to allow for base register updates. Variations of the load instructions provide splat (replication) capability for placing a smaller vector element into multiple element positions in a vector register.

Vector load and store instructions are outlined in Table 6-13.

Table 6-13. Vector Load and Store Instructions

Operation	Variants	Description
Load Byte	evlbbsplatb, evlbbsplatbu, evlbbsplatbx, evlbbsplatbmx	load byte and splat byte into 8 byte element positions
Load Double Word	evldb, evldbu, evldbx, evldbmx	load double word as byte elements
	evldd, evlddu, evlddx, evlddmx	load double word as double word
	evldh, evldhu, evldhx, evldhmx	load double word as half word elements
	evldw, evldwu, evldwx, evldwmx	load double word as word elements
Load Half Word	evlhhesplat, evlhhesplatu, evlhhesplatx, evlhhesplatmx	load half word into even half word elements, zeroing the odd half word s elements
	evlhhossplat, evlhhossplatu, evlhhossplatx, evlhhossplatmx	load half word into odd half word elements, sign-extending to word elements
	evlhhousplat, evlhhousplatu, evlhhousplatx, evlhhousplatmx	load half word into odd half word elements, zero-extending to word elements
	evlhhsplath, evlhhsplathu, evlhhsplathx, evlhhsplathmx	load half word into all half word elements

Table 6-13. Vector Load and Store Instructions (continued)

Operation	Variants	Description
Load Word	evlwbe, evlwbeu, evlwbex, evlwbemx	load word as four byte elements into the four even byte elements, zeroing the odd byte elements
	evlwbos, evlwbosu, evlwbosx, evlwbosmx	load word as four byte elements into the four odd byte elements, sign-extending to half word elements
	evlwbou, evlwbouu, evlwboux, evlwboumx	load word as four byte elements into the four odd byte elements, zero-extending to half word elements
	evlwbsplatw, evlwbsplatwu, evlwbsplatwx, evlwbsplatwmx	load word as four byte elements into both word elements
	evlwhe, evlwheu, evlwhex, evlwhemx	load word as two half word elements into the two even half word elements, zeroing the odd half word elements
	evlwhos, evlwhosu, evlwhosx, evlwhosmx	load word as two half word elements into the two odd half word elements, sign-extending to word elements
	evlwhou, evlwhouu, evlwhoux, evlwhoumx	load word into the two odd half word elements, zero-extending to word elements
	evlwhsplat, evlwhsplatu, evlwhsplatx, evlwhsplatmx	load word as two half word elements, placing the first half word into both upper half word elements, second half word into both lower half word elements
	evlwhsplatw, evlwhsplatwu, evlwhsplatwx, evlwhsplatwmx	load word as two half word elements, into both word elements
	evlwwsplat, evlwhsplatu, evlwhsplatx, evlwhsplatmx	load word as word element, into both word elements
Store Double Word	evstdb, evstdbu, evstdbx, evstdbmx	store double word as byte elements
	evstdd, evstddu, evstddx, evstddmx	store double word as double word
	evstdh, evstdhu, evstdhx, evstdhmx	store double word as half word elements
	evstdw, evstdwu, evstdwx, evstdwmx	store double word as word elements
Store Half Word	evsthb, evsthbu, evsthbx, evsthbmx	store half word as byte elements
Store Word	evstwb, evstwbu, evstwbx, evstwbmx	store word as four byte elements
	evstwbe, evstwbeu, evstwbex, evstwbemx	store word from four even byte elements
	evstwbo, evstwbou, evstwbox, evstwbomx	store word from four odd byte elements
	evstwhe, evstwheu, evstwhex, evstwhemx	store word from two even half word elements
	evstwho, evstwhou, evstwhox, evstwhomx	store word from two odd half word elements
	evstwwe, evstwweu, evstwwex, evstwwemx	store word from even word element
	evstwwo, evstwwou, evstwwox, evstwwomx	store word from odd word element

6.2.10 SPE Exceptions

The architecture defines the following SPE exceptions:

- SPE unavailable exception
- SPE vector alignment exception

Interrupt vector offset registers (IVOR) IVOR32 (SPE/embedded floating point unavailable interrupt) and IVOR5 (alignment interrupt), are used by the interrupt model. The SPR number for IVOR32 is 528, IVOR5 is defined by Power ISA. These registers are privileged.

6.2.10.1 SPE/Embedded Floating-point Unavailable Exception

The SPE/embedded floating-point unavailable exception is taken if MSR[SPE] is cleared and execution of a SPE instruction other than the **brinc** instruction is attempted. When the SPE/embedded floating-point unavailable exception occurs, the processor suppresses execution of the instruction causing the exception. The SRR0, SRR1, MSR, and ESR registers are modified as follows:

- SRR0 is set to the effective address of the instruction causing the exception.
- SRR1 is set to the contents of the MSR at the time of the exception.
- MSR[CE, ME, DE] are unchanged. All other bits are cleared.
- ESR[SPE] is set. All other ESR bits are cleared.

Instruction execution resumes at address IVPR[0–15]||IVOR32[16–27]||0b0000.

6.2.10.2 SPE Vector Alignment Exception

For e200z760n3, the SPE vector alignment exception is taken if the effective address of any of the following instructions is not aligned to a 32-bit boundary: **evldd[u]**, **evlddx**, **evldw[u]**, **evldwx**, **evldh[u]**, **evldhx**, **evstdd[u]**, **evstddx**, **evstdw[u]**, **evstdwx**, **evstdh[u]**, and **evstdhx**. When an SPE vector alignment exception occurs, the processor suppresses the execution of the instruction causing the alignment exception and takes an alignment interrupt.

SRR0, SRR1, MSR, ESR, and DEAR are modified as follows:

- SRR0 is set to the effective address of the instruction causing the alignment exception.
- SRR1 is set to the contents of the MSR at the time of the exception.
- MSR[CE, ME, DE] are unchanged. All other bits are cleared.
- ESR[SPE] (bit 24) is set. ESR[ST] is set only if the instruction causing the exception is a store and is cleared for a load. All other bits are cleared.
- DEAR is updated with the effective address of a byte of the load or store.

Instruction execution resumes at address IVPR[0–15]||IVOR5[16–27]||0b0000.

6.2.11 Exception Priorities

The following list shows the priority order in which exceptions are taken:

1. SPE Unavailable exception
2. SPE Vector Alignment exception

An SPE vector alignment exception is taken if an SPE double-word vector load or store access is attempted with an address which is not 32-bit aligned.

6.3 SPE Instruction Timing

Instruction timing in number of processor clock cycles for SPE instructions are shown in the following tables. Pipelined instructions are shown with cycles of total latency and throughput cycles. Divide instructions are not pipelined and block other instructions from executing during divide execution.

6.3.1 SPE Simple Vector Arithmetic Instructions Timing

Table 6-14 shows instruction timing for SPE integer simple instructions. The table is sorted by opcode. These instructions are issued as a pair of operations.

Table 6-14. Simple Vector Arithmetic Instruction Timing

Basic Operation	Instruction	Latency	Throughput
Absolute Value	evabsb, evabsh, evabs, evabsd	1	1
	evabsbs, evabshs, evabss, evabsds	1	1
Absolute Difference	evabsdifsb, evabsdifsh, evabsdifsw, evabsdifub, evabsdifuh, evabsdifuw	1	1
Add	evaddb, evaddh, evaddw, evaddd	1	1
	evaddbss, evaddhss, evaddwss, evadddss evaddbus, evaddhus, evaddwus, evadddus	1	1
	evaddhx, evaddhxss, evaddhxus	1	1
	evaddwx, evaddwxss, evaddwxus	1	1
	evaddib, evaddih, evaddiw	1	1
	evaddsmiaaw, evaddssiaaw, evaddumiaaw, evaddusiaaw	1	1
	evaddsmiaa, evaddssiaa, evaddusiaa	1	1
AddSubf	evadd2subf2h, evadd2subf2hss	1	1
	evaddsubfh, evaddsubfhss	1	1
	evaddsubfhx, evaddsubfhxss	1	1
	evaddsubfw, evaddsubfwss	1	1
	evaddsubfwx, evaddsubfwxss	1	1

Table 6-14. Simple Vector Arithmetic Instruction Timing (continued)

Basic Operation	Instruction	Latency	Throughput
Average	evavgbs, evavghs, evavgws, evavgds, evavgbsr, evavghsr, evavgwsr, evavgdsr evavgbu, evavghu, evavgwu, evavgdu evavgbur, evavghur, evavgwur, evavgdur	1	1
Count Leading	evcntlsh, evcntlzh evcntlsw, evcntlzw	1	1
Extend	evextsb, evextzb	1	1
	evextsbh	1	1
	evextsh, *evextzh* (use evclrh)	1	1
	evextsw	1	1
Maximum	evmaxbs, evmaxhs, evmaxws, evmaxds evmaxbu, evmaxhu, evmaxwu, evmaxdu	1	1
	evmaxbpsh, evmaxbpuh	1	1
	evmaxhpsw, evmaxhpuw	1	1
	evmaxwpsd, evmaxwpud	1	1
Maximum Magnitude	evmaxmagws	1	1
Minimum	evminbs, evminhs, evminws, evminds evminbu, evminhu, evminwu, evmindu	1	1
	evminbpsh, evminbpuh	1	1
	evminhpsw, evminhpuw	1	1
	evminwpsd, evminwpud	1	1
Negate	evnegb, evnegh, evneg, evnegd	1	1
	evnegbs, evneghs, evnegs, evnegds	1	1
	evnegbo, evnegho, evnegwo	1	1
	evnegbos, evneghos, evnegwos	1	1
Round	evrndhb, evrndhbss, evrndhbus	1	1
	evrndhnb, evrndhnbss, evrndhnbus	1	1
	evrndwh, evrndwhss, evrndwhus	1	1
	evrndwnh, evrndwnhss, evrndwnhus	1	1
	evrnddw, evrnddwss, evrnddwus	1	1
	evrndndw, evrndndwss, evrndndwus	1	1

Table 6-14. Simple Vector Arithmetic Instruction Timing (continued)

Basic Operation	Instruction	Latency	Throughput
Sum of Absolute Differences	evsad2sh, evsad2sha, evsad2shaaw	1	1
	evsad2uh, evsad2uha, evsad2uhaaw	1	1
	evsad4sb, evsad4sba, evsad4sbaaw	1	1
	evsad4ub, evsad4uba, evsad4ubaaw	1	1
	evsadsw, evsadswa, evsadswaa	1	1
	evsaduw, evsaduwa, evsaduwaa	1	1
Saturate	evsatsbub	1	1
	evsatubsb	1	1
	evsatsdsw, evsatsduw	1	1
	evsatuduw	1	1
	evsatshsb, evsatshub	1	1
	evsatshuh	1	1
	evsatuhub	1	1
	evsatuhsh	1	1
	evsatswgsdf	1	1
	evsatswsh, evsatswuh	1	1
	evsatswuw	1	1
	evsatuwuh	1	1
	evsatuwsw	1	1
Subf	evsubfb, evsubfh, evsubfw, evsubfd	1	1
	evsubfbss, evsubfhss, evsubfwss, evsubfdss evsubfbus, evsubfhus, evsubfwus, evsubfdus	1	1
	evsubfhx, evsubfhxss, evsubfhxus	1	1
	evsubfwx, evsubfwxss, evsubfwxus	1	1
	evsubifb, evsubifh, evsubifw	1	1
	evsubfsmiaaw, evsubfssiaaw, evsubfumiaaw, evsubfusiaaw	1	1
	evsubfsmiaa, evsubfssiaa, evsubfusiaa	1	1
SubfAdd	evsubf2add2h, evsubf2add2hss	1	1
	evsubfaddh, evsubfaddhss	1	1
	evsubfaddhx, evsubfaddhxss	1	1
	evsubfaddw, evsubfaddwss	1	1
	evsubfaddwx, evsubfaddwxss	1	1

Table 6-14. Simple Vector Arithmetic Instruction Timing (continued)

Basic Operation	Instruction	Latency	Throughput
Summation/ Diff	evsumws, evsumwu, evsumwsa, evsumwua	1	1
	evsumwsaa, evsumwuaa	1	1
	evsum2hs, evsum2hu, evsum2hsa, evsum2hua	1	1
	evsum2hsaaw, evsum2huaaw	1	1
	evsum4bs, evsum4bu, evsum4bsa, evsum4bua	1	1
	evsum4bsaaw, evsum4buaaw	1	1
	evsum2his, evsum2hisa	1	1
	evsum2hisaaw	1	1
	evdiff2his, evdiff2hisa	1	1
	evdiff2hisaaw	1	1

6.3.2 SPE Complex Integer Instruction Timing

Table 6-15 shows instruction timing for SPE complex integer instructions. For the divide instructions, the number of stall cycles is (latency) for following instructions.

Table 6-15. SPE Complex Integer Instruction Timing

Operation	Instruction	Latency	Throughput
Divide	evdivws, evdivwu, evdivwsf, evdivwuf evdivs, evdivu	12-32[1]	12-32[1]

[1] Timing is data dependent

6.3.3 SPE Vector Logical Instruction Timing

Table 6-16 shows instruction timing for SPE simple vector logical instructions.

Table 6-16. SPE Vector Logical Instruction Timing

Basic Operation	Instruction	Latency	Throughput
AND	evand	1	1
ANDC	evandc	1	1
Clear	evclrbe, evclrbo	1	1
	evclrh	1	1
NAND	evnand	1	1
NOR	evnor	1	1
OR	evor	1	1

Table 6-16. SPE Vector Logical Instruction Timing (continued)

Basic Operation	Instruction	Latency	Throughput
ORC	evorc	1	1
XNOR	eveqv	1	1
XOR	evxor	1	1

6.3.4 SPE Vector Shift/Rotate Instruction Timing

Instruction timing for SPE vector shift/rotate instructions is shown in Table 6-17.

Table 6-17. SPE Vector Shift/Rotate Instruction Timing

Basic Operation	Instruction	Latency	Throughput
Shift Left	evslb, evslh, evslw, evsl evslbi, evslhi, evslwi, evsli	1	1
	evsloi	1	1
Logical Shift Right	evsrbu, evsrhu, evsrwu, evsru evsrbiu, evsrhiu, evsrwiu, evsriu	1	1
	evsroiu	1	1
Arithmetic Shift Right	evsrbs, evsrhs, evsrws, evsrs evsrbis, evsrhis, evsrwis, evsris	1	1
	evsrois	1	1
Rotate Left	evrlb, evrlh, evrlw evrlbi, evrlhi, evrlwi	1	1

6.3.5 SPE Vector Compare and Vector Set Instruction Timing

Instruction timing for SPE vector compare and set instructions is shown in Table 6-18 and Table 6-19. Table 6-18 shows the SPE vector compare instruction timing.

Table 6-18. SPE Vector Compare Instruction Timing

Basic Comparison Operation	Instruction	Latency	Throughput
=	evcmpeq, evcmpeqd	1	1
>	evcmpgts, evcmpgtu, evcmpgtds, evcmpgtdu	1	1
<	evcmplts, evcmpltu, evcmpltds, evcmpltdu	1	1

Table 6-19 shows the SPE vector set instruction timing.

Table 6-19. SPE Vector Set Instruction Timing

Comparison Operation	Instruction	Latency	Throughput
=	evseteqb[.], evseteqh[.], evseteqw[.]	1	1
>	evsetgtbs[.], evsetgtbu[.], evsetgths[.], evsetgthu[.], evsetgtws[.], evsetgtwu[.]	1	1
<	evsetltbs[.], evsetltbu[.], evsetlths[.], evsetlthu[.], evsetltws[.], evsetltwu[.]	1	1

6.3.6 SPE Vector Select Instruction Timing

Table 6-20 shows instruction timing for SPE vector select instructions.

Table 6-20. SPE Vector Select Instruction Timing

Operation	Instruction	Latency	Throughput
Select	evsel	1	1
Select Bits	evselbit	1	1
	evselbitm0	1	1
	evselbitm1	1	1

6.3.7 SPE Vector Data Arrangement Instruction Timing

Table 6-21 shows the instruction timing for SPE vector data arrangement instructions.

Table 6-21. SPE Vector Data Arrangement Instruction Timing

Operation	Instruction	Latency	Throughput
De-interleave	evdlveb	1	1
	evdlveob	1	1
	evdlvob	1	1
	evdlvoeb	1	1
	evdlveh	1	1
	evdlveoh	1	1
	evdlvoh	1	1
	evdlvoeh	1	1

Table 6-21. SPE Vector Data Arrangement Instruction Timing (continued)

Operation	Instruction	Latency	Throughput
Interleave	evilveh	1	1
	evilveoh	1	1
	evilvhih	1	1
	evilvhiloh	1	1
	evilvloh	1	1
	evilvlohih	1	1
	evilvoeh	1	1
	evilvoh	1	1
Merge	evmergehi	1	1
	evmergehilo	1	1
	evmergelo	1	1
	evmergelohi	1	1
Permute	evperm	1	1
	evperm2	1	1
	evperm3	1	1
Pack	evpksdsws, evpkuduws	1	1
	evpksdswfrs	1	1
	evpksdshefrs	1	1
	evpkshsbs, evpkshubs, evpkuhubs	1	1
	evpkswgshefrs	1	1
	evpkswgswfrs	1	1
	evpkswshs, evpkswuhs, evpkuwuhs	1	1
	evpkswshilvs	1	1
	evpkswshfrs	1	1
	evpkswshilvfrs	1	1

Table 6-21. SPE Vector Data Arrangement Instruction Timing (continued)

Operation	Instruction	Latency	Throughput
Splat	evsplatb	1	1
	evsplath	1	1
	evsplatfib, splatfih, splatfi	1	1
	evsplatfiba, splatfiha, splatfia	1	1
	evsplatfid	1	1
	evsplatfida	1	1
	evsplatfibo, splatfiho, splatfio	1	1
	evsplatfiboa, splatfihoa, splatfioa	1	1
	evsplatib, evsplatih, evsplati	1	1
	evsplatiba, evsplatiha, evsplatia	1	1
	evsplatid	1	1
	evsplatida	1	1
	evsplatibe, evsplatihe, evsplatie	1	1
	evsplatibea, evsplatihea, evsplatiea	1	1
Swap	evswapbhilo	1	1
	evswapblohi	1	1
	evswaphe	1	1
	evswaphhi	1	1
	evswaphhilo	1	1
	evswaphlo	1	1
	evswaphlohi	1	1
	evswapho	1	1
Unpack	evunpkhibsi, evunpkhibui, evunpklobsi, evunpklobui	1	1
	evunpkhihf, evunpkhihsi, evunkpkhihui, evunpklohf, evunpklohsi, evunkpklohui	1	1
	evunpkhiwgsf, evunpklowgsf	1	1
Extract	evxtrb	1	1
	evxtrd	1	1
	evxtrh	1	1
Insert	evinsb	1	1
	evinsh	1	1

6.3.8 SPE Multiply and Multiply/Accumulate Instruction Timing

Table 6-22 shows instruction timing for SPE multiply and multiply/accumulate instructions.

Table 6-22. SPE Multiply and Multiply/Accumulate Instruction Timing

Instruction	Latency	Throughput
all evm{b,h,w} instructions	4	1

6.3.9 SPE Dot Product Instruction Timing

Table 6-23 shows instruction timing for SPE dot product instructions.

Table 6-23. SPE Dot Product Instruction Timing

Instruction	Latency	Throughput
all evdotp instructions	4	1

6.3.10 SPE Misc. Vector Instruction Timing

Table 6-24 shows instruction timing for SPE miscellaneous instructions.

Table 6-24. SPE Misc. Vector Instruction Timing

Operation	Instruction	Latency	Throughput
load vector for shift	evlvsl	1	1
	evlvsr	1	1
store Accumulator	evmar	1	1
load Accumulator	evmra	1	1
Bit reversed increment	brinc	1	1

6.3.11 SPE Load and Store Instruction Timing

Table 6-25 shows instruction timing for SPE load and store instructions.

Table 6-25. SPE Load and Store Instruction Timing

Instruction	Latency	Throughput
all ev loads	3	1
all ev stores	3	1

Chapter 7
Interrupts and Exceptions

The Power ISA embedded category architecture defines the mechanisms by which the e200 core implements interrupts and exceptions. This document uses the terminology ‘interrupt’ to indicate the action in which the processor saves its old context and initiates execution at a predetermined interrupt handler address. Exceptions are referred to as events, which when enabled, cause the processor to take an interrupt. This chapter uses the same terminology.

The Power ISA embedded category exception mechanism allows the processor to change to supervisor state as a result of unusual conditions arising in the execution of instructions, and from external signals, bus errors, or various internal conditions. When interrupts occur, information about the state of the processor is saved to machine state save/restore registers (SRR0/SRR1, CSRR0/CSRR1, or DSRR0/DSRR1, MCSRR0/MCSRR1) and the processor begins execution at an address (interrupt vector) determined by the interrupt vector prefix register (IVPR) and one of the interrupt vector offset registers (IVOR). Processing of instructions within the interrupt handler begins in supervisor mode.

Multiple exception conditions can map to a single interrupt vector and may be distinguished by examining registers associated with the interrupt. The exception syndrome register (ESR) is updated with information specific to the exception type when an interrupt occurs.

To prevent loss of state information, interrupt handlers must save the information stored in the machine state save/restore registers, soon after the interrupt has been taken. Four sets of these registers are implemented; SRR0 and SRR1 for noncritical interrupts, CSRR0 and CSRR1 for critical interrupts, DSRR0 and DSRR1 for debug interrupts (when the debug unit is enabled), and MCSRR0 and MCSRR1 for machine check interrupts. Hardware supports nesting of critical interrupts within noncritical interrupts, machine check interrupts within both critical and noncritical interrupts, and debug interrupts within both critical, noncritical, and machine check interrupts. It is up to the interrupt handler to save necessary state information if interrupts of a given class are re-enabled within the handler.

The following terms are used to describe the stages of exception processing:

Recognition	Exception recognition occurs when the condition that can cause an exception is identified by the processor. This is also referred to as an exception event.
Taken	An interrupt is said to be taken when control of instruction execution is passed to the interrupt handler; that is, the context is saved and the instruction at the appropriate vector offset is fetched and the interrupt handler routine begins.
Handling	Interrupt handling is performed by the software linked to the appropriate vector offset. Interrupt handling is begun in supervisor mode.

Returning from an interrupt is performed by executing an **rfi**, **rfci**, **rfdi**, or **rfmci** instruction (or **se_rfi**, **se_rfci**, **se_rfdi**, or **se_rfmci** VLE instruction) to restore state information from the respective machine state save/restore register pair.

7.1 e200 Interrupts

As specified by the Power ISA embedded category architecture, interrupts can be either precise or imprecise, synchronous or asynchronous, and critical or noncritical. Asynchronous exceptions are caused by events external to the processor's instruction execution; synchronous exceptions are directly caused by instructions or an event somehow synchronous to the program flow, such as a context switch. A precise interrupt architecturally guarantees that no instruction beyond the instruction causing the exception has (visibly) executed. Critical interrupts are provided with a separate save/restore register pair (CSRR0/CSRR1) to allow certain critical exceptions to be handled within a noncritical interrupt handler. Machine check interrupts are also provided with a separate save/restore register pair (MCSRR0/MCSRR1) to allow machine check exceptions to be handled within a noncritical or critical interrupt handler.

The types of interrupts handled are shown in Table 7-1. Refer to the "Interrupts and Exceptions" chapter in the *EREF* for exact details of each interrupt type.

Table 7-1. Interrupt Classifications

Interrupt Types	Synchronous/Asynchronous	Precise/Imprecise	Critical/Noncritical/ Debug/Machine Check
System reset	Asynchronous, nonmaskable	Imprecise	—
Machine check	—	—	Machine check
Nonmaskable input interrupt	Asynchronous, nonmaskable	Imprecise	Machine check
Critical input interrupt Watchdog timer interrupt	Asynchronous, maskable	Imprecise	Critical
External input interrupt Fixed-interval timer interrupt Decrementer interrupt	Asynchronous, maskable	Imprecise	Noncritical
Performance monitor interrupts	Synchronous/asynchronous, maskable	Imprecise	Noncritical
Instruction-based debug interrupts	Synchronous	Precise	Critical/debug
Debug interrupt (UDE) Debug imprecise interrupt	Asynchronous	Imprecise	Critical/debug
Data storage/alignment/TLB Interrupts Instruction storage/TLB interrupts	Synchronous	Precise	Noncritical

These classifications are discussed in greater detail in Section 7.6, “Interrupt Definitions.” Table 7-2 lists the interrupts implemented in the e200 and the exception conditions that cause them.

Table 7-2. Exceptions and Conditions

Interrupt Type	Interrupt Vector Offset Register	Causing Conditions
System reset	None, Vector to [*p_rstbase*[0:29]] || 0b00	Reset by assertion of **p_reset_b**.
Critical Input	IVOR0[1]	*p_critint_b* is asserted and MSR[CE] = 1.
Machine check	IVOR1	1. *p_mcp_b* transitions from negated to asserted 2. ISI, ITLB error on first instruction fetch for an exception handler 3. Parity error signaled on cache access 4. External bus error
Machine check (NMI)	IVOR1	*p_nmi_b* transitions from negated to asserted
Data storage	IVOR2	1. Access control. 2. Byte ordering due to misaligned access across page boundary to pages with mismatched E bits 3. Cache locking exception
Instruction storage	IVOR3	1. Access control. 2. Byte ordering due to misaligned instruction across page boundary to pages with mismatched VLE bits, or access to page with VLE set, and E indicating little endian. 3. Misaligned Instruction fetch due to a change of flow to an odd half word instruction boundary on a Power ISA (non-VLE) instruction page
External input	IVOR4[1]	*p_extint_b* is asserted and MSR[EE] = 1.
Alignment	IVOR5	1. **lmw**, **stmw** not word aligned 2. **lwarx** or **stwcx.** not word aligned, **lharx** or **sthcx.** not half word aligned 3. **dcbz** with disabled cache, or to W or I storage 4. SPE ld and st instructions not properly aligned
Program	IVOR6	Illegal, privileged, trap, FP enabled, AP enabled, Unimplemented operation
Floating-point unavailable	IVOR7	MSR[FP] = 0 and attempt to execute a Power ISA floating point operation
System call	IVOR8	Execution of the system call (**sc, se_sc**) instruction
AP unavailable	IVOR9	Unused by e200
Decrementer	IVOR10	As specified in the *EREF*, “Timer Facilities” chapter
Fixed interval timer	IVOR11	As specified in the *EREF*, “Timer Facilities” chapter
Watchdog timer	IVOR12	As specified in the *EREF*, “Timer Facilities” chapter
Data TLB error	IVOR13	Data translation lookup did not match a valid entry in the TLB
Instruction TLB error	IVOR14	Instruction translation lookup did not match a valid entry in the TLB

Table 7-2. Exceptions and Conditions (continued)

Interrupt Type	Interrupt Vector Offset Register	Causing Conditions
Debug	IVOR15	Trap, instruction address compare, data address compare, instruction complete, branch taken, return from interrupt, interrupt taken, debug counter, external debug event, unconditional debug event
Reserved	IVOR16–IVOR31	—
SPE/EFPU unavailable exception	IVOR32	See Section 5.2.5.1, “EFPU Unavailable Exception.”
EFPU data exception	IVOR33	See Section 5.2.5.2, “Embedded Floating-point Data Exception.”
EFPU round exception	IVOR34	See Section 5.2.5.3, “Embedded Floating-point Round Exception.”
Performance monitor	IVOR35	Performance monitor enabled condition or event

[1] Auto-vectored external and critical input interrupts use this IVOR. Vectored interrupts supply an interrupt vector offset directly.

7.2 Exception Syndrome Register

The exception syndrome register (ESR) provides a syndrome to differentiate between exceptions that can generate the same interrupt type. The e200 adds some implementation-specific bits to this register, as seen in Figure 7-1.

SPR 62 Access: Read/Write

	0–3	4	5	6	7	8	9	10	11	12	13	14	15
R W	—	PIL	PPR	PTR	FP	ST	—	DLK	ILK	AP	PUO	BO	PIE
Reset	All zeros												

	16–23	24	25	26	27–29	30	31
R W	—	SPE	—	VLEMI	0	MIF	0
Reset	All zeros						

Figure 7-1. Exception Syndrome Register (ESR)

The ESR bits are defined in Table 7-3.

Table 7-3. ESR Bit Settings

Bits	Name	Description	Associated Interrupt Type
0–3 (32–35)	—	Reserved	—
4 (36)	PIL	Illegal instruction exception	Program

Table 7-3. ESR Bit Settings (continued)

Bits	Name	Description	Associated Interrupt Type
5 (37)	PPR	Privileged instruction exception	Program
6 (38)	PTR	Trap exception	Program
7 (39)	FP	Floating-point operation	Alignment (not on the e200) Data storage (not on the e200) Data TLB (not on the e200) Program
8 (40)	ST	Store operation	Alignment Data storage Data TLB
9 (41)	—	Reserved	—
10 (42)	DLK	Data Cache Locking	Data storage
11 (43)	ILK	Instruction Cache Locking	Data storage
12 (44)	AP	Auxiliary Processor operation (Not used by the e200)	Alignment (not on the e200) Data storage (not on the e200) Data TLB (not on the e200) Program (not on the e200)
13 (45)	PUO	Unimplemented Operation exception	Program
14 (46)	BO	Byte Ordering exception Mismatched Instruction Storage exception	Data storage Instruction storage
15 (47)	PIE	Program Imprecise exception (Reserved)	Currently unused by the e200
16–23 (48–55)	—	Reserved	—
24 (56)	SPE	SPE/EFPU Operation	SPE/EFPU unavailable EFPU floating-point data exception EFPU floating-point round exception Alignment Data storage Data TLB
25 (57)	—	Reserved	—

Table 7-3. ESR Bit Settings (continued)

Bits	Name	Description	Associated Interrupt Type
26 (58)	VLEMI	VLE Mode Instruction	SPE/EFPU unavailable EFPU floating-point data exception EFPU floating-point round exception Data storage Data TLB Instruction storage Alignment Program System call
27–29 (59–61)	—	Reserved	—
30 (62)	MIF	Misaligned Instruction Fetch	Instruction storage Instruction TLB
31 (63)	—	Reserved	—

7.3 Machine State Register

The machine state register, shown in Figure 7-2, defines the state of the processor.

Access: Read/Write

	0	4	5	6	7	12	13	14	15
R W	—		UCLE	SPE	—		WE	CE	—
Reset	All zeros								

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R W	EE	PR	FP	ME	FE0	—	DE	FE1	—		IS	DS	—	PMM	RI	—
Reset	All zeros															

Figure 7-2. Machine State Register (MSR)

The MSR bits are defined in Table 7-4.

Table 7-4. MSR Bit Settings

Bits	Name	Description
0–4 (32–36)	—	Reserved
5 (37)	UCLE	User Cache Lock Enable 0 Execution of the cache locking instructions in user mode (MSR[PR] = 1) disabled; DSI exception taken instead, and ILK or DLK set in ESR. 1 Execution of the cache lock instructions in user mode enabled

Table 7-4. MSR Bit Settings (continued)

Bits	Name	Description
6 (38)	SPE	SPE/EFPU Available 0 Execution of SPE and EFPU vector instructions is disabled; SPE/EFPU unavailable exception taken instead, and SPE bit is set in ESR. 1 Execution of SPE and EFPU vector instructions is enabled.
7–12 (39–44)	—	Reserved
13 (45)	WE	Wait State (power management) Enable. This bit is defined as optional in the Power ISA embedded category architecture. 0 Power management is disabled 1 Power management is enabled. The processor can enter a power-saving mode when additional conditions are present. The mode chosen is determined by the DOZE, NAP, and SLEEP bits in the HID0 register, described in Section 2.4.11, “Hardware Implementation Dependent Register 0 (HID0).”
14 (46)	CE	Critical Interrupt Enable 0 Critical input and watchdog timer interrupts are disabled. 1 Critical input and watchdog timer interrupts are enabled.
15 (47)	—	Reserved
16 (48)	EE	External Interrupt Enable 0 External input, decrementer, and fixed-interval timer interrupts are disabled. 1 External input, decrementer, and fixed-interval timer interrupts are enabled.
17 (49)	PR	Problem State 0 The processor is in supervisor mode, can execute any instruction, and can access any resource (for example, GPRs, SPRs, MSR, etc.). 1 The processor is in user mode, cannot execute any privileged instruction, and cannot access any privileged resource.
18 (50)	FP	Floating-Point Available 0 Floating point unit is unavailable. The processor cannot execute floating-point instructions, including floating-point loads, stores, and moves. (An FP Unavailable interrupt will be generated on attempted execution of floating point instructions). 1 Floating-point unit is available. The processor can execute floating-point instructions. Note that for Zen, the floating point unit is not supported in hardware, and an Unimplemented Operation exception is generated for attempted execution of Power ISA floating point instructions when FP is set.
19 (51)	ME	Machine Check Enable 0 Asynchronous machine check interrupts are disabled 1 Asynchronous machine check interrupts are enabled
20 (52)	FE0	Floating-Point Exception Mode 0 (not used by the e200)
21 (53)	—	Reserved
22 (54)	DE	Debug Interrupt Enable 0 Debug interrupts are disabled 1 Debug interrupts are enabled
23 (55)	FE1	Floating-Point Exception Mode 1 (not used by the e200)

Table 7-4. MSR Bit Settings (continued)

Bits	Name	Description
24 (56)	—	Reserved
25 (57)	—	Reserved
26 (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).
27 (59)	DS	Data Address Space 0 The processor directs all data storage accesses to address space 0 (TS = 0 in the relevant TLB entry). 1 The processor directs all data storage accesses to address space 1 (TS = 1 in the relevant TLB entry).
28 (60)	—	Reserved
29 (61)	PMM	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 time. If this state matches an individual state specified in the performance monitor registers PMLCa *n*, the state for which monitoring is enabled, counting is enabled.
30 (62)	RI	Recoverable Interrupt. This bit is provided for software use to detect nested exception conditions. This bit is cleared by hardware when a machine check interrupt is taken.
31 (63)	—	Reserved

7.3.1 Machine Check Syndrome Register (MCSR)

When the processor takes a machine check interrupt, it updates the machine check syndrome register (MCSR) to differentiate between machine check conditions. The MCSR indicates the source of a machine check condition. When an async mchk or error report syndrome bit in the MCSR is set, the core complex asserts *p_mcp_out* for system information.

All bits in the MCSR are implemented as write one to clear. Software in the machine check handler is expected to clear the MCSR bits it has sampled prior to re-enabling MSR[ME] to avoid a redundant machine check exception and to prepare for updated status bit information on the next machine check interrupt. Hardware does not clear a bit in the MCSR other than at reset. Software typically samples MCSR early in the machine check handler and uses the sampled value to clear those bits that were set at the time of sampling. Note that additional bits may become set during the handler after sampling if an asynchronous event occurs. By writing back only the originally sampled bits, another machine check can be generated to process the new conditions after the original handler re-enables MSR[ME] either explicitly or by restoring the MSR from MSRR1 at the return.

Note that any set bit in the MCSR other than status-type bits causes a subsequent machine check interrupt once MSR[ME] = 1.

Figure 7-3 shows the MCSR.

SPR 572 Access: w1c

	0	1	2	3	4	5	6	7		9	10	11	12	13	14	15
R	MCP	IC_DPERR	CP_PERR	DC_DPERR	EXCP_ERR	IC_TPERR	DC_TPERR	IC_LKERR	DC_LKERR	—		NMI	MAV	MEA	—	IF
W	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c			w1c	w1c	w1c		w1c
Reset	All zeros															

	16	17	18	19	25	26	27	28	29	30	31
R	LD	ST	G	—		SNPERR	BUS_IRERR	BUS_DRERR	BUS_WRERR	—	
W	w1c	w1c	w1c			w1c	w1c	w1c	w1c		
Reset	All zeros										

Figure 7-3. Machine Check Syndrome Register (MCSR)

Table 7-5 describes MCSR fields.

Table 7-5. Machine Check Syndrome Register (MCSR)

Bit	Name	Description	Exception Type[1]	Recoverable
0 (32)	MCP	Machine check input pin	Async Mchk	Maybe
1 (33)	IC_DPERR	Instruction Cache data array parity error	Async Mchk	Precise
2 (34)	CP_PERR	Data Cache push parity error	Async Mchk	Unlikely
3 (35)	DC_DPERR	Data Cache data array parity error	Async Mchk	Maybe
4 (36)	EXCP_ERR	ISI, ITLB, or Bus Error on first instruction fetch for an exception handler	Async Mchk	Precise
5 (37)	IC_TPERR	Instruction Cache Tag parity error	Async Mchk	Precise
6 (38)	DC_TPERR	Data Cache Tag parity error	Async Mchk	Maybe
7 (39)	IC_LKERR	Instruction Cache Lock error Indicates a cache control operation or invalidation operation invalidated one or more locked lines in the Icache or encountered an uncorrectable lock error, or that an Icache miss with an uncorrectable lock error occurred. May also be set on locked line refill error.	Status	—
8 (40)	DC_LKERR	Data Cache Lock error Indicates a cache control operation or invalidation operation invalidated one or more locked lines in the Dcache or encountered an uncorrectable lock error, or that an Icache miss with an uncorrectable lock error occurred. May also be set on locked line refill error.	Status	—

Table 7-5. Machine Check Syndrome Register (MCSR) (continued)

Bit	Name	Description	Exception Type[1]	Recoverable
9–10 (41–42)	—	Reserved, should be cleared.	—	—
11 (43)	NMI	NMI input pin	NMI	—
12 (44)	MAV	MCAR Address Valid Indicates that the address contained in the MCAR was updated by hardware to correspond to the first detected Async Mchk error condition	Status	—
13 (45)	MEA	MCAR holds Effective Address If MAV = 1,MEA = 1 indicates that the MCAR contains an effective address and MEA = 0 indicates that the MCAR contains a physical address	Status	—
14 (46)	—	Reserved, should be cleared.	—	—
15 (47)	IF	Instruction Fetch Error Report An error occurred during the attempt to fetch an instruction. This could be due to a parity error, or an external bus error. MCSRR0 contains the instruction address.	Error Report	Precise
16 (48)	LD	Load type instruction Error Report An error occurred during the attempt to execute the load type instruction located at the address stored in MCSRR0. This could be due to a parity error or an external bus error.	Error Report	Precise
17 (49)	ST	Store type instruction Error Report An error occurred during the attempt to execute the store type instruction located at the address stored in MCSRR0. This could be due to a parity error, or on certain external bus errors.	Error Report	Precise
18 (50)	G	Guarded instruction Error Report An error occurred during the attempt to execute the load or store type instruction located at the address stored in MCSRR0 and the access was guarded and encountered an error on the external bus.	Error Report	Precise
19–:25 (51–57)	—	Reserved, should be cleared.	—	—
26 (58)	SNPERR	Snoop Lookup Error An error occurred during certain snoop operations. This is typically due to a data cache tag parity error, in which case DC_TPERR will also be set.	Async Mchk	Unlikely?
27 (59)	BUS_IRERR	Read bus error on Instruction fetch or linefill	Async Mchk	Precise if data used
28 (60)	BUS_DRERR	Read bus error on data load or linefill	Async Mchk	Precise if data used

Table 7-5. Machine Check Syndrome Register (MCSR) (continued)

Bit	Name	Description	Exception Type[1]	Recoverable
29 (61)	BUS_WRERR	Write bus error on store or cache line push	Async Mchk	Unlikely
30–31 (62–63)	—	Reserved, should be cleared.	—	—

[1] The Exception Type indicates the exception type associated with a given syndrome bit as follows:

- Error Report—indicates that this bit is only set for error report exceptions which cause machine check interrupts. These bits are only updated when the machine check interrupt is actually taken. Error report exceptions are not gated by MSR[ME]. These are synchronous exceptions. These bits remain set until cleared by software writing a 1 to the bit position(s) to be cleared.
- Status—indicates that this bit is provides additional status information regarding the logging of a machine check exception. These bits remain set until cleared by software writing a 1 to the bit position(s) to be cleared.
- NMI—indicates that this bit is only set for the non-maskable interrupt type exception which causes a machine check interrupt. This bit is only updated when the machine check interrupt is actually taken. NMI exceptions are not gated by MSR[ME]. This is an asynchronous exception. This bit remains set until cleared by software writing a 1 to the bit position.
- Async Mchk—indicates that this bit is set for an asynchronous machine check exception. These bits are set immediately upon detection of the error. Once any “Async Mchk” bit is set in the MCSR, a machine check interrupt will occur if MSR[ME] = 1. If MSR[ME] = 0, the machine check exception will remain pending. These bits remain set until cleared by software writing a 1 to the bit position(s) to be cleared.

7.4 Interrupt Vector Prefix Registers (IVPR)

The interrupt vector prefix register is used during interrupt processing for determining the starting address of a software handler used to handle an interrupt. The value contained in the vector offset field of the IVOR selected for a particular interrupt type is concatenated with the value held in the interrupt vector prefix register (IVPR) to form an instruction address from which execution is to begin. The format of IVPR is shown in Figure 7-4.

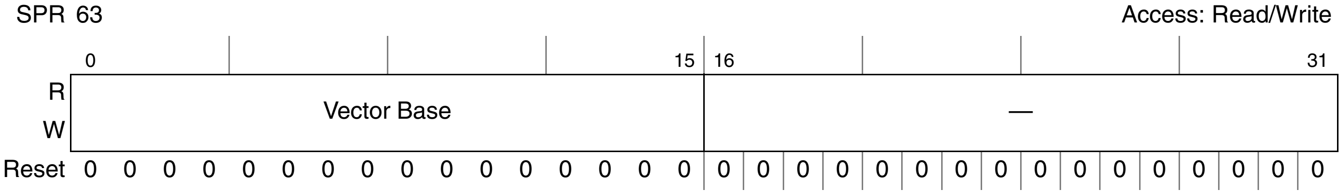


Figure 7-4. e200 Interrupt Vector Prefix Register (IVPR)

The IVPR fields are defined in Table 7-6.

Table 7-6. IVPR Register Fields

Bits	Name	Description
0–15 (32–47)	Vec Base	Vector Base This field is used to define the base location of the vector table, aligned to a 64-KB boundary. This field provides the high-order 16 bits of the location of all interrupt handlers. The contents of the IVOR*xx* register appropriate for the type of exception being processed are concatenated with the IVPR vector base to form the address of the handler in memory.
16–31 (48–63)	—	Reserved

7.5 Interrupt Vector Offset Registers (IVOR*xx*)

The interrupt vector offset registers are used during interrupt processing for determining the starting address of a software handler used to handle an interrupt. The value contained in the vector offset field of the IVOR selected for a particular interrupt type is concatenated with the value held in the interrupt vector prefix register (IVPR) to form an instruction address from which execution is to begin.

Figure 7-5 shows the format of the e200 IVORs.

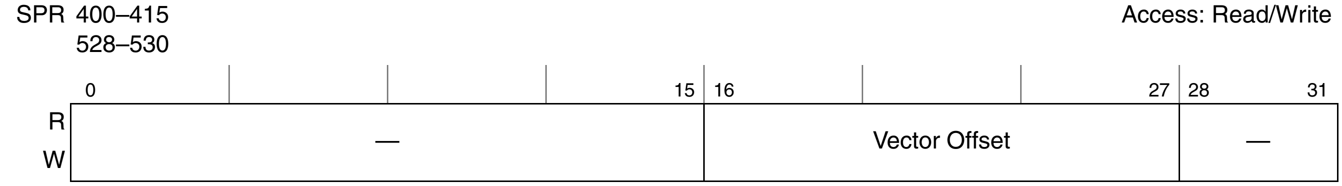


Figure 7-5. e200 Interrupt Vector Offset Register (IVOR)

The IVOR fields are defined in Table 7-7.

Table 7-7. IVOR Register Fields

Bits	Name	Description
0–15 (32–47)	—	Reserved
16–27 (48–59)	Vector Offset	Vector Offset. This field is used to provide a quad-word index from the base address provided by the IVPR to locate an interrupt handler.
28–31 (60–63)	—	Reserved

7.6 Interrupt Definitions

This section provides detailed descriptions of the interrupts listed in Table 7-8.

Table 7-8. Interrupts

IVOR Number	Type of Interrupt	Section/Page
IVOR0	Critical Input	7.6.1/7-14
IVOR1	Machine Check	7.6.2/7-14
IVOR2	Data Storage	7.6.3/7-28
IVOR3	Instruction Storage	7.6.4/7-29
IVOR4	External Input	7.6.5/7-30
IVOR5	Alignment	7.6.6/7-31
IVOR6	Program	7.6.7/7-31
IVOR7	Floating-Point Unavailable	7.6.8/7-32
IVOR8	System Call	7.6.9/7-33
IVOR9	Auxiliary Processor Unavailable	7.6.10/7-34
IVOR10	Decrementer	7.6.11/7-34
IVOR11	Fixer-Interval Timer	7.6.12/7-34
IVOR12	Watchdog Timer	7.6.13/7-35
IVOR13	Data TLB Error	7.6.14/7-36
IVOR14	Instruction TLB Error	7.6.15/7-36
IVOR15	Debug	7.6.16/7-37
IVOR16	System Reset	7.6.17/7-40
IVOR32	SPE/EFPU Unavailable	7.6.18/7-41
IVOR33	Embedded Floating-Point Data	7.6.19/7-41
IVOR34	Embedded Floating-Point Round	7.6.20/7-42
IVOR35	Performance Monitor	7.6.21/7-43

7.6.1 Critical Input Interrupt (IVOR0)

A critical input exception is signaled to the processor by the assertion of the critical interrupt pin (*p_critint_b*). When the e200 detects the exception, it takes the critical input interrupt if the exception is enabled by MSR[CE]. The *p_critint_b* input is a level-sensitive signal expected to remain asserted until the e200 acknowledges the interrupt. If *p_critint_b* is negated early, recognition of the interrupt request is not guaranteed. After the e200 begins execution of the critical interrupt handler, the system can safely negate *p_critint_b*.

A critical input interrupt may be delayed by other higher priority exceptions or if MSR[CE] is cleared when the exception occurs.

Table 7-9 lists register settings when a critical input interrupt is taken.

Table 7-9. Critical Input Interrupt—Register Settings

Register	Setting Description		
CSRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present.		
CSRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 SPE 0 WE 0 CE 0 EE 0 PR 0	FP 0 ME — FE0 0 DE —/0[1]	FE1 0 IS 0 DS 0 PMM 0 RI —
ESR	Unchanged		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	IVPR[0:15] || IVOR0[16:27] || 0b0000 (auto-vectored) IVPR[0:15] || *p_voffset*[0:11] || 0b0000 (non-auto-vectored)		

[1] DE is cleared when the debug unit is disabled. When the debug unit is enabled, control in HID0 optionally supports the clearing of DE.

When the debug unit is enabled, MSR[DE] is not automatically cleared by a critical input interrupt, but can be configured to be cleared via the HID0 register (HID0[CICLRDE]). Refer to Section 2.4.11, "Hardware Implementation Dependent Register 0 (HID0)."

IVOR0 is the vector offset register used by auto-vectored critical input interrupts to determine the interrupt handler location. e200 also provides the capability to directly vector critical input interrupts to multiple handlers by allowing a critical input interrupt request to be accompanied by a vector offset. The *p_voffset*[0:11] input signals are used in place of the value in IVOR0 to form the interrupt vector when a critical input interrupt request is not auto-vectored (*p_avec_b* negated when *p_critint_b* asserted).

7.6.2 Machine Check Interrupt (IVOR1)

The e200 implements the machine check exception as defined in the Freescale EIS machine check unit except for automatic clearing of MSR[DE]. This behavior is different from the definition in Power ISA embedded category architecture. The e200 initiates a machine check interrupt if any of the machine check sources listed in Table 7-2 is detected.

As defined in Freescale EIS machine check unit, a machine check interrupt is taken for error report and NMI type machine check conditions, even if MSR[ME] is cleared, without the processor generating an internal checkstop condition. MSR[ME] gates the processing of asynchronous type machine check sources (the sources reflected in the MCSR async mchk syndrome bits).

The Freescale EIS machine check unit defines a separate set of save/restore registers (MCSRR0–MCSRR1), a machine check syndrome register (MCSR) to record the source(s) of machine checks, and a machine check address register (MCAR) to hold an address associated with a machine check for certain classes of machine checks. Return from machine check instructions (**rfmci**, **se_rfmci**) are also provided to support returns using MCSRR0–MCSRR1.

The MSR[RI] status bit is provided for software use in determining if multiple nested machine check exceptions have occurred. Software may interrogate MCSRR1[RI] to determine whether a machine check occurred during the initial portion of a machine check handler prior to the handler code that sets MSR[RI] to one to indicate that the handler can now tolerate another machine check condition without losing state necessary for recovery.

MSR[DE] is not automatically cleared by a machine check exception, but can be configured to be cleared or left unchanged via the HID0 register (HID0[MCCLRDE]). Refer to Section 2.4.11, “Hardware Implementation Dependent Register 0 (HID0).”

7.6.2.1 Machine Check Causes

Machine check causes are divided into different types, as follows:

- Error report machine check conditions
- Nonmaskable interrupt (NMI) machine check exceptions
- Asynchronous machine check exceptions

This division is intended to facilitate machine check handling in uniprocessor, multiprocessor, and multithreaded systems. Although the initial implementation of the e200z7 does not implement multithreading, future versions are expected to, and the machine check model will remain compatible. In addition, the model is equally applicable to a single-threaded design.

7.6.2.1.1 Error Report Machine Check Exceptions

Error report machine check exceptions are directly associated with the current instruction execution stream and are presented to the interrupt mechanism in a manner analogous to an instruction storage or data storage interrupt. Since the execution stream cannot continue execution without suffering from corruption of architectural state, these exceptions are not masked by MSR[ME]. Error report machine check exceptions are not necessarily recoverable if they occur during the initial portion of a machine check handler. MSR[RI] and MCSRR1[RI] are provided to assist software in determining recoverability.

For error report machine check exceptions, the MCSR (machine check status register) is updated only when the machine check interrupt is actually taken. The MCAR is not updated for error report machine check exceptions.

Error report machine check exceptions encountered by program execution can be flushed if an older exception exists or if an asynchronous interrupt or machine check is taken before the instruction that encountered the error becomes the oldest instruction in the machine. In this case, the corresponding MCSR bit is not set due to the flushed exception condition (although the corresponding bit may have already been set by a previous instruction's exception).

Note that an async machine check condition may occur for the same error condition prior to the error report machine check. The error report machine check may be discarded.

Depending on the type of error, hardware sets the MCSR IF, LD, G, or ST bit(s) to reflect the error being reported. Software is responsible for clearing these syndrome bits by writing a one to the bit(s) to be cleared. Hardware does not clear an error report bit once it is set. The bits are set as follows:

- MCSR[IF] is set if the error occurs during an instruction fetch
- MCSR[LD] is set if the error occurs for a load instruction. If the error occurs for a guarded load and the error source was from the external bus, MCSR[G] is also set.
- MCSR[ST] is set if the error occurs in the data cache (parity) or MMU (DTLB error or DSI) for a store type instruction (including **dcbz**), if an external termination error is received on a cache-inhibited guarded store or on a store conditional instruction, or if an unsuccessful flush with invalidation occurs on a store conditional instruction due to a tag or data parity error or external bus error. If an external termination error occurs on a cache-inhibited guarded store or on a guarded store conditional, MCSR[G] is also set.

Note that most (if not all) error report machine check exceptions are accompanied by an associated asynchronous machine check exception on a single-threaded e200z7, although this is not generally the case for a multithreaded version.

Table 7-10 shows the error report machine check exceptions.

Table 7-10. Error Report Machine Check Exceptions

Synchronous Machine Check Source	Error Type	MCSR Updates	Precise[1]
Instruction Fetch	(Icache tag array parity error or data array parity error) and L1CSR1[ICEA] = 00	IF	Yes
	(Icache uncorrectable tag array parity error or data array parity error and L1CSR1[ICEA] = 01 and line potentially locked (locked or lock parity error) was invalidated	IF	Yes
	Cacheable miss and L1CSR1[ICEA] = 00 and any line with lock parity error	IF	Yes
	Cacheable miss and L1CSR1[ICEA] = 01 and and line with uncorrectable lock parity error was invalidated	IF	Yes
	External termination error	IF	Yes

Table 7-10. Error Report Machine Check Exceptions (continued)

Synchronous Machine Check Source	Error Type	MCSR Updates	Precise[1]
Load instruction	Dcache tag array parity error or data array parity error) and L1CSR0[DCEA] = 00	LD	Yes
	(Dcache uncorrectable tag array parity error or data array parity error) and L1CSR0[DCEA] = 01 and (line potentially locked (locked or lock parity error) was invalidated, or line potentially dirty (dirty or dirty parity error))	LD	Yes
	Cacheable miss and L1CSR0[DCEA] = 00 and any line with lock parity error, or dirty parity error on replacement line	LD	Yes
	Cacheable miss and L1CSR0[DCEA] = 01 and line with uncorrectable lock parity error was invalidated	LD	Yes
	External termination error on load data	LD, [G][2]	Yes
Load and reserve instruction	Dcache tag array parity error and L1CSR0[DCEA] = 00	LD	Yes
	Dcache hit and dirty parity error and L1CSR0[DCEA] = 00	LD	Yes
	(Dcache uncorrectable tag array parity error or data array parity error) and L1CSR0[DCEA] = 01 and line potentially dirty (dirty or dirty parity error)	LD	Yes
	Dcache data push parity error[3]	LD	Yes
	External termination error on dirty push[3]	LD	Yes
	External termination error on load	LD, [G][2]	Yes
Store instruction	Dcache tag array parity error and L1CSR0[DCEA] = 00	ST	Yes
	Dcache uncorrectable tag array parity error and L1CSR0[DCEA] = 01 and (line potentially locked (locked or lock parity error) was invalidated, or line potentially dirty (dirty or dirty parity error))	ST	Yes
	Cacheable miss and L1CSR0[DCEA] = 00 and any line with lock parity error, or dirty parity error on replacement line	ST	Yes
	Cacheable miss and L1CSR0[DCEA] = 01 and line with uncorrectable lock parity error was invalidated	ST	Yes
	External termination error on unbuffered store[4]	ST, [G][7]	Yes
	External termination error on CI+G store[5]	ST, G	Yes

Table 7-10. Error Report Machine Check Exceptions (continued)

Synchronous Machine Check Source	Error Type	MCSR Updates	Precise[1]
Store conditional instruction	Dcache tag array parity error and L1CSR0[DCEA] = 00	ST	Yes
	Dcache hit and dirty parity error and L1CSR0[DCEA]= 00	ST	Yes
	Dcache uncorrectable tag array parity error and L1CSR0[DCEA]= 01 and line potentially dirty (dirty or dirty parity error)	ST	Yes
	Dcache data push parity error[6]	ST	Yes
	External termination error on dirty push[6]	ST	Yes
	External termination error on store conditional	ST, [G][7]	Yes
dcbst instruction	Dcache tag array parity error and miss and L1CSR0[DCEA] = 00 and any line with error is potentially dirty (dirty or dirty parity error)	LD	Yes
	Dcache uncorrectable tag array parity error and cacheable miss and L1CSR0[DCEA] = 01 and line potentially dirty (dirty or dirty parity error)	LD	Yes
dcbf instruction	Dcache tag array parity error and miss and L1CSR0[DCEA] = 00 and (line potentially locked (locked or lock parity error) or line potentially dirty (dirty or dirty parity error))	LD	Yes
	Dcache uncorrectable tag array parity error and miss and L1CSR0[DCEA] = 01 and (line potentially locked (locked or lock parity error) or line potentially dirty (dirty or dirty parity error))	LD	Yes
dcblc instruction	Dcache tag array parity error and cacheable miss and L1CSR0[DCEA] = 00 and line potentially locked (locked or lock parity error)	LD	Yes
	Dcache uncorrectable tag array parity error and cacheable miss and L1CSR0[DCEA] = 01 and line potentially locked (locked or lock parity error)	LD	Yes

Table 7-10. Error Report Machine Check Exceptions (continued)

Synchronous Machine Check Source	Error Type	MCSR Updates	Precise[1]
dcbtls, dcbtstls instruction	(Dcache tag array parity error or lock error) and miss and L1CSR0[DCEA] = 00	LD	Yes
	Dcache uncorrectable tag array parity error and cacheable miss and L1CSR0[DCEA] = 01 and (line potentially locked (locked or lock parity error) was invalidated, or line potentially dirty (dirty or dirty parity error))	LD	Yes
	Cacheable miss and L1CSR0[DCEA] = 00 and any line with lock parity error, or dirty parity error on replacement line	LD	Yes
	Cacheable miss and L1CSR0[DCEA] = 01 and line with uncorrectable lock parity error was invalidated	LD	Yes
	External termination error on linefill	LD, [G][2]	Yes
dcbz instruction[8]	(Dcache tag array parity error or lock error) and cacheable miss and L1CSR0[DCEA] = 00	ST	Yes
	Dcache uncorrectable tag array parity error and cacheable miss and L1CSR0[DCEA] = 01 and (line potentially locked (locked or lock parity error) was invalidated, or line potentially dirty (dirty or dirty parity error))	ST	Yes
	Cacheable miss and L1CSR0[DCEA] = 00 and any line with lock parity error, or dirty parity error on replacement line	ST	Yes
dcbz instruction[8]	Cacheable miss and L1CSR0[DCEA] = 01 and line with uncorrectable lock parity error was invalidated	ST	Yes
L1FINV0 flush or flush with invalidate operation	Dcache tag parity error and L1CSR0[DCEA] = 00 and line potentially dirty (dirty or dirty parity error)	LD	Yes
	Dcache uncorrectable tag parity error and L1CSR0[DCEA] = 01 and line potentially dirty (dirty or dirty parity error)		
icblc instruction	Icache tag array parity error and cacheable miss and L1CSR1[ICEA] = 00 and line potentially locked (locked or lock parity error)	IF	Yes
	Icache uncorrectable tag array parity error and cacheable miss and L1CSR1[ICEA] = 01 and line potentially locked (locked or lock parity error) was invalidated	IF	Yes

Table 7-10. Error Report Machine Check Exceptions (continued)

Synchronous Machine Check Source	Error Type	MCSR Updates	Precise[1]
icbtls instruction	(Icache tag array parity error or lock error) and cacheable miss and L1CSR1[ICEA] = 00	IF	Yes
	Icache uncorrectable tag array parity error and cacheable miss and L1CSR1[ICEA] = 01 and line potentially locked (locked or lock parity error) was invalidated	IF	Yes
	External termination error on linefill	IF	Yes
Exception Vectoring	ISI, ITLB, or Bus Error on first instruction fetch for an exception handler	IF	Yes

[1] MCSRR0 will point to the instruction associated with the machine check condition

[2] G will be set if the load was a guarded load.

[3] Can only occur if the load and reserve causes a dirty line to be flushed

[4] Store may be unbuffered if the store buffer is disabled, and the store is not allocating a cache line

[5] Only reported if the store was a cache-inhibited guarded store

[6] Can only occur if the store conditional causes a dirty line to be flushed

[7] Only reported if the store was a guarded store.

[8] Alignment error may be generated concurrently

7.6.2.1.2 Nonmaskable Interrupt Machine Check Exceptions

Nonmaskable interrupt exceptions are reported via the *p_nmi_b* input pin, which is transition sensitive. MSR[ME] does not gate NMI exceptions, thus they are not necessarily recoverable if an NMI exception occurs during the initial part of a machine check exception handler. MSR[RI] and MCSRR1[RI] assist software in determining recoverability.

For NMI machine check exceptions, MCSR[NMI] is updated (set) only when the machine check interrupt is actually taken. Hardware does not clear the MCSR[NMI] syndrome bit. Software is responsible for clearing this syndrome bit by writing a one to the bit(s) to be cleared. Hardware does not clear an NMI bit once it is set.

The MCAR is not updated for NMI machine check exceptions.

7.6.2.1.3 Asynchronous Machine Check Exceptions

The remainder of machine check exceptions are classified as asynchronous machine check exceptions, as they are reported directly by the subsystem or resource which detected the condition. For many cases, the asynchronous condition is reported simultaneously with a corresponding error report condition. These conditions are reported by immediately setting the corresponding MCSR async mchk syndrome bit, regardless of the state of MSR[ME]. Interrupts due to asynchronous machine check exceptions are gated by MSR[ME]. If MSR[ME] = 0 at the time an async mchk bit becomes set, the interrupt is postponed until MSR[ME] is later set (although a machine check interrupt may occur at the time of the event due to an error report exception). Asynchronous events are cumulative; hardware does not clear an async mchk syndrome bit. Software is responsible for clearing these syndrome bits by writing a one to the bit(s) to be cleared. Hardware does not clear an async mchk bit once it is set.

If MCSR[MAV] is cleared at the time an asynchronous machine check exception occurs that has a corresponding address (either an effective or real address) to log in the MCAR, the MCAR and MCSR[MEA] are updated, and MCSR[MAV] is set. If MCSR[MAV] was previously set, the MCAR and MCSR[MEA] are not affected.

Table 7-11 details all asynchronous machine check sources.

Table 7-11. Asynchronous Machine Check Exceptions

Asynchronous Machine Check Source	Transaction Source	Error Type	MCSR Update[1]		MCAR Update[2]
External	N/A	Machine Check Input Pin[3]	MCP		None
Instruction Cache	Instruction Fetch	Tag array parity error and L1CSR1[ICEA] = 00	MAV	IC_TPERR	RA
		Icache hit, data array parity error and L1CSR1[ICEA] = 00		IC_DPERR	RA
		Icache cacheable miss, lock error, and L1CSR1[ICEA] = 00		IC_TPERR, IC_LKERR	RA
		L1CSR1[ICEA] = 01 and auto-invalidation of locked or potentially locked line due to uncorrectable tag parity error		IC_TPERR, IC_LKERR	RA
	icblc	Tag array parity error and cacheable miss and L1CSR1[ICEA] = 00 and line potentially locked (locked or lock parity error)		IC_TPERR, [IC_LKERR (if lock parity error)]	RA
	icbtls	(Tag array parity error or lock error) and cacheable miss and L1CSR1[ICEA] = 00		IC_TPERR, [IC_LKERR (if lock parity error)]	RA
	icblc icbtls	L1CSR1[ICEA] = 01 and Auto-invalidation of locked line due to uncorrectable tag parity error		IC_TPERR, IC_LKERR	RA
Data Cache	dcblc	Tag array parity error and cacheable miss and L1CSR0[DCEA] = 00 and line potentially locked (lock or lock parity error)	MAV	DC_TPERR, [DC_LKERR (if lock parity error)]	RA

Table 7-11. Asynchronous Machine Check Exceptions (continued)

Asynchronous Machine Check Source	Transaction Source	Error Type	MCSR Update[1]		MCAR Update[2]
Data Cache	load or store	Tag array parity error and L1CSR0[DCEA] = 00	MAV	DC_TPERR, [DC_LKERR (if lock parity error on line with tag parity error)]	RA
	L1FINV0 flush or flush w/inv & line dirty or potentially dirty	Tag array parity error and L1CSR0[DCEA] = 00		DC_TPERR	RA
	dcbtls dcbtstls dcbz	Tag array parity error and cacheable miss and L1CSR0[DCEA] = 00		DC_TPERR	RA
	dcbf	Tag array parity error and miss and L1CSR0[DCEA] = 00 and (line potentially locked (locked or lock parity error) or line potentially dirty (dirty or dirty parity error))		DC_TPERR, [DC_LKERR (if lock parity error)]	RA
	atomic load or store	Hit and L1CSR0[DCEA] = 00 and line has dirty parity error		DC_TPERR	RA
	dcbst, atomic load or store	Tag array parity error and miss and L1CSR0[DCEA] = 00 and line potentially dirty (dirty or dirty parity error)		DC_TPERR, [DC_LKERR (if lock parity error)]	RA
	load or store dcbtls dcbtstls dcbz	Dcache cacheable miss and L1CSR0[DCEA] = 00 and lock parity error		DC_TPERR, DC_LKERR	RA
	load or store dcbtls dcbtstls dcbz	Dcache cacheable miss and L1CSR0[DCEA] = 00 and dirty parity error on line to be replaced		DC_TPERR	RA
	load or store dcbtls dcbtstls dcbz	Dcache uncorrectable tag array parity error and L1CSR0[DCEA] = 01 and (line potentially locked (locked or lock parity error) was invalidated, or line potentially dirty (dirty or dirty parity error))		DC_TPERR, [DC_LKERR]	RA

Table 7-11. Asynchronous Machine Check Exceptions (continued)

Asynchronous Machine Check Source	Transaction Source	Error Type	MCSR Update[1]		MCAR Update[2]
Data Cache	L1FINV0 flush w/inv	Dcache uncorrectable tag array parity error and L1CSR0[DCEA] = 01 and line potentially dirty (dirty or dirty parity error))	MAV	DC_TPERR	RA
	dcblc	Dcache uncorrectable tag array parity error and L1CSR0[DCEA] = 01 and (line potentially locked (locked or lock parity error) was invalidated		DC_TPERR, [DC_LKERR]	RA
	dcbst, atomic load or store	Dcache uncorrectable tag array parity error and L1CSR0[DCEA] = 01 and line potentially dirty (dirty or dirty parity error)		DC_TPERR, [DC_LKERR (if uncorrectable lock parity error)]	RA
	dcbf	Dcache uncorrectable tag array parity error and L1CSR0[DCEA] = 01 and (line potentially locked (locked or lock parity error) or line potentially dirty (dirty or dirty parity error))		DC_TPERR, [DC_LKERR (if uncorrectable lock parity error)]	RA
	L1FINV0 flush	Dcache uncorrectable tag array parity error and L1CSR0[DCEA] = 01 and line potentially dirty (dirty or dirty parity error)		DC_TPERR	RA
	load	Dcache hit, data array parity error and L1CSR0[DCEA] = 00		DC_DPERR	RA
		Dcache hit, data array parity error and L1CSR0[DCEA] = 01 and line potentially dirty (dirty or dirty parity error)		DC_DPERR	RA
	replacement push dcbf push dcbst push L1FINV0 push reservation instruction forced-push	Data array push parity error		CP_PERR	RA
Data Cache	snoop lookup	Tag array parity error and (cacheable miss, or hit only to way with tag parity error)	MAV	DC_TPERR, SNPERR	RA (snoop address)

Table 7-11. Asynchronous Machine Check Exceptions (continued)

Asynchronous Machine Check Source	Transaction Source	Error Type	MCSR Update[1]		MCAR Update[2]
BIU	store or push	Bus error on write or push	MAV	BUS_WRERR	RA
	load store/w allocate dcbtls dcbtstls	Bus error on load fetch or linefill		BUS_DRERR	RA
	load	Bus error on error recovery refill		BUS_DRERR	RA
	instruction fetch	Bus error on error recovery refill		BUS_IRERR	RA
	icbtls CI or cache disabled Ifetch	Bus error on icbtls fill Bus error on CI Ifetch Bus error on cache disabled Ifetch		BUS_IRERR	RA
	load	Bus error on locked line error recovery refill		BUS_DRERR, DC_LKERR	RA
	instruction fetch	Bus error on locked line error recovery refill		BUS_IRERR, IC_LKERR	RA
Snoop Lookup	INV snoop command type	Tag array parity error and (miss, or hit only to way with tag parity error)	MAV	SNPERR, DC_TPERR	RA[4]
Exception Vectoring	first instruction fetch for an exception handler	ISI or Bus Error on first instruction fetch for an exception handler	MAV	EXCP_ERR	RA
	first instruction fetch for an exception handler	ITLB Error on first instruction fetch for an exception handler	MAV	EXCP_ERR	EA

[1] The MCSR update column indicates which bits in the MCSR will be updated when the exception is logged.

[2] The MCAR update column indicates whether or not the error will provide either a real address (RA), effective address (EA), or no address (none) which is associated with the error.

[3] The machine check input pin is used by the platform logic to indicate machine check type errors which are detected by the platform. Software must query error logging information within the platform logic to determine the specific error condition and source.

[4] The RA stored in the MCAR for this case will be Snoop Address value, with the index bits set to 0.

Table 7-12 details the priority of asynchronous machine check updates to the MCAR when multiple simultaneous async machine check conditions occur. Note that since a lower priority condition may occur and then a higher priority condition may subsequently occur prior to the machine check interrupt handler reading the MCSR and MCAR, the interrupt handler may not necessarily see the higher priority MCAR value, even though multiple MCSR bits are set.

Table 7-12. Asynchronous Machine Check MCAR Update Priority

Priority (0 = highest)	Asynchronous Machine Check Source	Transaction Source	Error Type	(MCSR Update)
0	Exception Vectoring	first instruction fetch for an exception handler	ISI or Bus Error on first instruction fetch for an exception handler	EXCP_ERR
		first instruction fetch for an exception handler	ITLB Error on first instruction fetch for an exception handler	EXCP_ERR
1	Data Cache	replacement push dcbf push dcbst push L1FINV0 push reservation-type instruction forced push	Dirty push parity error	CP_PERR
2	BIU	store or push	Bus error on write or push	BUS_WRERR
3	Data Cache	load or store dcblc dcbtls dcbtstls dcbz	Uncorrectable tag array parity error and L1CSR0[DCEA] = 01 and locked line invalidated	DC_TPERR, DC_LKERR
4	Instruction Cache	icblc icbtls instruction fetch	Uncorrectable tag array parity error, L1CSR1[ICEA] = 01, and locked line invalidated	IC_TPERR, IC_LKERR
5	BIU	load	Bus error on locked line error recovery refill	BUS_DRERR, DC_LKERR
6	BIU	instruction fetch	Bus error on locked line error recovery refill	BUS_IRERR, IC_LKERR
7	Data Cache	load or store dcbf dcbtls dcbtstls dcbz L1FINV0 flush or flush w/inv & line dirty	Tag array parity error and L1CSR0[DCEA] = 00	DC_TPERR
			Uncorrectable tag array parity error and L1CSR0[DCEA] = 01 and line dirty or potentially dirty	
7	Data Cache	load or store dcbtls dcbtstls dcbz	Cacheable miss and L1CSR0[DCEA] = 00 and dirty parity error on line to be replaced	DC_TPERR
7	Data Cache	load or store dcbtls dcbtstls dcbz	Cacheable miss and L1CSR0[DCEA] = 00 and lock parity error	DC_TPERR, DC_LKERR
			Cacheable miss and L1CSR0[DCEA] = 01 and uncorrectable lock parity error	

Table 7-12. Asynchronous Machine Check MCAR Update Priority (continued)

Priority (0 = highest)	Asynchronous Machine Check Source	Transaction Source	Error Type	(MCSR Update)
8	Data Cache	dcbst	Tag array parity error & L1CSR0[DCEA] = 00 & line potentially dirty (dirty or dirty parity error)	DC_TPERR, [DC_LKERR (if lock parity error)]
			Uncorrectable tag array parity error, L1CSR0[DCEA] = 01, line potentially dirty (dirty or dirty parity error)	DC_TPERR, [DC_LKERR (if uncorrectable lock parity error)]
9	Data Cache	dcblc	Tag array parity error, L1CSR0[DCEA] = 00, line potentially locked (locked or lock parity error)	DC_TPERR, [DC_LKERR (if lock parity error)]
			Uncorrectable tag array parity error, L1CSR0[DCEA] = 01, and line potentially locked (locked or lock parity error)	DC_TPERR, [DC_LKERR (if uncorrectable lock parity error)]
10	Data Cache	load	Data array parity error and L1CSR0[DCEA] = 00	DC_DPERR
			Data array parity error, line dirty or potentially dirty, L1CSR0[DCEA] = 01	
11	Instruction Cache	icblc	Tag array parity error, L1CSR1[ICEA] = 00, line locked or lock parity error	IC_TPERR, [IC_LKERR]
		icbtls	Tag array parity error and L1CSR1[ICEA] = 00	IC_TPERR
			Cacheable miss, L1CSR1[ICEA] = 00, lock parity error	IC_TPERR, IC_LKERR
			Cacheable miss, L1CSR1[ICEA] = 01, uncorrectable lock parity error	
12	BIU	load store/w allocate dcbtls dcbtstls CI or cache disabled Ifetch	Bus error on load or linefill or data refill Bus error on CI Ifetch Bus error on cache disabled Ifetch	BUS_DRERR

Table 7-12. Asynchronous Machine Check MCAR Update Priority (continued)

Priority (0 = highest)	Asynchronous Machine Check Source	Transaction Source	Error Type	(MCSR Update)
13	BIU	icbtls CI or cache disabled Ifetch	Bus error on linefill or data refill Bus error on CI Ifetch Bus error on cache disabled Ifetch	BUS_IRERR
14	Data Cache	snoop lookup	Tag parity error and (miss, or hit only to way with tag parity error)	DC_TPERR, SNPERR
15	Instruction Cache	Instruction Fetch	Tag array parity error and L1CSR1[ICEA] = 00	IC_TPERR
16	Instruction Cache		Data array parity error and L1CSR1[ICEA] = 00	IC_DPERR
17	Instruction Cache	Instruction Fetch	Cacheable miss, L1CSR1[ICEA] = 00, lock parity error	IC_TPERR, IC_LKERR
			Cacheable miss, L1CSR1[ICEA] = 01, uncorrectable lock parity error	

7.6.2.2 Machine Check Interrupt Actions

Machine check interrupts for error report conditions and NMI are enabled and taken regardless of the state of MSR[ME]. Machine check interrupts due to an async mchk syndrome bit being set in MCSR are only taken when MSR[ME] = 1. When a machine check interrupt is taken, registers are updated as shown in Table 7-13.

Table 7-13. Machine Check Interrupt—Register Settings

Register	Setting Description		
MCSRR0	On a best-effort basis, the e200 sets this to the address of some instruction that was executing or about to be executing when the machine check condition occurred.		
MCSRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 SPE 0 WE 0 CE 0 EE 0 PR 0	FP 0 ME 0 FE0 0 DE 0/—[1]	FE1 0 IS 0 DS 0 PMM 0 RI 0
ESR	Unchanged		
MCSR	Updated to reflect the source(s) of a machine check. Hardware only sets appropriate bits, no previously set bits are cleared by hardware.		
MCAR	See Table 7-11		
Vector	IVPR[0–15] || IVOR1[16–27] || 0b0000		

[1] DE is cleared when the debug unit is disabled. When the debug unit is enabled, control in HID0 optionally supports clearing DE.

The machine check syndrome register is provided to identify the source(s) of a machine check and may be used to identify recoverable events in conjunction with MCSRR1[RI].

The MSR[RI] status bit is provided for software use in determining if multiple nested machine check exceptions have occurred. Software may interrogate MCSRR1[RI] to determine if a machine check occurred during the initial portion of a machine check handler prior to the handler code that sets MSR[RI] to indicate that the handler can now tolerate another machine check condition without losing state necessary for recovery. The interrupt handler should set MSR[RI] as soon as possible after saving off working registers and MCSRR0,1 to avoid loss of state if another machine check condition were to occur.

The machine check input pin *p_mcp_b* can be masked by HID0[EMCP].

The nonmaskable interrupt machine check input pin *p_nmi_b* is never masked.

Precise external termination errors occur when a load or cache-inhibited or guarded store is terminated by assertion of *p_tea_b* (external bus ERROR termination response); these result in both an error report and an async mchk machine check exception.

Some machine check exceptions are unrecoverable in the sense that execution cannot resume in the context that existed before the interrupt. However, system software can use the machine check interrupt handler to try to identify and recover from the machine check condition.

7.6.2.3 Checkstop State

Machine checks no longer result in a checkstop and there is no checkstop state implemented on the e200z7.

7.6.3 Data Storage Interrupt (IVOR2)

A data storage interrupt (DSI) may occur if no higher priority exception exists and one of the following exception conditions exists:

- Read or write access control exception condition
- Byte ordering exception condition
- Cache locking exception condition

Access control is defined as in the Power ISA embedded category. A byte ordering exception condition occurs for any misaligned access across a page boundary to pages with mismatched E bits. Cache locking exception conditions occur for any attempt to execute a **dcbtls**, **dcbtstls**, **dcblc**, **icbtls**, or **icblc** in user mode with MSR[UCLE] = 0.

Table 7-14 lists register settings when a DSI is taken.

Table 7-14. Data Storage Interrupt—Register Settings

Register	Setting Description
SRR0	Set to the effective address of the excepting load/store instruction.
SRR1	Set to the contents of the MSR at the time of the interrupt

Table 7-14. Data Storage Interrupt—Register Settings (continued)

MSR	UCLE 0 SPE 0 WE 0 CE — EE 0 PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0 PMM 0 RI —
ESR	Access: Byte ordering: Cache locking:	[ST], [SPE], [VLEMI]. All other bits cleared. [ST], [SPE], [VLEMI], BO. All other bits cleared. (DLK, ILK), [VLEMI], [ST]. All other bits cleared.	
MCSR	Unchanged		
DEAR	For access and byte ordering exceptions, set to the effective address of a byte within the page whose access caused the violation. Undefined on cache locking exceptions (The e200 does not update the DEAR on a cache locking exception)		
Vector	IVPR[0–15] || IVOR2[16–27] || 0b0000		

7.6.4 Instruction Storage Interrupt (IVOR3)

An instruction storage interrupt (ISI) occurs when no higher priority exception exists and an execute access control exception occurs. This interrupt is implemented as defined by the Power ISA embedded category, with the addition of misaligned instruction fetch exceptions, and the extension of the byte ordering exception status to also cover mismatched instruction storage exceptions.

Exception extensions implemented in the e200 for Power ISA VLE involve extending the definition of the instruction storage interrupt to include byte ordering exceptions for instruction accesses, misaligned instruction fetch exceptions, and corresponding updates to the ESR as shown in Table 7-15 and Table 7-16.

Table 7-15. ISI Exceptions and Conditions

Interrupt Type	Interrupt Vector Offset Register	Causing Conditions
Instruction storage	IVOR 3	1. Access control. 2. Byte ordering due to misaligned instruction across page boundary to pages with mismatched VLE bits, or access to page with VLE set, and E indicating little endian. 3. Misaligned Instruction fetch due to a change of flow to an odd half word instruction boundary on a Power ISA (non-VLE) instruction page

Table 7-16 lists register settings when an ISI is taken.

Table 7-16. Instruction Storage Interrupt—Register Settings

Register	Setting Description
SRR0	Set to the effective address of the excepting instruction
SRR1	Set to the contents of the MSR at the time of the interrupt

Table 7-16. Instruction Storage Interrupt—Register Settings (continued)

<table>
<tr><td>MSR</td><td>UCLE 0
SPE 0
WE 0
CE —
EE 0
PR 0</td><td>FP 0
ME —
FE0 0
DE —</td><td>FE1 0
IS 0
DS 0
PMM 0
RI —</td></tr>
<tr><td>ESR</td><td colspan="3">[BO, MIF, VLEMI]. All other bits cleared.</td></tr>
<tr><td>MCSR</td><td colspan="3">Unchanged</td></tr>
<tr><td>DEAR</td><td colspan="3">Unchanged</td></tr>
<tr><td>Vector</td><td colspan="3">IVPR[0:15] || IVOR3[16:27] || 0b0000</td></tr>
</table>

7.6.5 External Input Interrupt (IVOR4)

An external input exception is signaled to the processor by the assertion of the external interrupt pin (*p_extint_b*). The *p_extint_b* input is a level-sensitive signal expected to remain asserted until the e200 acknowledges the external interrupt. If *p_extint_b* is negated early, recognition of the interrupt request is not guaranteed. When e200 detects the exception, if the exception is enabled by MSR[EE], the e200 takes the external input interrupt.

An external input interrupt may be delayed by other higher priority exceptions or if MSR[EE] is cleared when the exception occurs.

Table 7-17 lists register settings when an external input interrupt is taken.

Table 7-17. External Input Interrupt—Register Settings

<table>
<tr><th>Register</th><th colspan="3">Setting Description</th></tr>
<tr><td>SRR0</td><td colspan="3">Set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present.</td></tr>
<tr><td>SRR1</td><td colspan="3">Set to the contents of the MSR at the time of the interrupt</td></tr>
<tr><td>MSR</td><td>UCLE 0
SPE 0
WE 0
CE —
EE 0
PR 0</td><td>FP 0
ME —
FE0 0
DE —</td><td>FE1 0
IS 0
DS 0
PMM 0
RI —</td></tr>
<tr><td>ESR</td><td colspan="3">Unchanged</td></tr>
<tr><td>MCSR</td><td colspan="3">Unchanged</td></tr>
<tr><td>DEAR</td><td colspan="3">Unchanged</td></tr>
<tr><td>Vector</td><td colspan="3">IVPR[0–15] || IVOR4[16–27] || 0b0000
IVPR[0–15] || <i>p_voffset</i>[0:11] || 0b0000 (non-auto-vectored)</td></tr>
</table>

IVOR4 is the vector offset register used by auto-vectored external input interrupts to determine the interrupt handler location. The e200 also provides the capability to directly vector external input interrupts to multiple handlers by allowing a external input interrupt request to be accompanied by a vector offset.

The *p_voffset*[0:11] input signals are used in place of the value in IVOR4 when a external input interrupt request is not auto-vectored (*p_avec_b* negated when *p_extint_b* asserted).

7.6.6 Alignment Interrupt (IVOR5)

The e200 implements the alignment interrupt as defined by the Power ISA embedded category. An alignment exception is generated when any of the following occurs:

- The operand of **lmw** or **stmw** not word aligned.
- The operand of **lwarx** or **stwcx.** not word aligned.
- The operand of **lharx** or **sthcx.** not half word aligned.
- Execution of a **dcbz** instruction is attempted with a disabled cache.
- Execution of a **dcbz** instruction with an enabled cache and W or I =1.
- Execution of a SPE load or store instruction which is not properly aligned.

Table 7-18 lists register settings when an alignment interrupt is taken.

Table 7-18. Alignment Interrupt—Register Settings

Register	Setting Description		
SRR0	Set to the effective address of the excepting load/store instruction.		
SRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 SPE 0 WE 0 CE — EE 0 PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0 PMM 0 RI —
ESR	[ST], [SPE], [VLEMI]. All other bits cleared.		
MCSR	Unchanged		
DEAR	Set to the effective address of a byte of the load or store whose access caused the violation.		
Vector	IVPR[0–15] || IVOR5[16–27] || 0b0000		

7.6.7 Program Interrupt (IVOR6)

The e200 implements the program interrupt as defined by the Power ISA embedded category. A program interrupt occurs when no higher priority exception exists and one or more of the following exception conditions defined in Power ISA embedded category occur:

- Illegal instruction exception
- Privileged instruction exception
- Trap exception
- Unimplemented operation exception

The e200 invokes an illegal instruction program exception on attempted execution of the following instructions:

- Instruction from the illegal instruction class
- **mtspr** and **mfspr** instructions with an undefined SPR specified
- **mtdcr** and **mfdcr** instructions with an undefined DCR specified

The e200 invokes a privileged instruction program exception on attempted execution of the following instructions when MSR[PR] = 1 (user mode):

- A privileged instruction
- **mtspr** and **mfspr** instructions that specify a SPRN value with SPRN[5] = 1 (even if the SPR is undefined).

The e200 invokes a trap exception on execution of the **tw** and **twi** instructions if the trap conditions are met and the exception is not also enabled as a debug interrupt.

The e200 invokes an unimplemented operation program exception on attempted execution of the instructions **lswi**, **lswx**, **stswi**, **stswx**, **mfapidi**, **mfdcrx**, **mtdcrx**, or on any Power ISA embedded category floating point instruction when MSR[FP] = 1. All other defined or allocated instructions that are not implemented by the e200 cause an illegal instruction program exception.

Table 7-19 lists register settings when a program interrupt is taken.

Table 7-19. Program Interrupt—Register Settings

Register	Setting Description		
SRR0	Set to the effective address of the excepting instruction.		
SRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 SPE 0 WE 0 CE — EE 0 PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0 PMM 0 RI —
ESR	Illegal: Privileged: Trap: Unimplemented:	PIL, [VLEMI]. All other bits cleared. PPR, [VLEMI]. All other bits cleared. PTR, [VLEMI]. All other bits cleared. PUO, [FP], [VLEMI]. All other bits cleared.	
MCSR	Unchanged		
DEAR	Unchanged		
Vector	IVPR[0–15] ǁ IVOR6[16–27] ǁ 0b0000		

7.6.8 Floating-Point Unavailable Interrupt (IVOR7)

The floating-point unavailable exception is implemented as defined in the Power ISA embedded category. A floating-point unavailable interrupt occurs when no higher priority exception exists, an attempt is made to execute a floating-point instruction (including floating-point load, store, or move instructions), and the floating-point available bit in the MSR is disabled (MSR[FP] = 0).

Table 7-20 lists register settings when a floating-point unavailable interrupt is taken.

Table 7-20. Floating-Point Unavailable Interrupt—Register Settings

Register	Setting Description		
SRR0	Set to the effective address of the excepting instruction.		
SRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 SPE 0 WE 0 CE — EE 0 PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0 PMM 0 RI —
ESR	Unchanged		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	IVPR[0–15] || IVOR7[16–27] || 0b0000		

7.6.9 System Call Interrupt (IVOR8)

A system call interrupt occurs when a system call (**sc, se_sc)** instruction is executed and no higher priority exception exists.

Exception extensions implemented in the e200 for the Power ISA VLE include modification of the system call interrupt definition to include updating the ESR.

Table 7-21 lists register settings when a system call interrupt is taken.

Table 7-21. System Call Interrupt—Register Settings

Register	Setting Description		
SRR0	Set to the effective address of the instruction following the **sc** instruction.		
SRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 SPE 0 WE 0 CE — EE 0 PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0 PMM 0 RI —
ESR	[VLEMI] All other bits cleared.		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	IVPR[0–15] || IVOR8[16–27] || 0b0000		

7.6.10 Auxiliary Processor Unavailable Interrupt (IVOR9)

An auxiliary processor unavailable exception is defined by the Power ISA embedded category to occur when an attempt is made to execute an auxiliary processor unit instruction which is implemented but configured as unavailable, and no higher priority exception condition exists.

The e200 does not utilize this interrupt.

7.6.11 Decrementer Interrupt (IVOR10)

The e200 implements the decrementer exception as described in the *EREF*. A decrementer interrupt occurs when no higher priority exception exists, a decrementer exception condition exists (TSR[DIS] = 1), and the interrupt is enabled (both TCR[DIE] and MSR[EE] = 1).

The timer status register (TSR) holds the decrementer interrupt bit set by the timer facility when an exception is detected. Software must clear this bit in the interrupt handler to avoid repeated decrementer interrupts.

Table 7-22 lists register settings when a decrementer interrupt is taken.

Table 7-22. Decrementer Interrupt—Register Settings

Register	Setting Description		
SRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present.		
SRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 SPE 0 WE 0 CE — EE 0 PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0 PMM 0 RI —
ESR	Unchanged		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	IVPR[0–15] || IVOR10[16–27] || 0b0000		

7.6.12 Fixed-Interval Timer Interrupt (IVOR11)

The e200 implements the fixed-interval timer (FIT) exception as described in the *EREF*. The triggering of the exception is caused by selected bits in the time base register changing from 0 to 1.

A fixed-interval timer interrupt occurs when no higher priority exception exists, a FIT exception exists (TSR[FIS] = 1), and the interrupt is enabled (both TCR[FIE] and MSR[EE] = 1).

The timer status register (TSR) holds the FIT interrupt bit set by the timer facility when an exception is detected. Software must clear this bit in the interrupt handler to avoid repeated FIT interrupts.

Table 7-23 lists register settings when a FIT interrupt is taken.

Table 7-23. Fixed-Interval Timer Interrupt—Register Settings

Register	Setting Description		
SRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present.		
SRR1	Set to the contents of the MSR at the time of the interrupt.		
MSR	UCLE 0 SPE 0 WE 0 CE — EE 0 PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0 PMM 0 RI —
ESR	Unchanged		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	IVPR[0–15] || IVOR11[16–27] || 0b0000		

7.6.13 Watchdog Timer Interrupt (IVOR12)

The e200 implements the watchdog timer (WDT) exception as described in the *EREF*. The triggering of the exception is caused by the first enabled watchdog time-out.

A watchdog timer interrupt occurs when no higher priority exception exists, a watchdog timer exception exists (TSR[WIS] = 1), and the interrupt is enabled (both TCR[WIE] and MSR[CE] = 1).

The timer status register (TSR) holds the watchdog interrupt bit set by the timer facility when an exception is detected. Software must clear this bit in the interrupt handler to avoid repeated watchdog interrupts.

Table 7-24 lists register settings when a watchdog timer interrupt is taken.

Table 7-24. Watchdog Timer Interrupt—Register Settings

Register	Setting Description		
CSRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present.		
CSRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 SPE 0 WE 0 CE 0 EE 0 PR 0	FP 0 ME — FE0 0 DE 0/—[1]	FE1 0 IS 0 DS 0 PMM 0 RI —
ESR	Unchanged		
MCSR	Unchanged		

Table 7-24. Watchdog Timer Interrupt—Register Settings (continued)

DEAR	Unchanged
Vector	IVPR[0–15] || IVOR12[16–27] || 0b0000

[1] DE is cleared when the debug unit is disabled. Clearing of DE is optionally supported by control in HID0 when the debug unit is enabled.

MSR[DE] is not automatically cleared by a watchdog timer interrupt, but can be configured to be cleared via the HID0 register (HID0[CICLRDE]). Refer to Section 2.4.11, “Hardware Implementation Dependent Register 0 (HID0).”

7.6.14 Data TLB Error Interrupt (IVOR13)

A data TLB error interrupt occurs when no higher priority exception exists and a data TLB error exception exists due to a data translation lookup miss in the TLB.

Table 7-25 lists register settings when a DTLB interrupt is taken.

Table 7-25. Data TLB Error Interrupt—Register Settings

Register	Setting Description		
SRR0	Set to the effective address of the excepting load/store instruction.		
SRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 SPE 0 WE 0 CE — EE 0 PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0 PMM 0 RI —
ESR	[ST], [SPE], [VLEMI]. All other bits cleared.		
MCSR	Unchanged		
DEAR	Set to the effective address of a byte of the load or store whose access caused the violation.		
Vector	IVPR[0–15] || IVOR13[16–27] || 0b0000		

7.6.15 Instruction TLB Error Interrupt (IVOR14)

An instruction TLB error interrupt occurs when no higher priority exception exists and an instruction TLB error exception exists due to an instruction translation lookup miss in the TLB.

Exception extensions implemented in the e200 for the Power ISA VLE involve extending the definition of the instruction TLB error interrupt to include updating the ESR.

Table 7-26 lists register settings when an ITLB interrupt is taken.

Table 7-26. Instruction TLB Error Interrupt—Register Settings

Register	Setting Description
SRR0	Set to the effective address of the excepting instruction.

Table 7-26. Instruction TLB Error Interrupt—Register Settings (continued)

SRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 SPE 0 WE 0 CE — EE 0 PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0 PMM 0 RI —
ESR	[MIF] All other bits cleared.		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	IVPR[0:15] || IVOR14[16:27] || 0b0000		

7.6.16 Debug Interrupt (IVOR15)

The e200 implements the debug interrupt as defined in Power ISA embedded category with the following changes:

- When the debug unit is enabled, debug is no longer a critical interrupt, but uses DSRR0 and DSRR1 for saving machine state on context switch.
- A return from debug interrupt instruction (**rfdi** or **se_rfdi**) is implemented to support the new machine state registers.
- A critical interrupt taken debug event is defined to allow critical interrupts to generate a debug event.
- A critical return debug event is defined to allow debug events to be generated for **rfci** and **se_rfci** instructions.

There are multiple sources that can signal a debug exception. A debug interrupt occurs when no higher priority exception exists, a debug exception exists in the debug status register, and debug interrupts are enabled (both DBCR0[IDM] = 1 (internal debug mode) and MSR[DE] = 1). Enabling debug events and other debug modes are discussed further in Chapter 13, “Debug Support.” With the debug unit enabled, (see Section 2.4.11, “Hardware Implementation Dependent Register 0 (HID0)”), the debug interrupt has its own set of machine state save/restore registers (DSRR0, DSRR1) to allow debugging of both critical and noncritical interrupt handlers. In addition, interrupts can be handled while in a debug software handler. External and critical interrupts are not automatically disabled when a debug interrupt occurs but can be configured to be cleared via the HID0 register (HID0[DCLREE, DCLRCE]). Refer to Section 2.4.11, “Hardware Implementation Dependent Register 0 (HID0).” When the debug unit is disabled, debug interrupts use the CSRR0 and CSRR1 registers to save machine state.

NOTE

For additional details regarding the following descriptions of debug exception types, refer to Section 13.2, “Software Debug Events and Exceptions.”

An instruction address compare (IAC) debug exception occurs when there is an instruction address match as defined by the debug control registers and instruction address compare events are enabled. This could either be a direct instruction address match or a selected set of instruction addresses. IAC has the highest interrupt priority of all instruction-based interrupts, even if the instruction itself may have encountered an instruction TLB error or instruction storage exception.

A branch taken (BRT) debug exception is signaled when a branch instruction is considered taken by the branch unit and branch taken events are enabled. The debug interrupt is taken when no higher priority exception is pending.

A data address compare (DAC) exception is signaled when there is a data access address match as defined by the debug control registers and data address compare events are enabled. This could either be a direct data address match or a selected set of data addresses, or a combination of data address and data value matching. The debug interrupt is taken when no higher priority exception is pending.

The e200 implementation provides IAC linked with DAC exceptions. This results in a DAC exception only if one or more IAC conditions are also met. See Chapter 13, “Debug Support,” for more details.

A trap (TRAP) debug exception occurs when a program trap exception is generated while trap events are enabled. If MSR[DE] is set, the debug exception has higher priority than the program exception in this case and will be taken instead of a trap type program interrupt. The debug interrupt is taken when no higher priority exception is pending. If MSR[DE] is cleared when a trap debug exception occurs, a trap exception type program interrupt will occur instead.

A return (RET) debug exception occurs when executing an **rfi** or **se_rfi** instruction and return debug events are enabled. Return debug exceptions are not generated for **rfci** or **se_rfci** instructions. If MSR[DE] = 1 at the time of the execution of the **rfi** or **se_rfi**, a debug interrupt occurs provided that no higher priority exception is enabled to cause an interrupt. CSRR0 (debug unit disabled) or DSRR0 (debug unit enabled) is set to the address of the **rfi** or **se_rfi** instruction. If MSR[DE] = 0 at the time of the execution of the **rfi** or **se_rfi**, a debug interrupt does not occur immediately, but the event is recorded by setting the DBSR[RET] and DBSR[IDE] status bits.

A critical return (CRET) debug exception occurs when executing an **rfci** or **se_rfci** instruction and critical return debug events are enabled. Critical return debug exceptions are only generated for **rfci** or **se_rfci** instructions. If MSR[DE] = 1 at the time of the execution of the **rfci** or **se_rfci**, a debug interrupt occurs provided that no higher priority exception is enabled to cause an interrupt. CSRR0 (debug unit disabled) or DSRR0 (debug unit enabled) is set to the address of the **rfci** or **se_rfci** instruction. If MSR[DE] = 0 at the time of the execution of the **rfci** or **se_rfci**, a debug interrupt does not occur immediately, but the event is recorded by setting the DBSR[CRET] and DBSR[IDE] status bits. Note that critical return debug events should not normally be enabled unless the debug unit is enabled to avoid corruption of CSRR0/1.

An instruction complete (ICMP) debug exception is signaled following execution and completion of an instruction while this event is enabled.

A **mtmsr** or **mtdbcr0** that causes both MSR[DE] and DBCR0[IDM] to end up set, enabling precise debug mode, may cause an imprecise (delayed) debug exception to be generated due to an earlier recorded event in the debug status register.

An interrupt taken (IRPT) debug exception occurs when a noncritical interrupt context switch is detected. This exception is imprecise and unordered with respect to the program flow. Note that an IRPT debug interrupt only occurs when detecting a noncritical interrupt on the e200. The value saved in CSRR0/DSRR0 is the address of the noncritical interrupt handler.

A critical interrupt taken (CIRPT) debug exception occurs when a critical interrupt context switch is detected. This exception is imprecise and unordered with respect to the program flow. Note that a CIRPT debug interrupt only occurs when detecting a critical interrupt on the e200. The value saved in CSRR0/DSRR0 is the address of the critical interrupt handler. Note that critical interrupt taken debug events should not normally be enabled unless the debug unit is enabled to avoid corruption of CSRR0/1.

An unconditional debug event (UDE) exception occurs when the unconditional debug event pin (*p_ude*) transitions to the asserted state.

Debug counter debug exceptions occur when enabled and one of the debug counters decrements to zero.

External debug exceptions occur when enabled and one of the external debug event pins (*p_devt1*, *p_devt2*) transitions to the asserted state.

The debug status register (DBSR) provides a syndrome to differentiate between debug exceptions that can generate the same interrupt. For more details see Chapter 13, “Debug Support.”

Table 7-27 lists register settings when a debug interrupt is taken.

Table 7-27. Debug Interrupt—Register Settings

<table>
<tr><th>Register</th><th colspan="3">Setting Description</th></tr>
<tr><td>CSRR0/
DSRR0[1]</td><td colspan="3">Set to the effective address of the excepting instruction for IAC, BRT, RET, CRET, and TRAP.
Set to the effective address of the next instruction to be executed following the excepting instruction for DAC and ICMP.
For a UDE, IRPT, CIRPT, DCNT, or DEVT type exception, set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present.</td></tr>
<tr><td>CSRR1/
DSRR1</td><td colspan="3">Set to the contents of the MSR at the time of the interrupt</td></tr>
<tr><td>MSR</td><td>UCLE 0
SPE 0
WE 0
CE —/0[2]
EE —/0[2]
PR 0</td><td>FP 0
ME —
FE0 0
DE 0</td><td>FE1 0
IS 0
DS 0
PMM 0
RI —</td></tr>
<tr><td>DBSR[3]</td><td>Unconditional debug event:
Instr. complete debug event:
Branch taken debug event:
Interrupt taken debug event:
Critical interrupt taken debug event:
Trap instruction debug event:
Instruction address compare:
Data address compare:
Return debug event:
Critical return debug event:
Debug counter event:
External debug event:
and optionally, an
Imprecise debug event flag</td><td colspan="2">UDE
ICMP
BRT
IRPT
CIRPT
TRAP
{IAC1, IAC2, IAC3, IAC4}
{DAC1R, DAC1W, DAC2R, DAC2W}
RET
CRET
{DCNT1, DCNT2}
{DEVT1, DEVT2}

{IDE}</td></tr>
<tr><td>ESR</td><td colspan="3">Unchanged</td></tr>
<tr><td>MCSR</td><td colspan="3">Unchanged</td></tr>
</table>

Table 7-27. Debug Interrupt—Register Settings (continued)

DEAR	Unchanged
Vector	IVPR[0–15] || IVOR15[16–27] || 0b0000

[1] Assumes that the debug interrupt is precise.

[2] Conditional based on control bits in HID0.

[3] Note that multiple DBSR bits may be set.

7.6.17 System Reset Interrupt

The e200 implements the system reset interrupt as defined in the Power ISA embedded category. The system reset exception is a nonmaskable, asynchronous exception signaled to the processor through the assertion of system-defined signals.

A system reset may be initiated by either asserting the *p_reset_b* input signal or during power-on reset by asserting *m_por*. The *m_por* signal must be asserted during power up and must remain asserted for a period that allows internal logic to be reset. The *p_reset_b* signal must also remain asserted for a period that allows internal logic to be reset. This period is specified in the hardware specifications. If *m_por* or *p_reset_b* are asserted for less than the required interval, the results are not predictable.

When a reset request occurs, the processor branches to the system reset exception vector (value on *p_rstbase*[0:29] concatenated with 0b00) without attempting to reach a recoverable state. If reset occurs during normal operation, all operations cease and the machine state is lost. CPU internal state after a reset is defined in Section 2.6, “Reset Settings.”

Reset may also be initiated by watchdog timer or debug reset control. The watchdog timer and debug reset control provide the capability to assert the *p_wrs*[0:1] and *p_dbrstc*[0:1] signals. External logic may factor this into the *p_reset_b* input signal to cause an e200 reset to occur.

Table 7-28 shows the TSR register bits associated with watchdog timer reset status. Note that these bits are cleared when a processor reset occurs; thus if the *p_wrs*[0:1] outputs are factored into *p_reset_b*, they are only seen in the 00 state by software.

Table 7-28. TSR Watchdog Timer Reset Status

Bits	Name	Function
2–3 (34–35)	WRS	00 No action performed by watchdog timer 01 Watchdog Timer second time-out caused *p_wrs1* to be asserted 10 Watchdog Timer second time-out caused *p_wrs0* to be asserted 11 Watchdog Timer second time-out caused *p_wrs0* and *p_wrs1* to be asserted

Table 7-29 shows the DBSR register bits associated with reset status.

Table 7-29. DBSR Most Recent Reset

Bits	Name	Function
2–3 (34–35)	MRR	00 No reset occurred since these bits were last cleared by software 01 A reset occurred since these bits were last cleared by software 10 Reserved 11 Reserved

Table 7-30 lists register settings when a system reset interrupt is taken.

Table 7-30. System Reset Interrupt—Register Settings

Register	Setting Description		
CSRR0	Undefined		
CSRR1	Undefined		
MSR	UCLE 0 SPE 0 WE 0 CE 0 EE 0 PR 0	FP 0 ME 0 FE0 0 DE 0	FE1 0 IS 0 DS 0 PMM 0 RI 0
ESR	Cleared		
DEAR	Undefined		
Vector	[*p_rstbase*[0:29]] || 0b00		

7.6.18 SPE/EFPU Unavailable Interrupt (IVOR32)

The SPE unit unavailable exception is taken if MSR[SPE] is cleared and execution of a SPE or EFPU instruction other than the scalar floating-point instructions (**efs**$_{\mathbf{xxx}}$) or **brinc** is attempted. When the SPE/EFPU unavailable exception occurs, the processor suppresses execution of the instruction causing the exception. Table 7-31 lists register settings when a SPE/EFPU unavailable interrupt is taken.

Table 7-31. SPE/EFPU Unavailable Interrupt—Register Settings

Register	Setting Description		
SRR0	Set to the effective address of the excepting SPE/EFPU instruction.		
SRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 SPE 0 WE 0 CE — EE 0 PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0 PMM 0 RI —
ESR	SPE, [VLEMI]. All other bits cleared.		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	IVPR[0–15] || IVOR32[16–27] || 0b0000		

7.6.19 Embedded Floating-Point Data Interrupt (IVOR33)

The embedded floating-point data interrupt is taken if no higher priority exception exists and an EFPU floating-point data exception is generated. When a floating-point data exception occurs, the processor suppresses execution of the instruction causing the exception.

Table 7-32 lists register settings when an EFPU floating-point data interrupt is taken.

Table 7-32. Embedded Floating-Point Data Interrupt—Register Settings

Register	Setting Description		
SRR0	Set to the effective address of the excepting EFPU instruction.		
SRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 SPE 0 WE 0 CE — EE 0 PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0 PMM 0 RI —
ESR	SPE, [VLEMI]. All other bits cleared.		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	IVPR[0–15] || IVOR33[16–27] || 0b0000		

7.6.20 Embedded Floating-Point Round Interrupt (IVOR34)

The embedded floating-point round interrupt is taken when an EFPU floating-point instruction generates an inexact result and inexact exceptions are enabled.

Table 7-33 lists register settings when an EFPU floating-point round interrupt is taken.

Table 7-33. Embedded Floating-point Round Interrupt—Register Settings

Register	Setting Description		
SRR0	Set to the effective address of the instruction following the excepting EFPU instruction.		
SRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 SPE 0 WE 0 CE — EE 0 PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0 PMM 0 RI —
ESR	SPE, [VLEMI]. All other bits cleared.		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	IVPR[0–15] || IVOR34[16–27] || 0b0000		

7.6.21 Performance Monitor Interrupt (IVOR35)

The e200z7 provides a performance monitor interrupt that may be generated by an enabled condition or event. An enabled condition or event is as follows:

A PMC*x* register overflow condition occurs with the following settings:

- PMLCa*x*[CE] = 1; that is, for the given counter the overflow condition is enabled.
- PMC*x*[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.

Although an exception condition may occur with MSR[EE] = 0, the interrupt cannot be taken until MSR[EE] = 1.

The priority of the performance monitor interrupt is below all other asynchronous interrupts. For details, see Section 7.6.21, “Performance Monitor Interrupt (IVOR35).”

Table 7-34 lists register settings when an performance monitor interrupt is taken.

Table 7-34. Performance Monitor Interrupt—Register Settings

Register	Setting Description		
SRR0	Set to the effective address of the next instruction to be executed.		
SRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 SPE 0 WE 0 CE — EE 0 PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0 PMM 0 RI —
ESR	Unchanged		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	IVPR[0–15] || IVOR35[16–27] || 0b0000		

7.7 Exception Recognition and Priorities

The following list of exception categories describes how the e200 handles exceptions up to the point of signaling the appropriate interrupt to occur. Also, instruction completion is defined as updating all architectural registers associated with that instruction as necessary, and then removing the instruction from the pipeline.

- Interrupts caused by asynchronous events (exceptions). These exceptions are further distinguished by whether they are maskable and recoverable.
  - Asynchronous, nonmaskable, nonrecoverable:
    - System reset by assertion of *p_reset_b*

    - Has highest priority and is taken immediately regardless of other pending exceptions or recoverability. (Includes watchdog timer reset control and debug reset control.)
  - Asynchronous, nonmaskable, possibly nonrecoverable:
    - Nonmaskable interrupt by assertion of **p_nmi_b**
    - Has priority over any other pending exception except system reset conditions. Recoverability is dependent on whether MCSRR0/1 are holding essential state info and are overwritten when the NMI occurs.
  - Asynchronous, maskable/nonmaskable, recoverable/nonrecoverable:
    - Machine check interrupt
    - Has priority over any other pending exception except system reset conditions. Recoverability is dependent on the source of the exception.
  - Asynchronous, maskable, recoverable:
    - External input, fixed-interval timer, decrementer, critical input, performance monitor, unconditional debug, external debug event, debug counter event, and watchdog timer interrupts
    - Before handling this type of exception, the processor needs to reach a recoverable state. A maskable recoverable exception will remain pending until taken or canceled by software.
- Synchronous, non-instruction based interrupts. The only exception is this category is the interrupt taken debug exception, recognized by an interrupt taken event. It is not considered instruction-based but is synchronous with respect to the program flow.
  - Synchronous, maskable, recoverable:
    - Interrupt taken debug event
    - The machine will be in a recoverable state due to the state of the machine at the context switch triggering this event.
- Instruction-based interrupts. These interrupts are further organized by the point in instruction processing in which they generate an exception.
  - Instruction fetch:
    - Instruction storage, instruction TLB, and instruction address compare debug exceptions.
    - Once these types of exceptions are detected, the excepting instruction is tagged. When the excepting instruction is next to begin execution and a recoverable state has been reached, the interrupt is taken. If an event prior to the excepting instruction causes a redirection of execution, the instruction fetch exception is discarded (but may be encountered again).
  - Instruction dispatch/execution:
    - Program, system call, data storage, alignment, floating-point unavailable, SPE/EFPU unavailable, data tlb, embedded floating-point data, embedded floating-point round, debug (trap, branch taken, ret) interrupts.
    - These types of exceptions are determined during decode or execution of an instruction. The exception remains pending until all instructions before the exception causing instruction in program order complete. The interrupt is then taken without completing the exception-causing instruction. If completing previous instructions causes an exception, that

exception takes priority over the pending instruction dispatch/execution exception, which is discarded (but may be encountered again when instruction processing resumes).

- Post-instruction execution:
  - Debug (data address compare, instruction complete) interrupt.
  - These debug exceptions are generated following execution and completion of an instruction while the event is enabled. If executing the instruction produces conditions for another type of exception with higher priority, that exception is taken and the post-instruction exception is discarded for the instruction (but may be encountered again when instruction processing resumes)

7.7.1 Exception Priorities

Exceptions are prioritized as described in Table 7-35. Some exceptions may be masked or imprecise, which affects their priority. Nonmaskable exceptions such as reset and machine check may occur at any time and are not delayed even if an interrupt is being serviced; thus state information for any interrupt may be lost. Reset and certain machine checks are nonrecoverable.

Table 7-35. Zen Exception Priorities

Priority	Exception	Cause	IVOR
Asynchronous Exceptions			
0	System reset	Assertion of **p_reset_b**, Watchdog Timer Reset Control, or Debug Reset Control	None
1	Machine check	Assertion of **p_mcp_b**, assertion of **p_nmi_b**, Cache Parity errors, exception on fetch of first instruction of an interrupt handler, external bus errors	1
2	—	—	—
3[1]	Debug: 1. UDE 2. DEVT1 3. DEVT2 4. DCNT1 5. DCNT2 6. IDE	1. Assertion of **p_ude** (Unconditional Debug Event) 2. Assertion of **p_devt1** and event enabled (External Debug Event 1) 3. Assertion of **p_devt2** and event enabled (External Debug Event 2) 4. Debug Counter 1 exception 5. Debug Counter 2 exception 6. Imprecise Debug Event (event imprecise due to previous higher priority interrupt	15
4[1]	Critical Input	Assertion of **p_critint_b**	0
5[1]	Watchdog Timer	Watchdog Timer first enabled time-out	12
6[1]	External Input	Assertion of **p_extint_b**	4
7[1]	Fixed-Interval Timer	Posting of a FIT exception in TSR due to programmer-specified bit transition in the Time Base register	11
8[1]	Decrementer	Posting of a Decrementer exception in TSR due to programmer-specified Decrementer condition	10
9[1]	Performance Monitor	Performance Monitor Enabled Condition or Event	35

Table 7-35. Zen Exception Priorities (continued)

Priority	Exception	Cause	IVOR
Instruction Fetch Exceptions			
10	Debug: IAC (unlinked)	Instruction address compare match for enabled IAC debug event and DBCR0[IDM] asserted	15
11	ITLB Error	Instruction translation lookup miss in the TLB	14
12	Instruction Storage	Access control. Byte ordering due to misaligned instruction across page boundary to pages with mismatched VLE bits, or access to page with VLE set, and E indicating little-endian. Misaligned Instruction fetch due to a change of flow to an odd half word instruction boundary on a Power ISA (non-VLE) instruction page, due to value in LR, CTR, or xSRR0	3
Instruction Dispatch/Execution Interrupts			
13	Program: Illegal	Attempted execution of an illegal instruction.	6
14	Program: Privileged	Attempted execution of a privileged instruction in user-mode	6
15	Floating-point Unavailable	Any floating-point unavailable exception condition.	7
	SPE/EFPU Unavailable	Any SPE or EFPU unavailable exception condition.	32
16	Program: Unimplemented	Attempted execution of an unimplemented instruction.	6
17	Debug: 1. BRT 2. Trap 3. RET 4. CRET	1. Attempted execution of a taken branch instruction 2. Condition specified in **tw** or **twi** instruction met. 3. Attempted execution of a **rfi** instruction. 4. Attempted execution of an **rfci** instruction. **Note**: Exceptions requires corresponding debug event enabled, MSR[DE] = 1, and DBCR0[IDM] = 1.	15
18	Program: Trap	Condition specified in **tw** or **twi** instruction met and not trap debug.	6
	System Call	Execution of the System Call (**sc, se_sc**) instruction.	8
	EFPU Floating-point Data	Denormalized, NaN, or Infinity data detected as input or output, or underflow, overflow, divide by zero, or invalid operation in the EFPU.	33
	EFPU Round	Inexact Result	34
19	Alignment	**lmw**, **stmw, lwarx,** or **stwcx.** not word aligned. **lharx,** or **sthcx.** not half-word aligned. **dcbz** with cache disabled.	5

Table 7-35. Zen Exception Priorities (continued)

Priority	Exception	Cause	IVOR
20	Debug: Debug with concurrent DTLB or DSI exception, or concurrent async machine check: 1. DAC/IAC linked[2] 2. DAC unlinked[2]	Debug with concurrent DTLB or DSI exception, or async machine check condition on the DAC. DBSR[IDE] also set. 1. Data Address Compare linked with Instruction Address Compare 2. Data Address Compare unlinked **Note**: Exceptions requires corresponding debug event enabled, MSR[DE] = 1, and DBCR0[IDM] = 1. In this case, the debug exception is considered imprecise, and DBSR[IDE] will be set. Saved PC will point to the load or store instruction causing the DAC event.	15
21	Data TLB Error	Data translation lookup miss in the TLB.	13
22	Data Storage	Access control. Byte ordering due to misaligned access across page boundary to pages with mismatched E bits. Cache locking due to attempt to execute a **dcbtls**, **dcbtstls**, **dcblc**, **icbtls**, or **icblc** in user mode with MSR[UCLE] = 0.	2
23	Alignment	**dcbz** to W = 1 or I = 1 storage with cache enabled	5
24	Debug: 1. IRPT 2. CIRPT	1. Interrupt taken (non-critical) 2. Critical Interrupt taken (critical only) **Note**: Exceptions requires corresponding debug event enabled, MSR[DE] = 1 and DBCR0[IDM] = 1.	15
Post-Instruction Execution Exceptions			
25	Debug: 1. DAC/IAC linked[2] 2. DAC unlinked[2]	1. Data Address Compare linked with Instruction Address Compare 2. Data Address Compare unlinked **Notes**: Exceptions requires corresponding debug event enabled, MSR[DE] = 1, and DBCR0[IDM] = 1. Saved PC will point to the instruction following the load or store instruction causing the DAC event.	15
26	Debug: 1. ICMP	1. Completion of an instruction. **Note**: Exceptions requires corresponding debug event enabled, MSR[DE] = 1, and DBCR0[IDM] = 1.	15

[1] These asynchronous exceptions are sampled at instruction boundaries, thus may actually occur after exceptions which are due to a currently executing instruction. If one of these exceptions occurs during execution of an instruction in the pipeline, it is not processed until the pipeline has been flushed, and the exception associated with the excepting instruction may occur first.

[2] When no Data Storage Interrupt or Data TLB Error occurs, the Zen implements the data address compare debug exceptions as post-instruction exceptions which differs from the Power ISA definition. When a TEA (either a DTLB error or DSI or Machine Check (external TEA)) occurs in conjunction with an enabled DAC or linked DAC/IAC on a load or store class instruction, or a debug counter event based on a counted DAC, the debug Interrupt takes priority, and the saved PC value will point to the load or store class instruction, rather than to the next instruction.

7.8 Interrupt Processing

When an interrupt is taken, the processor uses the following:

- SRR0/SRR1 for noncritical interrupts
- CSRR0/CSRR1 for critical interrupts
- MCSRR0/MCSRR1 for machine check interrupts
- Either CSRR0/CSRR1 or DSRR0/DSRR1 for debug interrupts to save the contents of the MSR and to assist in identifying where instruction execution should resume after the interrupt is handled

When an interrupt occurs, one of SRR0/CSRR0/DSRR0/MCSRR0 is set to the address of the instruction that caused the exception or to the following instruction as appropriate.

- SRR1 is used to save machine state (selected MSR bits) on noncritical interrupts and to restore those values when an **rfi** instruction is executed.
- CSRR1 is used to save machine status (selected MSR bits) on critical interrupts and to restore those values when an **rfci** instruction is executed.
- DSRR1 is used to save machine status (selected MSR bits) on debug interrupts when the debug unit is enabled and to restore those values when an **rfdi** instruction is executed.
- MCSRR1 is used to save machine status (selected MSR bits) on machine check interrupts and to restore those values when an **rfmci** instruction is executed.

The exception syndrome register is loaded with information specific to the exception type. Some interrupt types can only be caused by a single exception type, and thus do not use an ESR setting to indicate the interrupt cause.

The machine state register is updated to preclude unrecoverable interrupts from occurring during the initial portion of the interrupt handler. Specific settings are described in Table 7-36.

- For alignment, data storage, or data TLB miss interrupts, the data exception address register (DEAR) is loaded with the address which caused the interrupt to occur.
- For machine check interrupts, the machine check syndrome register is loaded with information specific to the exception type. For certain machine checks, the MCAR is loaded with an address corresponding to the machine check.

Instruction fetch and execution resumes, using the new MSR value, at a location specific to the exception type. The location is determined by the interrupt vector prefix register (IVPR), and an interrupt vector offset register (IVOR) specific for each type of interrupt (see Table 7-2).

Table 7-36 shows the MSR settings for different interrupt categories.

Table 7-36. MSR Setting Due to Interrupt

Bits	MSR Definition	Reset Setting	Noncritical Interrupt	Critical Interrupt	Debug Interrupt	Machine Check Interrupt
5 (37)	UCLE	0	0	0	0	0
6 (38)	SPE	0	0	0	0	0
13 (45)	WE	0	0	0	0	0

Table 7-36. MSR Setting Due to Interrupt (continued)

Bits	MSR Definition	Reset Setting	Noncritical Interrupt	Critical Interrupt	Debug Interrupt	Machine Check Interrupt
14 (46)	CE	0	—	0	—/0[1]	0
16 (48)	EE	0	0	0	—/0[1]	0
17 (49)	PR	0	0	0	0	0
18 (50)	FP	0	0	0	0	0
19 (51)	ME	0	—	—	—	0
20 (52)	FE0	0	0	0	0	0
22 (54)	DE	0	—	—/0[1]	0	—/0[1]
23 (55)	FE1	0	0	0	0	0
26 (58)	IS	0	0	0	0	0
27 (59)	DS	0	0	0	0	0
29 (61)	PMM	0	0	0	0	0
30 (62)	RI	0	—	—	—	0

Reserved and preserved bits are unimplemented and read as 0.

[1] Conditionally cleared based on control bits in HID0.

7.8.1 Enabling and Disabling Exceptions

When a condition exists that may cause an exception to be generated, determine whether the exception is enabled for that condition according to the following.

- System reset exceptions cannot be masked.
- Machine check exceptions cannot be masked from sources other than the machine check pin, and certain other async machine check status settings. Assertion of *p_mcp_b* is only recognized if the machine check pin enable bit (HID0[EMCP]) is set. Certain machine check exceptions can be enabled and disabled through bit(s) in the HID0 register.
- Asynchronous, maskable noncritical exceptions (such as the external input and decrementer) are enabled by setting MSR[EE]. When MSR[EE] = 0, recognition of these exception conditions is delayed. MSR[EE] is cleared automatically when a noncritical or critical interrupt is taken to mask further recognition of conditions causing those exceptions.
- Asynchronous, maskable critical exceptions (such as critical input and watchdog timer) are enabled by setting MSR[CE]. When MSR[CE] = 0, recognition of these exception conditions is delayed. MSR[CE] is cleared automatically when a critical interrupt is taken to mask further recognition of conditions causing those exceptions.
- Synchronous and asynchronous debug exceptions are enabled by setting MSR[DE]. When MSR[DE] = 0, recognition of these exception conditions is masked. MSR[DE] is cleared automatically when a debug interrupt is taken to mask further recognition of conditions causing those exceptions. See Chapter 13, “Debug Support,” for more details on individual control of debug exceptions.

- The floating-point unavailable exception can be prevented by setting MSR[FP] (although an unimplemented instruction exception will be generated by e200 instead).

7.8.2 Returning from an Interrupt Handler

The return from interrupt (**rfi, se_rfi**), return from critical interrupt (**rfci, se_rfci**), return from debug interrupt (**rfdi, se_rfdi**), and return from machine check interrupt (**rfmci, se_rfmci**) instructions perform context synchronization by allowing previously-issued instructions to complete before returning to the interrupted process. In general, execution of return from interrupt type instructions ensures the following:

- All previous instructions have completed to a point where they can no longer cause an exception. This includes post-execute type exceptions.
- Previous instructions complete execution in the context (privilege and protection) under which they were issued.
- The **rfi** and **se_rfi** instructions copy SRR1 bits back into the MSR.
- The **rfci** and **se_rfci** instructions copy CSRR1 bits back into the MSR.
- The **rfdi** and **se_rfdi** instructions copy DSRR1 bits back into the MSR.
- The **rfmci** and **se_rfmci** instructions copy MCSRR1 bits back into the MSR.
- Instructions fetched after this instruction execute in the context established by this instruction.
- Program execution resumes at the instruction indicated by SRR0 for **rfi** and **se_rfi**, CSRR0 for **rfci** and **se_rfci**, MCCSRR0 for **rfmci** and **se_rfmci**, and DSRR0 for **rfdi** and **se_rfdi**.

Note that the return instructions **rfi** and **se_rfi** may be subject to a return type debug exception and that the return from critical interrupt instructions **rfci** and **se_rfci** may be subject to a critical return type debug exception. For a complete description of context synchronization, refer to the *EREF*.

7.9 Process Switching

The following instructions are useful for restoring proper context during process switching:

- The **msync** instruction orders the effects of data memory instruction execution. All instructions previously initiated appear to have completed before the **msync** instruction completes, and no subsequent instructions appear to be initiated until the **msync** instruction completes.
- The **isync** instruction waits for all previous instructions to complete and then discards any fetched instructions, causing subsequent instructions to be fetched (or refetched) from memory and to execute in the context (privilege, translation, and protection) established by the previous instructions.
- The **stwcx.** instructions clears any outstanding reservations, ensuring that a load and reserve instruction in an old process is not paired with a store conditional instruction in a new one.

Chapter 8
Performance Monitor

This chapter describes the performance monitor, which is generally defined by the Freescale EIS and implemented as a unit on the e200z7 core. Although the programming model is defined by the EIS, some features are defined by the implementation—in particular, the events that can be counted.

8.1 Overview

The performance monitor provides the ability to count predefined events and processor clocks associated with particular operations, such as 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 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 consists of 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 **p_*pm_event***.
- The external outputs **p_pmc0_ov**, **p_pmc1_ov**, **p_pmc2_ov**, and **p_pmc3_ov**
- PMRs, as follow:
  - The performance monitor counter registers PMC0–PMC3 are 32-bit counters used to count software-selectable events. UPMC0–UPMC3 provide user-level read access to these registers. Counted events are those that should be of general value. They are identified in Table 8-10.
  - 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 and PMLCb0–PMLCb3 control individual performance monitor counters. Each counter has a corresponding PMLCa and PMLCb register. UPMLCa0–UPMLCa3 and UPMLCb0–UPMLCb3 provide user-level read access to PMLCa0–PMLCa3 and PMLCb0–PMLCb3.

- The performance monitor interrupt follows the embedded category in the Power ISA interrupt model and is assigned to interrupt vector offset register 35 (IVOR35). It has the lowest priority of all asynchronous interrupts.

Software communication with the performance monitor APU is achieved through PMRs rather than SPRs.

8.2 Performance Monitor Instructions

The performance monitor defines the **mfpmr** and **mtpmr** instructions for reading and writing the PMRs as follows.

mfpmr mfpmr

Move from Performance Monitor Register

mfpmr **r**D,**PMRN** Form: X

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	rD	**PMRN**[5–9]	**PMRN**[0–4]	0 1 0 1 0 0 1 1 1 0	/

```
GPR(rD) ← PMREG(PMRN)
```

The contents of the performance monitor register designated by PMRN are placed into GPR[**r**D].

MSR[PR] has the following results:

- When MSR[PR] = 1, specifying a performance monitor register that is not implemented or is write only and is not privileged (i.e. PMRN[5] = 0) results in an illegal instruction exception-type Program Interrupt.
- When MSR[PR] = 1, specifying a performance monitor register that is not implemented or is write only and is privileged (i.e. PMRN[5] = 1) results in a privileged instruction exception-type Program Interrupt.
- When MSR[PR] = 0, specifying a performance monitor register that is not implemented or is write-only results in an illegal instruction exception type Program Interrupt.

mtpmr mtpmr

Move to Performance Monitor Register

mtpmr **PMRN, r**S Form: X

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	rS	**PMRN**$_{5:9}$	**PMRN**$_{0:4}$	0 1 1 1 0 0 1 1 1 0	/

```
PMREG(PMRN) ← GPR(rS)
```

The contents of GPR[**r**S] are placed into the performance monitor register designated by PMRN.

MSR[PR] has the following results:

- When MSR[PR] = 1, specifying a performance monitor register that is not implemented or is read-only and is not privileged (i.e. PMRN[5] = 0) results in an illegal instruction exception-type Program Interrupt.
- When MSR[PR] = 1, specifying a performance monitor register that is not implemented or is read-only and is privileged (i.e. PMRN[5] = 1) results in a privileged instruction exception-type Program Interrupt.
- When MSR[PR] = 0, specifying a performance monitor register that is not implemented or is read-only results in an illegal instruction exception type Program Interrupt.

8.3 Performance Monitor Registers

The Freescale EIS defines a set of register resources used exclusively by the performance monitor. PMRs are similar to the SPRs defined in the embedded category in the Power ISA architecture and are accessed by **mtpmr** and **mfpmr** instructions, which are also defined by the Freescale EIS. Table 8-1 lists supervisor-level (privileged) PMRs.

Table 8-1. Supervisor-Level PMRs (PMR[5] = 1)

Name	Register Name	PMR Number	pmr[0–4]	pmr[5–9]	Section/ Page
PMC0	Performance monitor counter 0	16	00000	10000	8.3.9/8-12
PMC1	Performance monitor counter 1	17	00000	10001	
PMC2	Performance monitor counter 2	18	00000	10010	
PMC3	Performance monitor counter 3	19	00000	10011	
PMGC0	Performance monitor global control register 0	400	01100	10000	8.3.3/8-5
PMLCa0	Performance monitor local control a0	144	00100	10000	8.3.5/8-6
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	8.3.7/8-7
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	

Table 8-2 shows the user-level PMRs, which are read-only and accessed with **mfpmr**.

Table 8-2. User-Level PMRs (PMR[5] = 0) (Read-Only)

Name	Register Name	PMR Number	pmr[0–4]	pmr[5–9]	Section/ Page
UPMC0	User performance monitor counter 0	0	00000	00000	8.3.10/8-13
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	
UPMGC0	User performance monitor global control register 0	384	01100	00000	8.3.4/8-6
UPMLCa0	User performance monitor local control a0	128	00100	00000	8.3.6/8-7
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	8.3.8/8-12
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	

8.3.1 Invalid PMR References

Behavior when an invalid PMR is referenced depends on the privilege level of the register and MSR[PR]. Table 8-3 shows the response for various references to invalid PMRs.

Table 8-3. Response to an Invalid PMR Reference

PMR Address Bit 5	MSR[PR]	Response
0 (user)	x	Illegal exception
1 (supervisor)	0 (supervisor)	Illegal exception
	1 (user)	Privileged exception

8.3.2 References to Read-only PMRs

If a **mtpmr** instruction is executed to a read-only PMR, the e200z7 takes an illegal exception.

8.3.3 Global Control Register 0 (PMGC0)

The performance monitor global control register (PMGC0), shown in Figure 8-1, controls all performance monitor counters.

PMR 400 Access: Read/Write

	0	1	2	3 … 18	19 20	21 22	23	24 … 31
R W	FAC	PMIE	FCECE	—	TBSEL	—	TBEE	—
Reset	All zeros							

Figure 8-1. Performance Monitor Global Control Register (PMGC0)

PMGC0 is cleared by reset. Reading this register does not change its contents. Table 8-4 describes PMGC0's fields.

Table 8-4. PMGC0 Field Descriptions

Bits	Name	Description
0 (32)	FAC	Freeze All Counters When FAC is set by hardware or software, it has no effect on PMLCa*x*[FC]; PMLCa*x*[FC] maintains it's current value until changed by software. FAC setting by hardware is controlled by PMGC0[FCECE]. 0 The PMCs are incremented (if permitted by other PMGC/PMLC control bits). 1 The PMCs are not incremented.
1 (33)	PMIE	Performance monitor interrupt Enable Software can clear PMIE to prevent performance monitor interrupts. Performance monitor interrupts are caused by time base events or PMCx overflow. 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
2 (34)	FCECE	Freeze Counters on Enabled Condition or Event 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 PMGC0[TBSEL] transitions to 1 and PMGC0[TBEE] = 1. The use of the trigger and freeze counter conditions depends on the enabled conditions and events described in Section 8.4, "Performance Monitor Interrupt." 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 to 1. It is up to software to clear PMGC0[FAC] to 0.
3–18 (35–50)	—	Reserved, should be cleared.
19–20 (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). Time-base frequency is implementation dependent, so software should invoke a system service program to obtain the frequency before choosing a TBSEL value. 00 TB[63] (TBL[31]) 01 TB[55] (TBL[23]) 10 TB[51] (TBL[19]) 11 TB[47] (TBL[15])

Table 8-4. PMGC0 Field Descriptions (continued)

Bits	Name	Description
21–22 (53–54)	—	Reserved, should be cleared.
23 (55)	TBEE	Time base transition Event Enable Time base transition events can be used to freeze counters (PMGC0[FCECE]) or signal an exception (PMGC0[PMIE]). Although the exception signal condition may occur with MSR[EE] = 0, the interrupt cannot be taken until MSR[EE] = 1. 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. 0 Time base transition events are disabled. 1 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.
24–31 (56–63)	—	Reserved, should be cleared.

8.3.4 User Global Control Register 0 (UPMGC0)

UPMGC0 provides user-level read access to PMGC0. UPMGC0 can be read by user-level software with the **mfpmr** instruction using PMR 384.

8.3.5 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 is used in conjunction with the corresponding PMLCb register. PMLCa registers are shown in Figure 8-2.

PMR 144–147 Access: Read/Write

	0	1	2	3	4	5	6 7	8 15	16	17 18 19	20 31
R W	FC	FCS	FCU	FCM1	FCM0	CE	—	EVENT	—	PMP	—
Reset	All zeros										

Figure 8-2. Performance Monitor Local Control A Registers (PMLCa0–PMLCa3)

PMLCa registers are cleared by reset. Table 8-5 describes PMLCa fields.

Table 8-5. PMLCa0–PMLCa3 Field Descriptions

Bits	Name	Description
0 (32)	FC	Freeze Counter. 0 The PMC can be incremented (if enabled by other performance monitor control fields). 1 The PMC will not be incremented.
1 (33)	FCS	Freeze Counter in Supervisor state. 0 The PMC can be incremented (if enabled by other performance monitor control fields). 1 The PMC will not be incremented if MSR[PR] is cleared.

Table 8-5. PMLCa0–PMLCa3 Field Descriptions (continued)

Bits	Name	Description
2 (34)	FCU	Freeze Counter in User state. 0 The PMC can be incremented (if enabled by other performance monitor control fields). 1 The PMC will not be incremented if MSR[PR] is set.
3 (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 will not be incremented if MSR[PMM] is set.
4 (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 will not be incremented if MSR[PMM] is cleared.
5 (37)	CE	Condition Enable. It is recommended that CE be cleared when counter PMC*n* is selected for chaining. 0 Overflow conditions for PMC*n* cannot occur (PMC*n* cannot cause interrupts or freeze counters) 1 An overflow condition is present when the most-significant-bit of PMC*n* is equal to 1.
6–7 (38–39)	—	Reserved for EVENT expansion, should be cleared.
8–15 (40–47)	EVENT	Event selector. See Section 8.7, “Event Selection”
16 (48)	—	Reserved, should be cleared.
17–19 (49–51)	PMP	Performance Monitor Watchpoint Periodicity Select 000 Performance Monitor Watchpoint *x* triggers on any change of counter$_x$ bit 32 (period=2^{31}) 001 Performance Monitor Watchpoint *x* triggers on any change of counter$_x$ bit 43 (period=2^{20}) 010 Performance Monitor Watchpoint *x* triggers on any change of counter$_x$ bit 49 (period=2^{14}) 011 Performance Monitor Watchpoint *x* triggers on any change of counter$_x$ bit 55 (period=2^{8}) 100 Performance Monitor Watchpoint *x* triggers on any change of counter$_x$ bit 59 (period=2^{4}) 101 Performance Monitor Watchpoint *x* triggers on any change of counter$_x$ bit 61 (period=2^{2}) 110 Performance Monitor Watchpoint *x* triggers on any change of counter$_x$ bit 62 (period=2^{1}) 111 Performance Monitor Watchpoint *x* triggers on any change of counter$_x$ bit 63 (period=2^{0})[1]
20–31 (52–63)	—	Reserved, should be cleared.

[1] For certain events which may count an even number of times per cycle, this watchpoint is not guaranteed to assert with PMP = 111.

8.3.6 User Local Control A Registers (UPMLCa0–UPMLCa3)

The PMLCa register contents are aliased to UPMLCa0–UPMLCa3, which can be read by user-level software with **mfpmr** using PMR numbers in Table 8-2.

8.3.7 Local Control B Registers (PMLCb0–PMLCb3)

Local control B registers (PMLCb0–PMLCb3), shown in Figure 8-3, specify triggering conditions, a threshold value, and a multiple to apply to a threshold event selected for the corresponding performance monitor counter. For the e200z7, thresholding is supported only for PMC0 and PMC1. PMLCb is used in conjunction with the corresponding PMLCa register.

PMR 272–275 Access: Read/Write

	0	1 3	4	7	8	9 12	13	14 15
R W	—	TRIGONCTL	—	TRIGOFFCTL	—	TRIGONSEL	—	TRIGOFFSEL
Reset	All zeros							

	16 17	18	19 20	21 22 23	24 25	26 31
R W	TRIGOFFSEL	TRIGGERED	—	THRESHMUL	—	THRESHOLD
Reset	All zeros					

Figure 8-3. Performance Monitor Local Control B Registers (PMLCb0–PMLCb3)

PMLCb is cleared by reset. Table 8-6 describes PMLCb fields.

Table 8-6. PMLCb0–PMLCb3 Field Descriptions

Bits	Name	Description
0 (32)	—	Reserved, should be cleared.
1:3 (33:35)	TRIGONCNTL	Trigger-on Control Class—Class of Trigger-on source Indicates the condition under which triggering to start counting occurs. No triggering will occur while PMGC0[FAC] or PMLCa*n*[FC] is set. 000 Trigger-on control is disabled if TRIGONSEL is 0000 (i.e. counting is not affected by triggers). All other values for TRIGONSEL are reserved. 001 Trigger-on control based on selected PMC condition(s) 010 Trigger-on based on selected processor event(s) 011 Trigger-on based on selected hardware signal(s) 100 Trigger-on based on selected watchpoint occurrence (watchpoint #0–15) 101 Trigger-on based on selected watchpoint occurrence (extension for watchpoint #16–31) 11x Reserved
4 (36)	—	Reserved, should be cleared.
5:7 (37:39)	TRIGOFFCNTL	Trigger-off Control Class—Class of Trigger-off source Indicates the condition under which triggering to stop counting occurs. No triggering will occur while PMGC0[FAC] or PMLCa*n*[FC] is set. 000 Trigger-off control is disabled if TRIGOFFSEL is 0000 (i.e. counting is not affected by triggers) All other values for TRIGOFFSEL are reserved. 001 Trigger-off control based on selected PMC condition(s) 010 Trigger-off based on selected processor event(s) 011 Trigger-off based on selected hardware signal(s) 100 Trigger-off based on selected watchpoint occurrence (watchpoint #0–15) 101 Trigger-off based on selected watchpoint occurrence (extension for watchpoint #16–31) 11x Reserved
8 (40)	—	Reserved, should be cleared.

Table 8-6. PMLCb0–PMLCb3 Field Descriptions (continued)

Bits	Name	Description
9:12 (41:44)	TRIGONSEL	Trigger-on Source Select—Source Select based on setting of TRIGONCTL • TRIGONCTL = 000: 0000 Trigger-on control is disabled 0001–1111 Reserved • TRIGONCTL = 001: This field should be to the ID of the PMCy that should trigger event counting to start. When PMCy overflows, the trigger will be generated. When TRIGONSEL = PMCx (i.e. self-select), no triggering will occur due to any counter change. If TRIGONSEL = TRIGOFFSEL, triggering results are undefined. 0000 Trigger-on when PMC0[OV] transitions to a 1. 0001 Trigger-on when PMC1[OV] transitions to a 1. 0010 Trigger-on when PMC2[OV] transitions to a 1. 0011 Trigger-on when PMC3[OV] transitions to a 1. 0100–1111 Reserved • TRIGONCTL = 010: 0000 Trigger-on when next processor interrupt occurs (software may want to set PMGC0[PMIE] = 0 for this setting). 0001–1111 Reserved • TRIGONCTL = 011: 0000 Trigger on assertion of **p_devnt_out[0]** 0001 Trigger on assertion of **p_devnt_out[1]** 0010 Trigger on assertion of **p_devnt_out[2]** 0011 Trigger on assertion of **p_devnt_out[3]** 0100 Trigger on assertion of **p_devnt_out[4]** 0101 Trigger on assertion of **p_devnt_out[5]** 0110 Trigger on assertion of **p_devnt_out[6]** 0111 Trigger on assertion of **p_devnt_out[7]** 1000 Trigger on rise of **p_pmc*n*_qual** input 1001–1111 Reserved • TRIGONCTL = 100: 0000 Trigger-on based on watchpoint #0 occurrence 0001 Trigger-on based on watchpoint #1 occurrence 0010 Trigger-on based on watchpoint #2 occurrence … 1110 Trigger-on based on watchpoint #14 occurrence 1111 Trigger-on based on watchpoint #15 occurrence • TRIGONCTL = 101: 0000 Trigger-on based on watchpoint #16 occurrence 0001 Trigger-on based on watchpoint #17 occurrence 0010 Trigger-on based on watchpoint #18 occurrence …. 1000 Trigger-on based on watchpoint #24 occurrence 1001 Trigger-on based on watchpoint #25 occurrence 1100–1111 Reserved
13 (45)	—	Reserved, should be cleared.

Table 8-6. PMLCb0–PMLCb3 Field Descriptions (continued)

Bits	Name	Description
14:17 (46:49)	TRIGOFFSEL	Trigger-off Source Select - Source Select based on setting of TRIGOFFCTL • TRIGOFFCTL = 000: 0000 Trigger-off control is disabled 0001–1111 Reserved • TRIGOFFCTL = 001: This field should be to the ID of the PMCy that should trigger event counting to stop. When PMCy overflows, the trigger will be generated. When TRIGOFFSEL = PMCx (i.e. self-select), no triggering will occur due to any counter change. If TRIGONSEL = TRIGOFFSEL, triggering results are undefined. 0000 Trigger-off when PMC0[OV] transitions to a 1. 0001 Trigger-off when PMC1[OV] transitions to a 1. 0010 Trigger-off when PMC2[OV] transitions to a 1. 0011 Trigger-off when PMC3[OV] transitions to a 1. 0100–1111 Reserved • TRIGOFFCTL = 010: 0000 Trigger-on when next processor interrupt occurs (software may want to set PMGC0[PMIE] = 0 for this setting). 0001–1111 Reserved • TRIGOFFCTL = 011: 0000 Trigger-off based on assertion of **p_devnt_out[0]** 0001 Trigger-off based on assertion of **p_devnt_out[1]** 0010 Trigger-off based on assertion of **p_devnt_out[2]** 0011 Trigger-off based on assertion of **p_devnt_out[3]** 0100 Trigger-off based on assertion of **p_devnt_out[4]** 0101 Trigger-off based on n assertion of **p_devnt_out[5]** 0110 Trigger-off based on assertion of **p_devnt_out[6]** 0111 Trigger-off based on assertion of **p_devnt_out[7]** 1000 Trigger-off based on fall of **p_pmc*n*_qual** input 1001–1111 Reserved • TRIGOFFCTL = 100: 0000 Trigger-off based on watchpoint #0 occurrence 0001 Trigger-off based on watchpoint #1 occurrence 0010 Trigger-off based on watchpoint #2 occurrence … 1110 Trigger-off based on watchpoint #14 occurrence 1111 Trigger-off based on watchpoint #15 occurrence • TRIGOFFCTL = 101: 0000 Trigger-off based on watchpoint #16 occurrence 0001 Trigger-off based on watchpoint #17 occurrence 0010 Trigger-off based on watchpoint #18 occurrence … 1000 Trigger-off based on watchpoint #24 occurrence 1001 Trigger-off based on watchpoint #25 occurrence 1100–1111 Reserved

Table 8-6. PMLCb0–PMLCb3 Field Descriptions (continued)

Bits	Name	Description
18 (50)	TRIGGERED	Triggered 0 Counter has not been triggered 1 Counter has been triggered TRIGGERED can be set or cleared by hardware or software. PMLCbx[TRIGONCTL] controls TRIGGERED setting by hardware. If PMLCbx[TRIGONCTL] is set to enable trigger-on control, TRIGGERED will be set by hardware when the next trigger-on event occurs and TRIGGERED is currently cleared. PMLCbx[TRIGOFFCTL] controls TRIGGERED clearing by hardware. If PMLCbx[TRIGOFFCTL] is set to enable trigger-off control, TRIGGERED will be cleared by hardware when the next trigger-off event occurs and TRIGGERED is currently set. The state of TRIGGERED qualifies counting if either PMLCbx[TRIGONCTL] or PMLCbx[TRIGOFFCTL] is set to enable triggering (other qualifiers on counting such as PMGC0[FAC] and PMLCa controls operate independently of TRIGGERED). If both PMLCbx[TRIGONCTL] and PMLCbx[TRIGOFFCTL] are cleared to disable triggering, the state of TRIGGERED has no effect on counting. TRIGGERED has no effect on PMLCa*x*[FC]; PMLCa*x*[FC] maintains its current value until changed by software.
19:20 (51:52)	—	Reserved, should be cleared.
21:23 (53:55)	THRESHMUL[1]	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)
24:25 (56:57)	—	Reserved, should be cleared.
26:31 (58:63)	THRESHOLD[1]	Threshold Only events that exceed this value multiplied by THRESHMUL 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 by monitoring a program repeatedly using a different threshold value each time.

[1] These fields are not implemented in PMLCb2 and PMLCb3 and are read as zero.

8.3.8 User Local Control B Registers (UPMLCb0–UPMLCb3)

The contents of PMLCb0–PMLCb3 are aliased to UPMLCb0–UPMLCb3, which can be read by user-level software with **mfpmr** using PMR numbers in Table 8-2.

8.3.9 Performance Monitor Counter Registers (PMC0–PMC3)

The performance monitor counter registers PMC0–PMC3 shown in Figure 8-4 are 32-bit counters that can be programmed to generate overflow event signals when they overflow. Each counter is enabled to count up to 128 processor events.

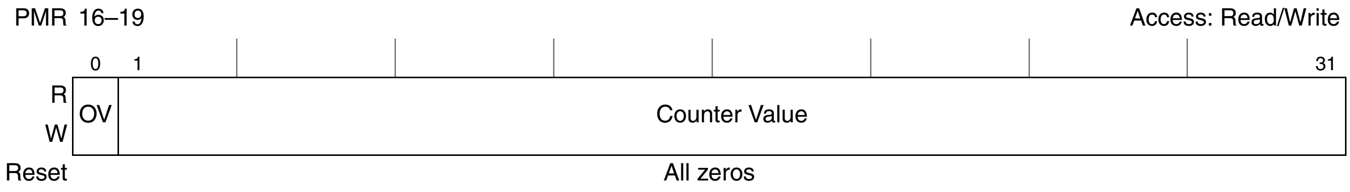


Figure 8-4. Performance Monitor Counter Registers (PMC0–PMC3)

PMCs are cleared by reset. Table 8-7 describes the PMC register fields.

Table 8-7. PMC0–PMC3 Field Descriptions

Bits	Name	Description
0 (32)	OV	Overflow 0 Counter has not reached an overflow state. 1 Counter has reached an overflow state.
1–31 (33–63)	Counter Value	Indicates the number of occurrences of the specified event.

A counter can increment by 0, 1, 2, 3, or 4 (based on the number of events occurring in a given counter cycle) up to the maximum value and then wraps to the minimum value.

A counter enters the overflow state when the high-order bit is set. 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 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 may be falsely generated before any events are counted.

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

- If PMLCa*n*[CE] is clear, no special actions occur on overflow of PMC*n*: the counter continues incrementing, and no event is signaled.
- If PMLCa*n*[CE] and PMGC0[FCECE] are both set, all counters are frozen when PMC*n* overflows.

- If PMLCa*n*[CE] and PMGC0[PMIE] are set, an exception is signaled on overflow of PMC*n*. Performance Monitor Interrupts are masked when MSR[EE] = 0. An exception may be signaled while MSR[EE] = 0, but the interrupt is not taken until MSR[EE] = 1 and is only guaranteed to be taken if the overflow condition is still present and the configuration has not been changed in the meantime to disable the exception. If PMLCa*n*[CE] or PMGC0[PMIE] is cleared, the exception is no longer 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**.

8.3.10 User Performance Monitor Counter Registers (UPMC0–UPMC3)

The contents of PMC0–PMC3 are aliased to UPMC0–UPMC3, which can be read by user-level software with the **mfpmr** instruction using PMR numbers in Table 8-2.

8.4 Performance Monitor Interrupt

The performance monitor interrupt is triggered by an enabled condition or event. The enabled condition or events defined for the e200z7 are 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.
- A time base event occurs with the following settings:
  - Time base events are enabled with PMGC0[TBEE] = 1
  - the TBL bit specified in PMGC0[TBSEL] changes from 0 to 1

The two performance monitor exception conditions are treated differently with respect to whether or not the conditions are level sensitive or edge sensitive. A performance monitor exception condition which is caused by a PMCn overflow condition is level sensitive to the values of PMLCA*n*[CE] and PMC*n*[OV]. This means that as long as these values are both set to '1', then the exception condition continues to exist and the performance monitor interrupt can be taken if the remainder of the performance monitor interrupt gating conditions are met. However, the exception due to the time base event is set only when both PMGC0[TBEE] = 1 and the transition from '0' to '1' occurs in the specified TBL bit. This condition is not cleared once it occurs, regardless of whether the TBL bit subsequently transitions to a '0', but this exception is automatically cleared whenever any performance monitor interrupt is subsequently taken.

- 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 forces all performance monitor counters to freeze.

Although the performance monitor exception condition may occur with MSR[EE] = 0, the interrupt cannot be taken until MSR[EE] = 1. If PMC*n* overflows and would signal an exception (PMLCa*n*[CE] = 1 and

PMGC0[PMIE] = 1) while MSR[EE] = 0, and freezing of the counters is not enabled (PMGC0[FCECE] is clear), it is possible that PMC*n* could wrap around to all zeros again without the performance monitor interrupt being taken.

Interrupt handlers should clear a counter overflow condition or the corresponding Condition Enable to avoid a repeated interrupt to occur for the same event.

The priority of the performance monitor interrupt is specified in Section 7.7.1, “Exception Priorities.

8.5 Event Counting

This section describes configurability and specific unconditional counting modes.

8.5.1 MSR-based Context Filtering

Counting can be configured to be conditionally 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] and MSR[PMM] 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, counting is enabled for PMC*n*.

For the e200z7 implementation, a given event may or may not support MSR-based context filtering. For events that do not support MSR-based context filtering, the FCS, FCU, FCM1, and FCM0 controls have no effect on the counting of that event.

The processor states and the settings of the FCS, FCU, FCM1, and FCM0 bits in PMLCa*n* necessary to enable monitoring of each processor state are shown in Table 8-8.

Table 8-8. Processor States and PMLCa0–PMLCa3 Bit Settings

Processor State	FCS	FCU	FCM1	FCM0
All (no context filtering)	0	0	0	0
Marked	0	0	0	1
Not marked	0	0	1	0
Supervisor	0	1	0	0
Marked and supervisor	0	1	0	1
Not marked and supervisor	0	1	1	0
User	1	0	0	0
Marked and user	1	0	0	1
Not marked and user	1	0	1	0

Table 8-8. Processor States and PMLCa0—PMLCa3 Bit Settings (continued)

Processor State	FCS	FCU	FCM1	FCM0
None (counting disabled)	X	X	1	1
None (counting disabled)	1	1	X	X

8.6 Examples

The following sections provide examples of how to use the performance monitor facility.

8.6.1 Chaining Counters

The counter chaining feature can be used to allow a higher event count than is possible with a single counter. Chaining two counters together effectively adds 32 bits to a counter register where rollover of the first counter generates a carry out feeding the second counter. By defining the event of interest to be another PMC's rollover occurrence, 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, a rollover 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.

8.6.2 Thresholding

Threshold event measurement enables the counting of duration and usage events. For example, data cache load miss cycles (events C0:xx and C1:xx) require a threshold value. A data cache load miss cycles event is counted only when the number of cycles spent waiting for the miss is greater than the threshold. Because this event is supported by 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 may expedite code profiling significantly.

8.7 Event Selection

Event selection is specified through the PMLCa*n* registers described in Section 8.3.5, “Local Control A Registers (PMLCa0–PMLCa3). The event-select fields in PMLCa*n*[EVENT] are described in Table 8-10, which lists encodings for the selectable events to be monitored. Table 8-10 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 operations that were later flushed.
- Non-speculative counts do not include speculative operations, which are flushed.

The PR, PMM filtering column indicates whether a given event supports MSR-based context filtering.

Table 8-9 describes how event types are indicated in Table 8-10.

Table 8-9. Event Types

Event Type	Label	Description
Reference	Ref:#	Shared across counters PMC0–PMC3.
Common	Com:#	Shared across counters PMC0–PMC3.
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 PMC0 is shown as C0:#.

Table 8-10 describes performance monitor events.

Table 8-10. Performance Monitor Event Selection

Number	Event	Spec/ Nonspec	PR, PMM Filtering[1]	Count Description
General Events				
Com:0	Nothing	Nonspec	—	Register counter holds current value
Ref:1[2]	Processor cycles	Nonspec	Yes	Every processor cycle not in waiting, halted, stopped states and not in a debug session.
Com:2[3]	Instructions completed	Nonspec	Yes	Completed instructions. 0, 1, 2, or 3 per cycle.
Com:3[2]	Processor cycles with 0 instructions issued	Nonspec	Yes	Ref:1 cycles with no instructions entering execution
Com:4[2]	Processor cycles with 1 instruction issued	Nonspec	Yes	Ref:1 cycles with one instruction entering execution
Com:5[2]	Processor cycles with 2 instructions issued	Nonspec	Yes	Ref:1 cycles with two instructions entering execution
Com:6[3]	Instruction words fetched	Spec	Yes	Fetched instruction words. 0, 1, or 2, 3, or 4 per cycle. (note that an instruction word may hold 1 or 2 instructions, or 2 partial instructions when fetching from a VLE page)
Com:7	—	—	—	—
Com:8	PM_EVENT transitions	—	—	0 to 1 transitions on the *p_pm_event* input.
Com:9	PM_EVENT cycles	—	—	Processor (Ref:1) cycles that occur when the *p_pm_event* input is asserted.
Instruction Types Completed				
Com:10[3]	Branch instructions completed	Nonspec	Yes	Completed branch instructions, includes branch and link type instructions

Table 8-10. Performance Monitor Event Selection (continued)

Number	Event	Spec/ Nonspec	PR, PMM Filtering[1]	Count Description
Com:11[3]	Branch and link type instructions completed	Nonspec	Yes	Completed branch and link type instructions
Com:12[3]	Conditional branch instructions completed	Nonspec	Yes	Completed conditional branch instructions
Com:13[3]	Taken Branch instructions completed	Nonspec	Yes	Completed branch instructions which were taken. Includes branch and link type instructions.
Com:14[3]	Taken Conditional Branch instructions completed	Nonspec	Yes	Completed conditional branch instructions which were taken.
Com:15[3]	Load instructions completed	Nonspec	Yes	Completed load, load-multiple type instructions
Com:16[3]	Store instructions completed	Nonspec	Yes	Completed store, store-multiple type instructions
Com:17[3]	Load micro-ops completed	Nonspec	Yes	Completed load micro-ops. (**l***, **evl***, load-update (1 load micro-op), load-multiple (1–32 micro-ops), **dcbt**, **dcbtls**, **dcbtst**, **dcbtstls**, and **dcbtst**, **dcbf**, **dcblc**, **dcbst**, **icbi**, **icblc**, **icbt**, **icbtls)**. Misaligned loads crossing a 64-bit boundary count as two micro-ops.
Com:18[3]	Store micro-ops completed	Nonspec	Yes	Completed store micro-ops. (**st***, **evst***, store-update (1 store micro-op), store-multiple (1–32 micro-ops), **dcbi**, **dcbz**). Misaligned stores crossing a 64-bit boundary count as two micro-ops.
Com:19[3]	Integer instructions completed	Nonspec	Yes	Completed simple integer instructions (not a load-type/store-type/branch/mul/div, EFPU, or SPE)
Com:20[3]	Multiply instructions completed	Nonspec	Yes	Completed Multiply instructions (non-EFPU)
Com:21[3]	Divide instructions completed	Nonspec	Yes	Completed Divide instructions including SPE (non-EFPU)
Com:22[3]	Divide instruction execution cycles	Nonspec	Yes	Cycles of execution for all Divide instructions (non-EFPU)
Com:23[3]	SPE/EFPU instructions completed	Nonspec	Yes	Completed SPE/EFPU instructions. Does not include SPE/EFPU load and store instructions.
Com:24[3]	SPE simple instructions completed	Nonspec	Yes	Completed SPE simple instructions. All SPE instructions included except SPE load and store instructions, div, dotp, mul and mac-type instructions.
Com:25[3]	SPE mul/mac/dotp instructions completed	Nonspec	Yes	Completed SPE mul/mac/dotp instructions. Does not include other SPE instructions, or **brinc** instructions.
Com:26[3]	EFPU FP instructions completed	Nonspec	Yes	Completed EFPU FP (evfs, efs) instructions.
Com:27[3]	Number of return from interrupt instructions	Nonspec	Yes	Includes all types of return from interrupts (i.e. **rfi, rfci, rfdi, rfmci,** and VLE variants)

Table 8-10. Performance Monitor Event Selection (continued)

Number	Event	Spec/ Nonspec	PR, PMM Filtering[1]	Count Description
Branch Prediction and Execution Events				
Com:28[3]	Finished branches that miss the BTB	Spec	Yes	Includes all taken branch instructions which missed in the BTB
Com:29[3]	Branches mispredicted (for any reason)	Spec	Yes	Counts branch instructions mispredicted due to direction or target (for example if the LR or CTR contents change).
Com:30[3]	Branches in the BTB mispredicted due to direction prediction.	Spec	Yes	Counts branch instructions which hit the BTB with mispredicted due to direction prediction.
Com:31[3]	Incorrect target prediction using the link stack	Spec	Yes	—
Com:32[3]	BTB hits	Spec	Yes	Branch instructions that hit in the BTB
Com:33	—	—	—	—
Com:34	—	—	—	—
Pipeline Stalls				
Com:35	—	—	—	—
Com:36	—	—	—	—
Com:37[2]	Cycles decode stalled due to no instructions available	Spec	Yes	No instruction available to decode
Com:38[2]	Cycles issue stalled	Spec	Yes	Cycles the issue buffer is not empty but 0 instructions issued
Com:39[2]	Cycles branch issue stalled	Spec	Yes	Branch held in decode awaiting resolution
Com:40[2]	Cycles execution stalled waiting for load data	Spec	Yes	load stalls
Com:41[2]	Cycles execution stalled waiting for non-load/store SPE/EFPU result data	Spec	Yes	Stalled waiting on mul, div, FP or MAC results
Load/Store, Data Cache, and Data Line Fill Events				
Com:42	—	—	—	—
Com:43	—	—	—	—
Com:44[3]	Total translation hits	Spec	Yes	—
Com:45[3]	Load translation hits	Spec	Yes	Cacheable **l*** or **evl*** micro-ops translated. (includes load micro-ops from load-multiple and load-update instructions)
Com:46[3]	Store translation hits	Spec	Yes	Cacheable **st*** or **evst*** micro-ops translated. (includes micro-ops from store-multiple, and store-update instructions)

Table 8-10. Performance Monitor Event Selection (continued)

Number	Event	Spec/ Nonspec	PR, PMM Filtering[1]	Count Description
Com:47[3]	Touch translation hits	Spec	Yes	Cacheable **dcbt** and **dcbtst** instructions translated (L1 only) (Doesn't count touches that are converted to nops i.e. exceptions, non-cacheable, HID0[NOPTI] is set.)
Com:48[3]	Data cache op translation hits	Spec	Yes	**dcba**, **dcbf**, **dcbst**, and **dcbz** instructions translated
Com:49[3]	Data cache lock set instructions completed	Nonspec	Yes	**dcbtls** and **dcbtstls** instructions completed
Com:50[3]	Data cache lock clear instructions completed	Nonspec	Yes	**dcblc** instructions completed
Com:51[3]	Cache-inhibited load access translation hits	Spec	Yes	Cache inhibited load accesses translated
Com:52[3]	Cache-inhibited store access translation hits	Spec	Yes	Cache inhibited store accesses translated
Com:53[3]	Guarded load translation hits	Spec	Yes	Guarded loads translated
Com:54[3]	Guarded store translation hits	Spec	Yes	Guarded stores translated
Com:55[3]	Write-through store translation hits	Spec	Yes	Write-through stores translated
Com:56[3]	Misaligned load or store accesses translated	Spec	Yes	Misaligned load or store accesses translated. Count once per misaligned load or store.
Com:57[3]	Dcache linefills	Spec	Yes	Counts dcache reloads for any reason, including touch-type reloads. Typically used to determine approximate data cache miss rate (along with loads/stores completed).
Com:58[3]	Dcache copybacks	Spec	Yes	Does not count copybacks due to **dcbf**, **dcbst**, or L1FINV0 operations
Com:59[3]	Dcache sequential accesses	Spec	Yes	Number of sequential accesses
Com:60[3]	Dcache stream hits	Spec	Yes	Number of load hits due to streaming
Com:61[3]	Dcache linefill buffer hits	Spec	Yes	Number of load hit to the linefill buffer
Com:62[3]	Store stalls due to store to line of active linefill	Spec	Yes	Stall cycles due to store to linefill in progress
Com:63[3]	Store buffer full stalls	Spec	Yes	Stall cycles due to store buffer full
Com:64[2]	Dcache throttling stalls	Spec	Yes	Cycles the data cache asserts **p_d_halt_zlb** which actually cause a CPU stall
Com:65[3]	Dcache recycled accesses	Spec	Yes	Number of loads or stores recycled for a re-lookup
Com:66[3]	Dcache recycled access stalls	Spec	Yes	Number of stall cycles due to recycled accesses for a re-lookup

Table 8-10. Performance Monitor Event Selection (continued)

Number	Event	Spec/ Nonspec	PR, PMM Filtering[1]	Count Description
Com:67[3]	Dcache CPU aborted accesses	Spec	Yes	Number of aborted requests
Com:68[3]	Data MMU miss	Spec	Yes	Counts number of DTLB events
Com:69[3]	Data MMU error	Spec	Yes	Counts number of DSI events
Fetch, Instruction Cache, Instruction Line Fill, and Instruction Prefetch Events				
Com:70	—	—	—	—
Com:71	—	—	—	—
Com:72[3]	Icache linefills	Spec	Yes	Counts icache reloads due to demand fetch. Used to determine instruction cache miss rate (along with instructions completed)
Com:73[3]	Number of fetches	Spec	Yes	Counts fetches that write at least one instruction to the instruction buffer. (With instruction fetched (com:4), can used to compute instructions-per-fetch)
Com:74[3]	Icache lock set instructions completed	Nonspec	Yes	**icbtls** instructions completed
Com:75[3]	Icache lock clear instructions completed	Nonspec	Yes	**icblc** instructions completed
Com:76[3]	Cache-inhibited instruction access translation hits	Spec	Yes	Cache-inhibited instruction accesses translated
Com:77[2]	Icache throttling stalls	Spec	Yes	Cycles the instruction cache asserts **p_i_halt_zlb** that actually cause a CPU stall
Com:78[3]	Icache recycled accesses	Spec	Yes	Number of instruction access requests recycled for a re-lookup
Com:79[3]	Icache recycled access stalls	Spec	Yes	Number of stall cycles due to recycled accesses for a re-lookup
Com:80[3]	Icache CPU aborted accesses	Spec	Yes	Number of aborted requests
Com:81[3]	Instruction MMU miss	Spec	Yes	Counts number of events
Com:82[3]	Instruction MMU error	Spec	Yes	Counts number of events
BIU Interface Usage				
Com:83	—	—	—	—
Com:84	—	—	—	—
Com:85[3]	BIU instruction-side requests	Spec	Yes	instruction-side transactions
Com:86[3]	BIU instruction-side cycles	Spec	Yes	instruction-side transaction cycles
Com:87[3]	BIU data-side requests	Spec	Yes	data-side transactions

Table 8-10. Performance Monitor Event Selection (continued)

Number	Event	Spec/ Nonspec	PR, PMM Filtering[1]	Count Description
Com:88[3]	BIU data-side copyback requests	Spec	Yes	Replacement pushes including **dcbf**, **dcbst,** L1FINV0, copybacks.
Com:89[3]	BIU data-side cycles	Spec	Yes	data-side transaction cycles
Com:90[3]	BIU single-beat write cycles	Non-Spec	Yes	single beat write transaction cycles
Com:91	—	—	—	—
Snoop				
Com:92	Snoop requests	N/A	—	Externally generated snoop requests. (Counts snoop TSs.)
Com:93	Snoop hits	N/A	—	Snoop hits on all data-side resources regardless of the cache state (modified, shared, or exclusive)
Com:94[3]	Snoop induced CPU to Dcache stalls	N/A	—	Cycles a pending Dcache access from CPU is stalled due to contention with snoops
Com:95	Snoop Queue full cycles	N/A	—	Cycles the snoop queue is full
Com:96	—	—	—	—
Chaining Events[4]				
Com:97	PMC0 rollover	N/A	—	PMC0[OV] transitions from 1 to 0.
Com:98	PMC1 rollover	N/A	—	PMC1[OV] transitions from 1 to 0.
Com:99	PMC2 rollover	N/A	—	PMC2[OV] transitions from 1 to 0.
Com:100	PMC3 rollover	N/A	—	PMC3[OV] transitioned from 1 to 0.
Interrupt Events				
Com:101	—	—	—	—
Com:102	—	—	—	—
Com:103	Interrupts taken	Nonspec	—	—
Com:104	External input interrupts taken	Nonspec	—	—
Com:105	Critical input interrupts taken	Nonspec	—	—
Com:106	Watchdog timer interrupts taken	Nonspec	—	—
Com:107	System call and trap interrupts	Nonspec	Yes	—
Com:108[2]	Cycles in which MSR[EE] = 0	Nonspec	—	—
Com:109[2]	Cycles in which MSR[CE] = 0	Nonspec	—	—

Table 8-10. Performance Monitor Event Selection (continued)

Number	Event	Spec/ Nonspec	PR, PMM Filtering[1]	Count Description
Ref:110	Transitions of TBL bit selected by PMGC0[TBSEL].	Nonspec	—	Counts transitions of the TBL bit selected by PMGC0[TBSEL]. Counts both 0→1 and 1→0.
DEVENT Events				
Com:111	DEVNT0 is generated	Nonspec	Yes	assertion of **p_devnt_out0** detected
Com:112	DEVNT1 is generated	Nonspec	Yes	assertion of **p_devnt_out1** detected
Com:113	DEVNT2 is generated	Nonspec	Yes	assertion of **p_devnt_out2** detected
Com:114	DEVNT3 is generated	Nonspec	Yes	assertion of **p_devnt_out3** detected
Com:115	DEVNT4 is generated	Nonspec	Yes	assertion of **p_devnt_out4** detected
Com:116	DEVNT5 is generated	Nonspec	Yes	assertion of **p_devnt_out5** detected
Com:117	DEVNT6 is generated	Nonspec	Yes	assertion of **p_devnt_out6** detected
Com:118	DEVNT7 is generated	Nonspec	Yes	assertion of **p_devnt_out7** detected
Watchpoint Events				
Com:119[2]	Watchpoint #0 occurs	Nonspec	Yes	assertion of **jd_watchpt0** detected
Com:120[2]	Watchpoint #1 occurs	Nonspec	Yes	assertion of **jd_watchpt1** detected
Com:121[2]	Watchpoint #2 occurs	Nonspec	Yes	assertion of **jd_watchpt2** detected
Com:122[2]	Watchpoint #3 occurs	Nonspec	Yes	assertion of **jd_watchpt3** detected
Com:123[2]	Watchpoint #4 occurs	Nonspec	Yes	assertion of **jd_watchpt4** detected
Com:124[2]	Watchpoint #5 occurs	Nonspec	Yes	assertion of **jd_watchpt5** detected
Com:125[2]	Watchpoint #6 occurs	Nonspec	Yes	assertion of **jd_watchpt6** detected
Com:126[2]	Watchpoint #7 occurs	Nonspec	Yes	assertion of **jd_watchpt7** detected
Com:127[2]	Watchpoint #8 occurs	Nonspec	Yes	assertion of **jd_watchpt8** detected
Com:128[2]	Watchpoint #9 occurs	Nonspec	Yes	assertion of **jd_watchpt9** detected
Com:129	Watchpoint #10 occurs	Nonspec	Yes	assertion of **jd_watchpt10** detected
Com:130	Watchpoint #11 occurs	Nonspec	Yes	assertion of **jd_watchpt11** detected
Com:131	Watchpoint #12 occurs	Nonspec	Yes	assertion of **jd_watchpt12** detected
Com:132	Watchpoint #13 occurs	Nonspec	Yes	assertion of **jd_watchpt13** detected
Com:133[2]	Watchpoint #14 occurs	Nonspec	Yes	assertion of **jd_watchpt14** detected
Com:134[2]	Watchpoint #15 occurs	Nonspec	Yes	assertion of **jd_watchpt15** detected
Com:135[2]	Watchpoint #16 occurs	Nonspec	Yes	assertion of **jd_watchpt16** detected
Com:136[2]	Watchpoint #17 occurs	Nonspec	Yes	assertion of **jd_watchpt17** detected
Com:137[2]	Watchpoint #18 occurs	Nonspec	Yes	assertion of **jd_watchpt18** detected

Table 8-10. Performance Monitor Event Selection (continued)

Number	Event	Spec/ Nonspec	PR, PMM Filtering[1]	Count Description
Com:138[2]	Watchpoint #19 occurs	Nonspec	Yes	assertion of **jd_watchpt19** detected
Com:139	Watchpoint #20 occurs	Nonspec	Yes	assertion of **jd_watchpt20** detected
Com:140	Watchpoint #21 occurs	Nonspec	Yes	assertion of **jd_watchpt21** detected
Com:141	Watchpoint #22 occurs	Nonspec	Yes	assertion of **jd_watchpt22** detected
Com:142	Watchpoint #23 occurs	Nonspec	Yes	assertion of **jd_watchpt23** detected
Com:143	Watchpoint #24 occurs	Nonspec	Yes	assertion of **jd_watchpt24** detected
Com:144	Watchpoint #25 occurs	Nonspec	Yes	assertion of **jd_watchpt25** detected
Com:145	Watchpoint #26 occurs	Nonspec	Yes	assertion of **jd_watchpt26** detected
Com:146	—	—	—	—
Com:147	—	—	—	—
Com:148	—	—	—	—
Com:149	—	—	—	—
Com:150	—	—	—	—
NEXUS Events				
Com:151[3]	Cycle CPU is stalled by Nexus3 FIFO full	Nonspec	Yes	OVCR stall control set to stall on FIFO fullness
Threshold Events				
C0:152[3] C1:152[3]	Data cache load miss cycles	Spec	Yes	Instances when the number of cycles between a load miss in the data cache and update of the data cache exceeds the threshold.
C0:153[3] C1:153[3]	Instruction cache fetch miss cycles	Spec	Yes	Instances when the number of cycles between miss in the instruction cache and update of the instruction cache exceeds the threshold.
C0:154[3] C1:154[3]	External input interrupt latency cycles	N/A	—	Instances when the number of cycles between request for interrupt (*p_int_b*) asserted (but possibly masked/disabled) and redirecting fetch to external interrupt vector exceeds threshold. Once the redirection has occurred, no further threshold comparisons are made until either the interrupt request negates, or the external input interrupt is re-enabled by setting MSR[EE].
C0:155[3] C1:155[3]	Critical input interrupt latency cycles	N/A	—	Instances when the number of cycles between request for critical interrupt (*p_critint_b*) is asserted (but possibly masked/disabled) and redirecting fetch to the critical interrupt vector exceeds threshold. Once the redirection has occurred, no further threshold comparisons begin until either the interrupt request negates and is then re-asserted, or the critical input interrupt is re-enabled by setting MSR[CE].

Table 8-10. Performance Monitor Event Selection (continued)

Number	Event	Spec/ Nonspec	PR, PMM Filtering[1]	Count Description
C0:156[3] C1:156[3]	Watchdog timer interrupt latency cycles	N/A	—	Instances when the number of cycles between watchdog timer time-out request for critical interrupt becomes pending (watchdog interrupt enabled (TCR[WIE] set) and time-out occurs (TSR[ENW, WIS] become 0b11)) and redirecting fetch to the critical interrupt vector exceeds the threshold. Once the redirection has occurred, no further threshold comparisons begin until either the watchdog interrupt request negates and is then re-asserted, or the watchdog interrupt is re-enabled by setting MSR[CE].
C0:157[3] C1:157[3]	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. Once the redirection has occurred, no further threshold comparisons are made until either the interrupt request negates and is then re-asserted, or the external input interrupt is re-enabled by setting MSR[EE].
C0:158[3] C1:158[3]	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. Once the redirection has occurred, no further threshold comparisons are made until either the interrupt request negates and is then re-asserted, or the critical input interrupt is re-enabled by setting MSR[CE].

[1] The notation for the PR, and PMM filtering column either contains a 'yes' or a '—'. A 'yes' indicates that the MSR-based context filtering function is available for that event. A '—' indicates that the MSR-based context filtering is not available for that event and has no effect on the counting of that event. See Section 8.5.1, "MSR-based Context Filtering" for more information.

[2] This event is not counted while the processor is in the waiting, halted, or stopped states, or during a debug session

[3] This event is not counted while the processor is in a debug session.

[4] For chaining events, if a counter is configured to count its own rollover, the result is undefined.

Chapter 9
L1 Cache

This chapter describes the organization of the on-chip L1 caches, cache control instructions, and various cache operations. It describes the interaction between the caches, the load/store unit (LSU), the instruction unit, and the memory subsystem. This chapter also describes the replacement algorithm used for the L1 caches.

The L1 caches incorporate the following features:

- 16-KB I + 16-KB D Harvard cache design
- Virtually indexed, physically tagged
- 32-byte line size
- 64-bit data, 32-bit address
- Pseudo round-robin replacement algorithm
- 8-entry store buffer
- Push (copyback) buffer
- Linefill buffer
- Hit under fill/copyback
- Supports up to two outstanding misses
- Parity or Multibit EDC protection for the ICache data and tag arrays, with correction/auto-invalidation capability
- Parity or Multibit EDC protection for the DCache tag arrays, parity protection for the DCache data arrays with correction/auto-invalidation capability

9.1 Overview

The processor supports a pair of 16-KB, 4-way set-associative, split instruction and data caches with a 32-byte line size. The caches improve system performance by providing low-latency data to the e200z7 instruction and data pipelines, which decouples processor performance from system memory performance. The caches are virtually indexed and physically tagged.

Instruction and data addresses from the processor to the caches are virtual addresses used to index the cache array. The MMU provides the virtual to physical translation for use in performing the cache tag compare. If the physical address matches a valid cache tag entry, the access hits in the cache. For a read operation, the cache supplies the data to the processor, and for a write operation, the data from the processor updates the cache. If the access does not match a valid cache tag entry (misses in the cache) or a write access must be written through to memory, the cache performs a bus cycle on the system bus.

Figure 9-1 shows the e200z7 caches.

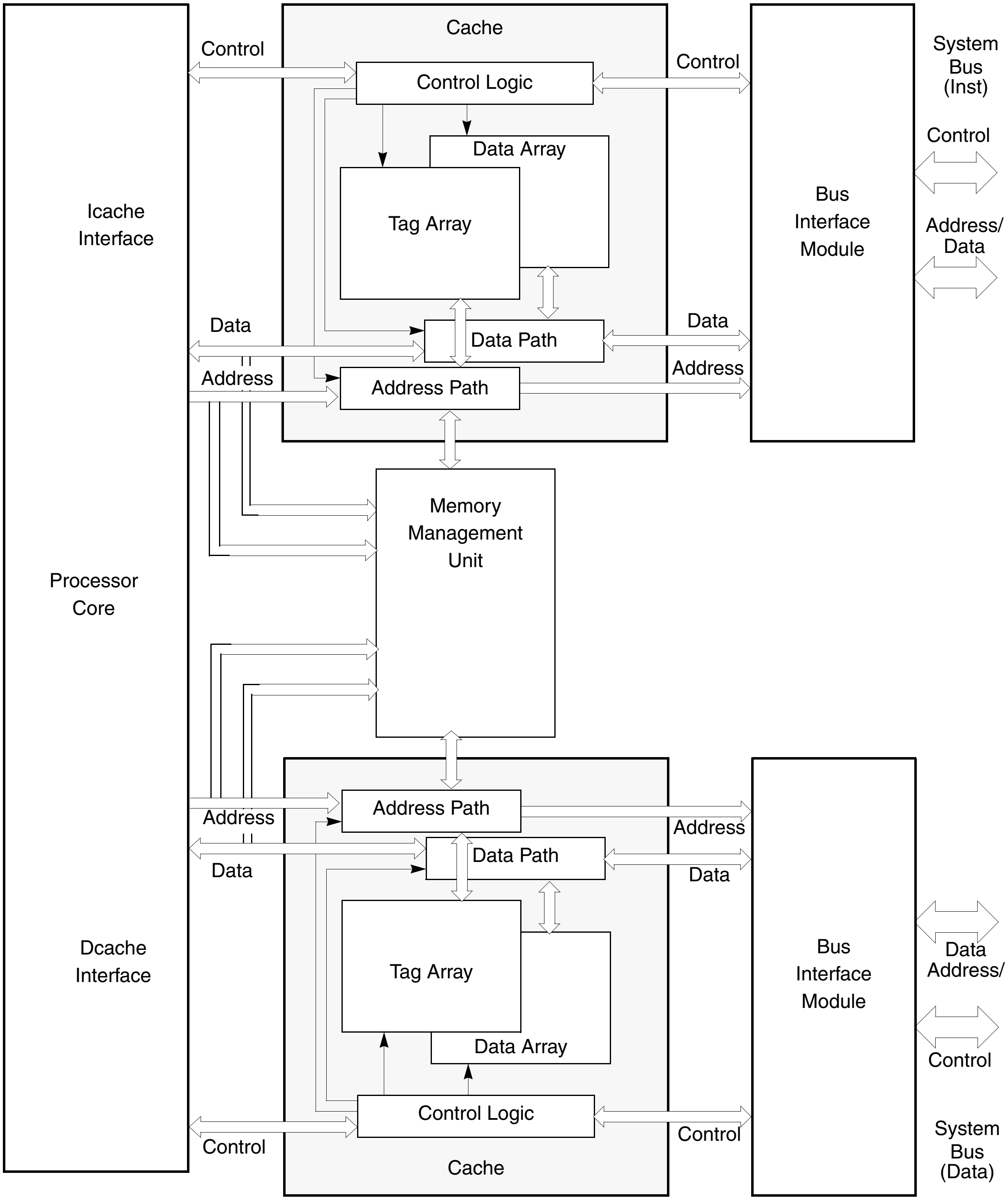
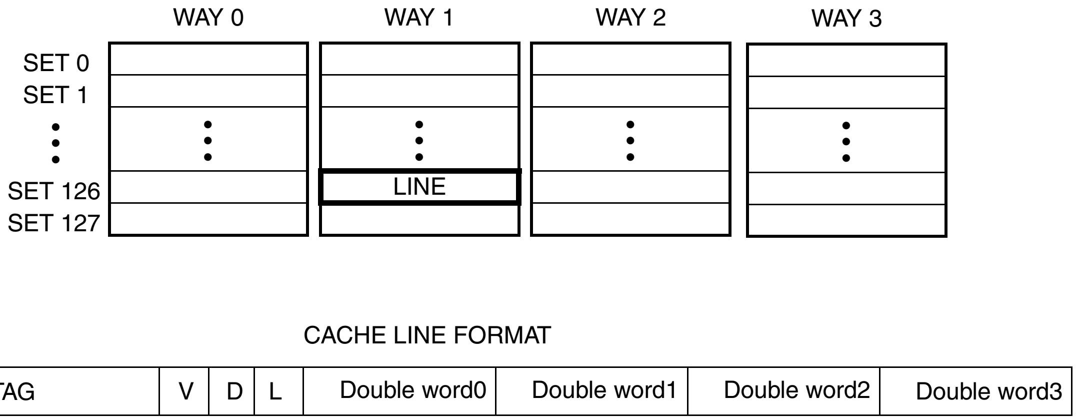


Figure 9-1. e200z7 Caches

9.2 16-KB Cache Organization

Each 16-KB cache is organized as four ways of 128 sets with each line containing 32 bytes (four double words) of storage. Figure 9-2 illustrates the cache organization along with the cache line format:



TAG = 22 bit Physical Address Tag + Parity

L = Lock bits
D = Dirty bits (DCACHE Only)
V = Valid bit

Figure 9-2. 16-KB Cache Organization and Line Format

Virtual address bits A[20–26] provide an index to select a set. Ways are selected according to the rules of set association.

Each line consists of a physical address tag, status bits, and four double words of data. Address bits A[27–29] select the word within the line.

9.3 Cache Lookup

Once enabled, the appropriate cache will be searched for a tag match on instruction fetches and data accesses from the CPU. If a match is found, the cached data is forwarded on a read access to the instruction fetch unit or the load/store unit (data access), or it is updated on a write access. It may also be written-through to memory if required.

When a read miss occurs, if there is a TLB hit and the I bit of the hitting TLB entry is clear, the translated physical address is used to fetch a four double-word cache line beginning with the requested double-word (critical double-word first). The line is fetched into a linefill buffer and the critical double-word is forwarded to the CPU. Subsequent double-words may be streamed to the CPU if they have been requested, or they may be forwarded from the linefill buffer if the data has already been received from the bus and is valid in the buffer.

When a write miss occurs, if there is a TLB hit, and the I and G bits of the hitting TLB entry are clear and write allocation is enabled via the L1CSR0[DCWA] control bit, the translated physical address is used to fetch a four double-word cache line beginning with the double word corresponding to the store address (critical double-word first). The line is fetched into the linefill buffer and merged with the store data.

Subsequently, the line is placed into the appropriate cache block. If write allocation is disabled, or the write is not cacheable or is guarded, no cache line fetch is performed for the write.

During a cache line fill, double words received from the bus are placed into the cache linefill buffer, and may be forwarded (streamed) to the CPU if such a read request is pending. Accesses from the CPU following delivery of the critical double word may be satisfied from the cache (hit under fill, non-blocking) or from the linefill buffer if the requested information has been already received.

If write allocation is enabled, subsequent stores that hit the linefill buffer address while a linefill is in progress for a previous store or **dcbtst** miss are merged into the linefill buffer. No merging of stores are performed during a linefill initiated by a load miss.

When a cache linefill occurs, the linefill buffer contents are placed into the cache array using two accesses; each occurs after receiving a pair of double words.

The cache always fills an entire line, thereby providing validity on a line-by-line basis. A DCache line is always in one of the following states: invalid, valid, or dirty (and valid). The state settings are as follows:

- For invalid lines, the V bit is clear, causing the cache line to be ignored during lookups.
- For valid lines, the V bit is set and D bits are cleared, indicating the line contains valid data consistent with memory.
- For dirty lines, the D and V bits are set, indicating that the line has valid entries that have not been written to memory.

ICache lines are either invalid or valid. In addition, a cache line in either cache may be locked (L bits set), indicating the line is not available for replacement.

The caches should be explicitly invalidated after a hardware reset; reset does not invalidate the cache lines. Following initial power-up, the cache contents are undefined. The L, D, and V bits may be set on some lines, necessitating the invalidation of the caches by software before being enabled.

Figure 9-3 illustrates the general flow of cache operation for each 16KB cache to determine if the address is already allocated in the cache,

1. The cache set index, virtual address bits A[20–26] are used to select one cache set. A set is defined as the grouping of four lines (one from each way), corresponding to the same index into the cache array.
2. The higher order physical address bits A[0–21] are used as a tag reference or used to update the cache line tag field.
3. The tags from the selected cache set are compared with the tag reference. If any one of the tags matches the tag reference and the tag status is valid, a cache hit has occurred.
4. Virtual address bits A[27–28] are used to select one of the four double words in each line. A cache hit indicates that the selected double word in that cache line contain valid data (for a read access), or can be written with new data depending on the status of the W access control bit from the MMU (for a write access to the DCache).

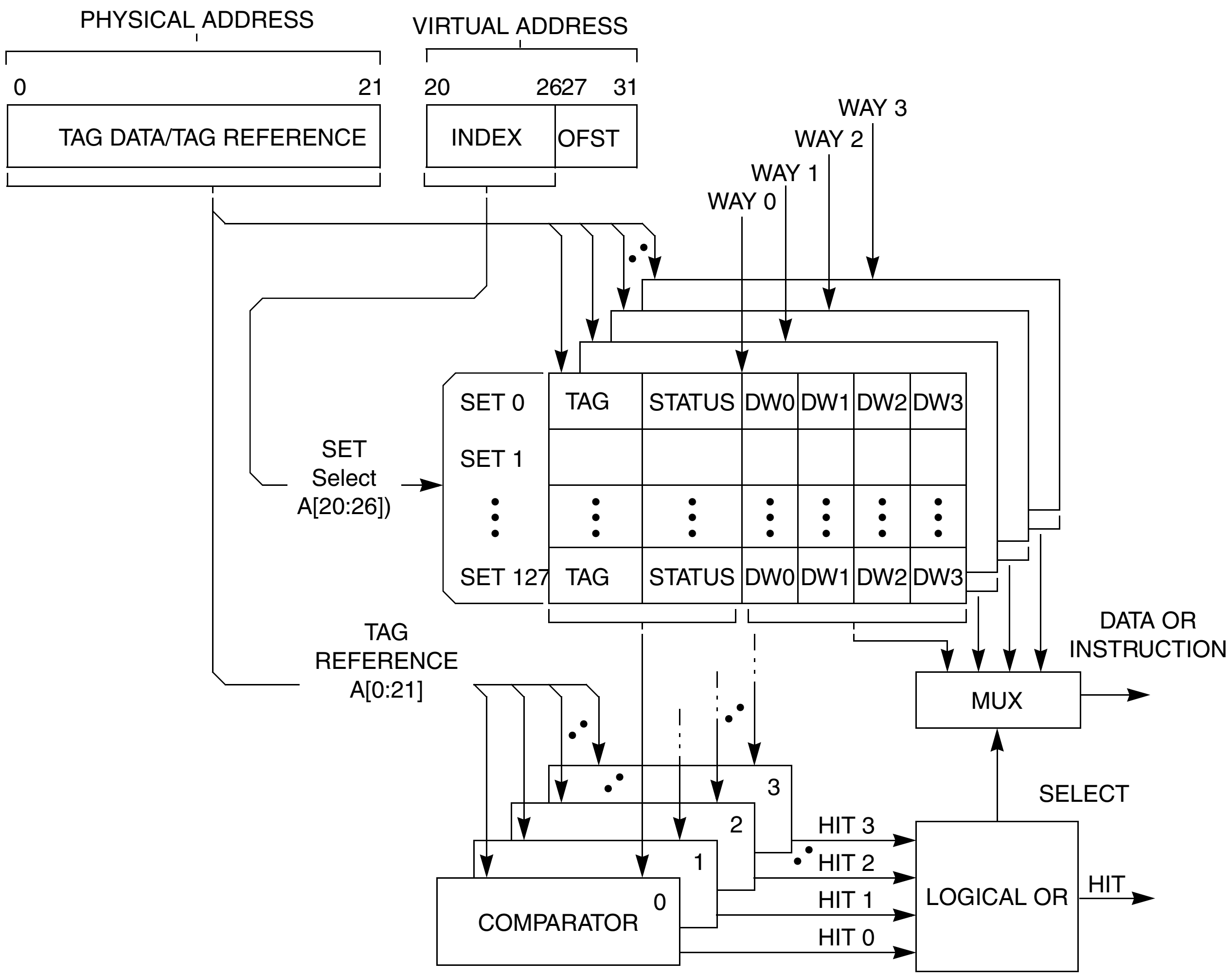


Figure 9-3. 16-KB Cache Lookup Flow

9.4 Cache Control

Control of the cache is provided by bits in the L1 cache control and status registers (L1CSR0, L1CSR1). Control bits are provided to enable/disable the cache and to invalidate it of all entries. In addition, availability of each way of the caches may be selectively controlled for use. This way control provides cache way locking capability, as well as controlling way availability on a cache line replacement. Ways 0–3 may be selectively disabled for instruction miss replacements and data miss replacements in the respective caches by using the WID and WDD control bits. Software is responsible for maintaining coherency between instruction and data caches, since independent copies of a cache line may be present in both caches: one allocated by an instruction access and another by a data access.

9.4.1 L1 Cache Control and Status Register 0 (L1CSR0)

The L1 cache control and status register 0 (L1CSR0) is a 32-bit register used for general control of the data cache as well as providing general control over disabling ways in both caches. The L1CSR0 register is accessed using a **mfspr** or **mtspr** instruction. The SPR number for L1CSR0 is 1010 in decimal. The L1CSR0 register is shown in Figure 9-4.

SPR 1010 Access: Read/Write

	0 3	4 7	8 10	11	12 13	14	15
R W	WID	WDD	—	DCWM	DCWA	—	DCECE
Reset	All zeros						

	16	17	18 19	20	21	22	23	24	25 26	27	28	29	30	31
R W	DCEI	—	DCEDT	DCSLC	DCUL	DCLO	DCLFC	DCLOA	DCEA	—	DCBZ32	DCABT	DCINV	DCE
Reset	All zeros													

Figure 9-4. L1 Cache Control and Status Register 0 (L1CSR0)

The L1CSR0 bits are described in Table 9-1.

Table 9-1. L1CSR0 Field Descriptions

Bits	Name	Description
0–3	WID	Way Instruction Disable. 0 The corresponding way in the instruction cache is available for replacement by instruction miss line fills. 1 The corresponding way instruction cache is not available for replacement by instruction miss line fills. • Bit 0 corresponds to way 0. • Bit 1 corresponds to way 1. • Bit 2 corresponds to way 2. • Bit 3 corresponds to way 3. The WID bits may be used for locking ways of the instruction cache and also are used to determine the replacement policy of the instruction cache.
4–7	WDD	Way Data Disable. 0 The corresponding way in the data cache is available for replacement by data miss line fills. 1 The corresponding way in the data cache is not available for replacement by data miss line fills. • Bit 4 corresponds to way 0. • Bit 5 corresponds to way 1. • Bit 6 corresponds to way 2. • Bit 7 corresponds to way 3. The WDD bits may be used for locking ways of the data cache and also are used to determine the replacement policy of the data cache.
8–10	—	Reserved[1]
11	DCWM	Data Cache Write Mode 0 Data Cache operates in writethrough mode 1 Data Cache operates in copyback mode When set to writethrough mode, the “W” page attribute from the MMU is ignored and all writes are treated as writethrough required. When set, write accesses are performed in copyback mode unless the “W” page attribute from the MMU is set.

Table 9-1. L1CSR0 Field Descriptions (continued)

Bits	Name	Description
12–13	DCWA	Data Cache Write Allocation Policy 00 Cache line allocation on a cacheable write miss is disabled 01 Cache line allocation on a cacheable copyback write miss is enabled 10 Cache line allocation on a cacheable copyback or writethrough write miss is enabled 11 Reserved This field also controls merging of store data into the linefill buffer while a cache linefill is in progress. Store data will not be merged when write allocation is disabled. If DCWA is non-zero, store data merging is enabled regardless of the type (writethrough/copyback) of write.
14	—	Reserved[1]
15	DCECE	Data Cache Error Checking Enable 0 Error Checking is disabled 1 Error Checking is enabled
16	DCEI	Data Cache Error Injection 0 Cache Error Injection is disabled 1 parity errors will be purposefully injected into every byte subsequently written into the cache. The parity bit of each 8-bit data element written will be inverted. This includes writes due to store hits as well as writes due to cache line refills. DCEI will cause injection of errors regardless of the setting of DCECE, although reporting of errors will be masked while DCECE = 0.
17	—	Reserved[1]
18–19	DCEDT	Data Cache Error Detection Type 00 Parity Error Detection is selected for both the tag and data arrays 01 EDC Error Detection is selected for the tag array and parity is selected for the data arrays 1x Reserved
20	DCSLC	Data Cache Snoop Lock Clear 0 Snoop has not invalidated a locked line 1 Snoop has invalidated a locked line Indicates a cache line lock was cleared by a snoop operation which caused an invalidation. This bit is set by hardware and will remain set until cleared by software writing 0 to this bit location.
21	DCUL	Data Cache Unable to Lock Indicates a lock set instruction was not effective in locking a cache line. This bit is set by hardware on an “unable to lock” condition (other than lock overflows) and will remain set until cleared by software writing 0 to this bit location.
22	DCLO	Data Cache Lock Overflow Indicates a lock overflow (overlocking) condition occurred. This bit is set by hardware on an “overlocking” condition and will remain set until cleared by software writing 0 to this bit location.
23	DCLFC	Data Cache Lock Bits Flash Clear When written to a ‘1’, a cache lock bits flash clear operation is initiated by hardware. Once complete, this bit is reset to ‘0’. Writing a ‘1’ while a flash clear operation is in progress will result in an undefined operation. Writing a ‘0’ to this bit while a flash clear operation is in progress will be ignored. Cache Lock Bits Flash Clear operations require approximately 134 cycles to complete. Clearing occurs regardless of the enable (DCE) value.
24	DCLOA	Data Cache Lock Overflow Allocate Set by software to allow a lock request to replace a locked line when a lock overflow situation exists. 0 Indicates a lock overflow condition will not replace an existing locked line with the requested line 1 Indicates a lock overflow condition will replace an existing locked line with the requested line

Table 9-1. L1CSR0 Field Descriptions (continued)

Bits	Name	Description
25–26	DCEA	Data Cache Error Action 00 Error Detection causes Machine Check exception. 01 Error Detection causes Correction/Auto-invalidation. No machine check is generated for uncorrectable errors unless the cache line was locked and invalidated or is dirty. Dirty lines are not auto-invalidated. In EDC mode, correction is performed for single-bit tag errors, single-bit lock errors, and single or multi-bit dirty errors. In parity mode, tag and lock errors will result in invalidation of clean lines. In parity mode, tag and lock errors will result in invalidation of clean lines. For both modes, correction is performed for data errors by reloading of the line. 1x Reserved
27	—	Reserved[1]
28	DCBZ32	Data Cache **dcba**, **dcbz** operation length 0 **dcba**, **dcbz** operations operate on an entire cache line 1 **dcba**, **dcbz** operations operate on 32bytes of a cache line This bit is implemented for forward compatibility. Since cache lines are 32 bytes, this bit is ignored for **dcba**, **dcbz** operations
29	DCABT	Data Cache Operation Aborted Indicates a cache Invalidate or a Cache Lock Bits Flash Clear operation was aborted prior to completion. This bit is set by hardware on an aborted condition, and will remain set until cleared by software writing 0 to this bit location.
30	DCINV	Data Cache Invalidate 0 No cache invalidate 1 Cache invalidation operation When written to a ‘1’, a cache invalidation operation is initiated by hardware. Once complete, this bit is reset to ‘0’. Writing a ‘1’ while an invalidation operation is in progress will result in an undefined operation. Writing a ‘0’ to this bit while an invalidation operation is in progress will be ignored. Cache invalidation operations require approximately 134 cycles to complete. Invalidation occurs regardless of the enable (DCE) value. During cache invalidations, the parity check bits are written with a value dependent on the DCEDT selection. DCEDT should be written with the desired value for subsequent cache operation when DCINV is set to ‘1’ for proper operation of the cache.
31	DCE	Data Cache Enable 0 Cache is disabled 1 Cache is enabled When disabled, cache lookups are not performed for normal load or store accesses, or for snoop requests. Other L1CSR0 cache control operations are still available. Also, operation of the store buffer is not affected by DCE.

[1] These bits are not implemented and should be written with zero for future compatibility.

9.4.2 L1 Cache Control and Status Register 1 (L1CSR1)

The L1 cache control and status register 1 (L1CSR1), shown in Figure 9-5, is a 32-bit register used for general control of the instruction cache. The L1CSR1 register is accessed using an **mfspr** or **mtspr** instruction. The SPR number for L1CSR1 is 1011 in decimal.

SPR 1011 Access: Read/Write

	0–14	15	16	17	18–19	20	21	22	23	24	25–26	27–28	29	30	31
R W	—	ICECE	ICEI	—	ICEDT	—	ICUL	ICLO	ICLFC	ICLOA	ICEA	—	ICABT	ICINV	ICE
Reset	All zeros														

Figure 9-5. L1 Cache Control and Status Register 1 (L1CSR1)

The L1CSR1 bits are described in Table 9-2.

Table 9-2. L1CSR1 Field Descriptions

Bits	Name	Description
0–14	—	Reserved
15	ICECE	Instruction Cache Error Checking Enable 0 Error Checking is disabled 1 Error Checking is enabled
16	ICEI	Instruction Cache Error Injection Enable 0 Cache Error Injection is disabled 1 When ICEDT = 00, parity errors are purposefully injected into every byte subsequently written into the cache. The parity bit of each 8-bit data element written is inverted on cache linefills. When ICEDT = 01, a double-bit error is injected into each double word written into the cache by inverting the two uppermost parity check bits (p_chk[0:1]). ICEI causes injection of errors regardless of the setting of ICECE, although reporting of errors is masked when ICECE = 0.
17	—	Reserved
18–19	ICEDT	Instruction Cache Error Detection Type 00 Parity Error Detection is selected for both the tag and data arrays 01 EDC Error Detection is selected 1x Reserved
20	—	Reserved
21	ICUL	Instruction Cache Unable to Lock Indicates a lock set instruction was not effective in locking a cache line. This bit is set by hardware on an “unable to lock” condition (other than lock overflows) and remains set until cleared by software writing 0 to this bit location.
22	ICLO	Instruction Cache Lock Overflow Indicates a lock overflow (overlocking) condition occurred. This bit is set by hardware on an “overlocking” condition and remains set until cleared by software writing 0 to this bit location.
23	ICLFC	Instruction Cache Lock Bits Flash Clear When written to a 1, a cache lock bits flash clear operation is initiated by hardware. Once complete, this bit is reset to 0. Writing a 1 while a flash clear operation is in progress will result in an undefined operation. Writing a 0 to this bit while a flash clear operation is in progress will be ignored. Cache Lock Bits Flash Clear operations require approximately 134 cycles to complete. Clearing occurs regardless of the enable (ICE) value.

Table 9-2. L1CSR1 Field Descriptions (continued)

Bits	Name	Description
24	ICLOA	Instruction Cache Lock Overflow Allocate Set by software to allow a lock request to replace a locked line when a lock overflow situation exists. 0 Indicates a lock overflow condition will not replace an existing locked line with the requested line 1 Indicates a lock overflow condition will replace an existing locked line with the requested line
25–26	ICEA	Instruction Cache Error Action 00 Error Detection causes machine check exception. 01 Error Detection causes correction/auto-invalidation. No machine check is generated unless a locked line is invalidated. In EDC mode, correction is performed for single-bit tag and lock errors, and lines with multi-bit tag or lock errors are invalidated. In parity mode, tag or lock errors will result in invalidation of lines. For both modes, correction is performed for single or multi-bit data errors by reloading of the line. 1x Reserved
27–28	—	Reserved
29	ICABT	Instruction Cache Operation Aborted Indicates a Cache Invalidate or a Cache Lock Bits Flash Clear operation was aborted prior to completion. This bit is set by hardware on an aborted condition, and will remain set until cleared by software writing 0 to this bit location.
30	ICINV	Instruction Cache Invalidate 0 No cache invalidate 1 Cache invalidation operation When written to a 1, a cache invalidation operation is initiated by hardware. Once complete, this bit is reset to 0. Writing a 1 while an invalidation operation is in progress will result in an undefined operation. Writing a 0 to this bit while an invalidation operation is in progress will be ignored. Cache invalidation operations require approximately 134 cycles to complete. Invalidation occurs regardless of the enable (ICE) value. During cache invalidations, the parity check bits are written with a value dependent on the ICEDT selection. ICEDT should be written with the desired value for subsequent cache operation when ICINV is set to '1' for proper operation of the cache.
31	ICE	Instruction Cache Enable 0 Cache is disabled 1 Cache is enabled When disabled, cache lookups are not performed for instruction accesses. Other L1CSR1 cache control operations are still available and are not affected by ICE.

9.4.3 L1 Cache Configuration Register 0 (L1CFG0)

The L1 cache configuration register 0 (L1CFG0) is a 32-bit read-only register that provides information about the configuration of the e200z7 L1 data cache design. The contents of the L1CFG0 register can be read using a **mfspr** instruction. Figure 9-6 shows the L1CFG0 register.

SPR 515

Access: Read only

	0 1	2	3	4	5 6	7 8	9 10	11	12	13 … 20	21 … 31
R	CARCH	CWPA	CFAHA	DCFISWA	—	DCBSIZE	DCREPL	DCLA	DCECA	DCNWAY	DCSIZE
W											
Reset	0 0	1	0	1	0 0	0 0	1 0	1	1	0 0 0 0 0 0 1 1	0 0 0 0 0 0 1 0 0 0 0

Figure 9-6. L1 Cache Configuration Register 0 (L1CFG0)

The L1CFG0 bits are described in Table 9-3.

Table 9-3. L1CFG0 Field Descriptions

Bits	Name	Description
0–1	CARCH	Cache Architecture 00 The cache architecture is Harvard
2	CWPA	Cache Way Partitioning Available 1 The caches support partitioning of way availability for I/D accesses
3	DCFAHA	Data Cache Flush All by Hardware Available 0 The data cache does not support Flush All in Hardware
4	DCFISWA	Data Cache Flush/Invalidate by Set and Way Available 1 The data cache supports flushing/invalidation by Set and Way via the L1FINV0 spr
5–6	—	Reserved—read as zeros
7–8	DCBSIZE	Data Cache Block Size 00 The data cache implements a block size of 32 bytes
9–10	DCREPL	Data Cache Replacement Policy 10 The data cache implements a pseudo-round-robin replacement policy
11	DCLA	Data Cache Locking unit Available 1 The data cache implements the line locking unit
12	DCECA	Data Cache Error Checking Available 1 The data cache implements error checking
13–20	DCNWAY	Data Cache Number of Ways 0x03 The data cache is 4-way set-associative
21–31	DCSIZE	Data Cache Size 0x010 The size of the data cache is 16 KB

9.4.4 L1 Cache Configuration Register 1 (L1CFG1)

The L1 cache configuration register 1 (L1CFG1) is a 32-bit read-only register that provides information about the configuration of the e200z760n3 L1 instruction cache design. The contents of the L1CFG1 register can be read using a **mfspr** instruction. Figure 9-7 shows the L1CFG1 register.

SPR 516 Access: Read only

	0–3	4	5–6	7–8	9–10	11	12	13–20	21–31
R	—	ICFISWA	—	ICBSIZE	ICREPL	ICLA	ICECA	ICNWAY	ICSIZE
W									
Reset	0 000	1	0 0	0 0	1 0	1	1	0 0 0 0 0 0 1 1	0 0 0 0 0 0 1 0 0 0 0

Figure 9-7. L1 Cache Configuration Register 1 (L1CFG1)

The L1CFG1 bits are described in Table 9-4.

Table 9-4. L1CFG1 Field Descriptions

Bits	Name	Description
0–3	—	Reserved—read as zeros
4	ICFISWA	Instruction Cache Flush/Invalidate by Set and Way Available 1 The instruction cache supports invalidation by Set and Way via the L1FINV1 spr
5–6	—	Reserved—read as zeros
7–8	ICBSIZE	Instruction Cache Block Size 00 The instruction cache implements a block size of 32 bytes
9–10	ICREPL	Instruction Cache Replacement Policy 10 The instruction cache implements a pseudo-round-robin replacement policy
11	ICLA	Instruction Cache Locking unit Available 1 The instruction cache implements the line locking unit
12	ICECA	Instruction Cache Error Checking Available 1 The instruction cache implements error checking
13–20	ICNWAY	Instruction Cache Number of Ways 0x03 The instruction cache is 4-way set-associative
21–31	ICSIZE	Instruction Cache Size 0x010 The size of the data cache is 16 KB

9.5 Data Cache Software Coherency

Data cache coherency is supported through software operations to invalidate, flush dirty lines to memory, or invalidate dirty lines. The data cache may operate in either write-through or copyback modes, and in conjunction with a MMU, may designate certain accesses as write-through or copyback. Data cache misses force the push and store buffers to empty prior to performing the access to ensure coherency.

9.6 Address Aliasing

Each cache is virtually indexed and physically tagged, thus the problems associated with potential cache synonyms due to effective address aliasing are eliminated, unless 1 KB or 2 KB pages are used. If 1 KB or 2 KB pages are used and multiple virtual addresses are mapped to the same physical address, the low order virtual address bits used to index the cache (A[20–21] for 1 KB pages, A20 for 2 KB pages) must be the same for each of the virtual pages, and these index bit(s) must match the corresponding physical address bit(s) value. For example, if logical pages X and Y map to physical page P, then X, Y, and P must have the same values of A[20–21] for 1 KB pages, and A20 for 2 KB pages. Note that this limitation should be already met because of the requirements on 1 KB and 2 KB page usage mandated by Section 10.2.6, “Restrictions on 1-KB and 2-KB Page Size Usage.”

9.7 Cache Operation

This section contains the following subsections, which discuss cache operation in detail:

- Section 9.7.1, “Cache Enable/Disable”
- Section 9.7.2, “Cache Fills”

- Section 9.7.3, "Cache Line Replacement"
- Section 9.7.4, "Cache Miss Access Ordering"
- Section 9.7.5, "Cache-Inhibited Accesses"
- Section 9.7.6, "Guarded Accesses"
- Section 9.7.7, "Cache-Inhibited Guarded Accesses"
- Section 9.7.8, "Cache Invalidation"
- Section 9.7.9, "Cache Flush/Invalidate by Set and Way"

9.7.1 Cache Enable/Disable

The caches are enabled or disabled by using the cache enable bits L1CSR0[DCE] and L1CSR1[ICE] respectively. Cache enable bits are cleared by power-on reset or normal reset, disabling the caches.

When a cache is disabled, the cache tag status bits are ignored, and the cache is not accessed for snoops, normal loads, stores, or instruction fetches. All normal accesses are propagated to the system bus as single-beat (non-burst) transactions.

Note that the state of the Cache Inhibited access attribute (the I bit) remains independent of the state of L1CSR0[DCE] and L1CSR1[ICE]. Disabling a cache does not affect the translation logic in the memory management unit. Translation attributes are still used when generating attribute information on the system buses.

The store buffer is still available for use even when the data cache is disabled.

Altering the DCE or ICE bit must be preceded by an **isync** and **msync** to prevent the cache from being disabled or enabled in the middle of a data or instruction access. In addition, the cache may need to be globally flushed before it is disabled to prevent coherency problems when it is re-enabled.

All cache operations are affected by disabling the cache. Cache management instructions (except for **mtspr** L1FINV0,1 and **mtspr** L1CSR0,1) do not affect a cache when it is disabled.

9.7.2 Cache Fills

Cache line fills are requested when a cacheable load or instruction miss occurs. Cacheable store misses only allocate cache lines if data cache write allocation is enabled for the type of store being performed. In addition, no allocation is performed for a write-through store when the store buffer is disabled.

The cache line fill is performed critical double word first on the bus that is using a burst access. The critical double word is forwarded to the requesting unit before being written to the cache, thus minimizing stalls due to fill delays. Cache line fills load a four double word linefill buffer, and updates to the cache array are performed as half-lines are received.

Read accesses may hit in the line buffer and data supplied from the buffer to the CPU. On writes which hit to the buffer address, when write allocation is disabled, the writes stall until the cache fill has been completed. When write allocation is enabled, these writes update the linefill buffer if the buffer is being filled due to a store miss only; otherwise the write also stalls until the linefill completes.

Data may be streamed to the CPU as it arrives from the bus if a corresponding request is pending. In addition, the cache supports hit under fill, allowing subsequent CPU accesses to be satisfied by cache hits while the remainder of the line fill completes. This non-blocking capability improves performance by hiding a portion of the line fill latency when data already in the cache or linefill buffer is subsequently requested by the CPU.

The cache supports up to three outstanding misses and forwards these miss requests to the BIU. Miss data is always returned from the BIU to the cache in-order.

Cache fill operations are performed as wrapping bursts on the system bus. If an error response is received on any element of the burst, the burst will be terminated, and the cache line will be marked invalid.

If one or more store hit updates occur to the linefill buffer during allocation of a line for a store miss and a subsequent error response is received during the linefill, the original store miss access and each individual hitting store access are performed on the system bus as if they were non-allocating. In this case, an async machine check exception is signaled for the linefill.

9.7.3 Cache Line Replacement

On a cache miss, the cache controller uses a pseudo-round-robin replacement algorithm to determine which cache line will be selected to be replaced. There is a single replacement counter for each cache. The replacement algorithm acts as follows: On a miss, if the replacement pointer is pointing to a way that is not enabled for replacement (the selected line or way is locked), it is incremented until an available way is selected (if any). After a cache line is successfully filled without error, the replacement pointer increments to point to the next cache way. If no way is available for the replacement, the access is treated as a single beat access and no cache linefill occurs.

Lines selected for replacement which are dirty (modified) must be copied back to main memory. This is performed by first storing the replaced line in a 32-byte push buffer while the missed data is fetched. After filling the new line, the contents of the buffer are written to memory beginning with double word 0.

Each replacement counter is initialized to point to way 0 on a reset or on a respective cache invalidate all operation. A replacement counter may also be set to a specific value via a L1FINV0/L1FINV1 command.

9.7.4 Cache Miss Access Ordering

Cacheable cache misses may be processed out-of-order by the e200z760n3. Load misses which are not cache-inhibited are allowed to bypass buffered stores and push buffer pushes as long as no address alias exists. Alias checking is performed by comparing the index of the load with the index of each buffered store and push. If no alias match exists, the load is allowed to bypass buffered stores and pushes, regardless of the attributes associated with those stores. Load misses are performed in-order with respect to other load misses. Store accesses do not bypass loads. Stores are not necessarily performed in order from the point of view of the memory system, since a store miss may cause a linefill to satisfy the store prior to previously buffered stores being completed, as long as no aliasing occurs.

Memory access ordering must be enforced by software where required, using the **mbar** and/or **msync** instructions according to the Power Architecture storage ordering rules.

9.7.5 Cache-Inhibited Accesses

When the Cache-inhibited attribute is indicated by translation and a cache miss occurs, all accesses are performed as single beat transactions on the system bus. Cache-inhibited status is ignored on all cache hits. For cache-inhibited load access misses, the processor termination is withheld for the load until the store buffer has been flushed of all entries, the push buffer has been emptied, and the load has completed to memory. Cache-inhibited store accesses that are not marked as Guarded are placed in the store buffer (when enabled) and the processor termination occurs when the store buffer entry is allocated. (see Section 9.9, "Push and Store Buffers")

9.7.6 Guarded Accesses

When the Guarded attribute is indicated by translation and a cache miss occurs, the access does not proceed on the external bus until all previously initiated demand-accesses have been terminated to the processor without error. Buffered stores are considered terminated to the processor when they are placed into the store buffer. Guarded load misses that are not cache-inhibited are allowed to bypass buffered stores and push buffer pushes as long as no address alias exists, regardless of whether a buffered store is guarded. Guarded stores do not allocate cache lines on a miss. Instead, if the access is not cache-inhibited, they are buffered in the store buffer (when enabled), regardless of whether or not they are write through required (regardless of W bit or L1CSR0[DCWM] values), and performed as single-beat accesses on the bus.

9.7.7 Cache-Inhibited Guarded Accesses

When the Cache-inhibited and Guarded attributes are indicated by translation and a cache miss occurs, accesses are performed as single beat transactions on the system bus. Cache-inhibited status is normally ignored on all cache hits. Cache-inhibited status for write-through stores that are also guarded is not ignored, however. For cache-inhibited guarded access misses, or for cache-inhibited guarded write-through store hits, the processor termination is withheld until the store buffer has been flushed of all entries, the push buffer has been emptied, and the access has completed to memory (see Section 9.9, "Push and Store Buffers"). Cache-inhibited guarded stores with W = 0 or L1CSR0[DCWM] = 1, which hit ignore Cache-inhibited and Guarded status.

9.7.8 Cache Invalidation

The e200z7 supports full invalidation of the caches under software control. The cache may be invalidated through the L1CSR0[DCINV] and L1CSR1[ICINV] cache invalidate control bits. This function is available even when a cache is disabled.

Reset does not invalidate a cache automatically. Software must use the DCNV/ICINV control for invalidation after a reset. Proper use of this bit is to determine that it is clear and then set it with a pair of **mfspr mtspr** operations. A 0-to-1 transition on DCNV/ICINV causes a flash invalidation to be initiated, which lasts for multiple (approximately 134) CPU cycles. Once set, the DCNV/ICINV bit is cleared by hardware after the operation is complete. It remains set during the invalidation interval and may be tested by software to determine when the operation has completed. An **mtspr** operation to L1CSR0/1 that attempts to change the state of DCNV/ICINV during invalidation does not affect the state of that bit.

To properly generate the tag parity/check bits during the invalidation process, the error detection type control located in L1CSR0[DCEDT]/L1CSR1[ICEDT] should be configured properly at the time that the invalidation operation is initiated. A subsequent change to the error detection type control requires a new invalidation to avoid improper interpretation of previously stored tag parity/check bits.

During the process of performing the invalidation, a cache does not respond to accesses that are not snoop accesses and remains busy. Interrupts may still be recognized and processed, potentially aborting the invalidation operation. When this occurs, L1CSR0,1[ABT] is set to indicate unsuccessful completion of the operation. Software should read the L1CSR0/L1CSR1 register to determine that the operation has completed (L1CSR0,1[CINV] cleared), and then check the status of the L1CSR0,1[ABT] to determine completion status.

NOTE

> Note that while this implementation of the e200z7 stalls further instruction execution during this invalidation interval, this is not guaranteed across all implementations. Thus, software should be written using these guidelines.

Individual cache lines may be invalidated using the **icbi**, **dcbi**, or **dcbf** instructions. These instructions require the respective cache to be enabled in order to operate normally.

9.7.9 Cache Flush/Invalidate by Set and Way

The e200z7 supports cache flushing under software control. The caches may be flushed and/or invalidated by index and way through a **mtspr l1finv{0,1}** instruction.

The L1 flush and invalidate control registers (L1FINV0, L1FINV1) are 32-bit SPRs used to select a cache set and way to be flushed/invalidated. No tag match is required. This function is available even when a cache is disabled. L1FINV0 is used for data cache operations, while L1FINV1 is used for instruction cache operations.

9.7.9.1 L1 Flush/Invalidate Register 0 (L1FINV0)

The SPR number for L1FINV0 is 1016 in decimal. The L1FINV0 register is shown in Figure 9-8.

SPR 1016

Access: Read/Write

	0 5	6 7	8 19	20 26	27 29	30 31
R W	—	CWAY	—	CSET	—	CCMD
Reset	All zeros					

Figure 9-8. L1 Flush/Invalidate Register 0 (L1FINV0)

The L1FINV0 bits are described in Table 9-5.

Table 9-5. L1FINV0 Field Descriptions

Bits	Name	Description
0–5	—	Reserved[1] for way extension
6–7	CWAY	Cache Way Specifies the data cache way to be selected
8–19	—	Reserved[1] for set extension
20–26	CSET	Cache Set Specifies the cache set to be selected
27–29	—	Reserved[1] for set/command extension
30–31	CCMD	Cache Command 00 The data contained in this entry is invalidated without flushing 01 The data contained in this entry is flushed if dirty and valid without invalidation 10 The data contained in this entry is flushed if dirty and valid and then is invalidated 11 Reset way replacement pointer to the way indicated by CWAY

[1] These bits are not implemented and should be written with zero for future compatibility.

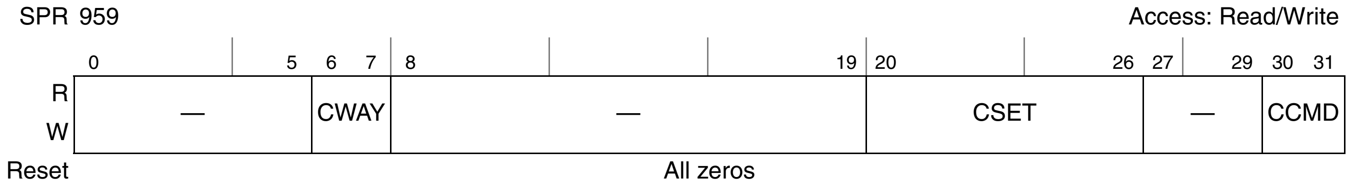
For cache flush operations, if a transfer error occurs on a data cache line flush, the push of the remaining portion of the cache line is aborted; the line remains marked dirty and valid; and a machine check condition is signaled.

For flush and flush with invalidation operations, data parity errors do not abort a flush to memory, but a machine check is generated at the completion of the flush. In both cases, the cache line is left unchanged. For flush with invalidation operations to clean lines, tag parity errors and data parity errors are ignored, and the line is invalidated. Note that only the line indicated by CSET and CWAY is checked for errors; lines in the other ways are ignored.

For invalidation without flushing operations, tag parity errors, data parity errors, and dirty-bit parity errors are ignored, and the line is invalidated.

9.7.9.2 L1 Flush/Invalidate Register 1 (L1FINV1)

The SPR number for L1FINV1 is 959 in decimal. The L1FINV1 register is shown in Figure 9-9.



SPR 959 Access: Read/Write

	0–5	6–7	8–19	20–26	27–29	30–31
R W	—	CWAY	—	CSET	—	CCMD
Reset	All zeros					

Figure 9-9. L1 Flush/Invalidate Register 1 (L1FINV1)

The L1FINV1 bits are described in Table 9-6.

Table 9-6. L1FINV1 Field Descriptions

Bits	Name	Description
0–5	—	Reserved[1] for way extension
6–7	CWAY	Cache Way Specifies the instruction cache way to be selected
8–19	—	Reserved[1] for set extension
20–26	CSET	Cache Set Specifies the instruction cache set to be selected
27–29	—	Reserved[1] for set/command extension
30–31	CCMD	Cache Command 00 The data contained in this entry is invalidated 01 Reserved 10 Reserved 11 Reset way replacement pointer to the way indicated by CWAY

[1] These bits are not implemented and should be written with zero for future compatibility.

9.8 Cache Parity and EDC Protection

Cache parity is supported for both the tag and data arrays of each cache. Six parity check bits are provided for each tag entry for the tag arrays of both caches to support multi-bit error detection (EDC), and redundant dirty bits are provided in the data cache to provide dirty-bit parity checking without requiring a read-modify-write operation when the dirty bit is set. Redundant lock bits are provided as well for both the Icache and the Dcache. Byte parity is supported for the data arrays of the data cache. Eight parity check bits are provided for each double word in the data arrays of the ICache, which can be used either standard byte parity checking (single-bit error detection) or for multibit error detection (EDC—DED, double error detection). When utilizing EDC protection, many multibit errors are also detected.

Parity and EDC checking is controlled by the L1CSR0[DCECE], L1CSR0[DCEDT], L1CSR1[ICECE], and L1CSR1[ICEDT] control fields. When error checking is enabled, checking is performed on each cache access, whether for lookup, snoop lookup, or for dirty line replacement. Parity or EDC errors are not signaled by the respective cache when cache error checking is disabled for that cache (L1CSR0[DCECE] or L1CSR1[ICECE] = 0).

For normal cache lookups due to instruction fetching, loads, or stores, if an uncorrectable tag parity or EDC error is detected on any portion of the accessed tags, a parity error is signaled, regardless of whether a cache hit or miss occurs. Otherwise, if a cache hit for a load occurs and a data parity error is detected on any portion of the accessed double word of data, a parity error is also signaled. Data parity errors are ignored for store hits, since the parity is updated for the data being stored. Data parity errors are ignored for misses unless the replacement line is dirty or incurs a dirty bit parity error, since the parity will be updated for the new linefill data being stored.

Signaling of a parity error may not cause an exception to occur, depending on the error detection action to be taken. Instead, a correction/auto-invalidation cycle may be performed.

A dirty line push is not generated for a dirty line replacement that incurs an uncorrectable tag parity or EDC error. In this case, a machine check is generated, but no push was requested to the external bus, and the cache line is left unchanged. For dirty line pushes from the data cache, accessing the data arrays for the push data may occur after the burst write has been requested on the external bus. Therefore, a push of dirty data may actually push data that contains a parity error. A machine check is signaled, but the burst is not aborted, and the line is invalidated and replaced.

Dirty bit parity is checked when invalidation or replacement operations are required. If a dirty parity error is detected on a cache line replacement, in correction/auto-invalidation mode, it is ignored, and the line is pushed normally. In machine check mode, a machine check exception is signaled, indicating a tag parity error. Dirty status or dirty parity errors prevent the auto-invalidation of cache lines with tag parity or EDC errors. If a dirty parity error occurs in correction/auto-invalidation mode, the line is assumed to be dirty. If correction/auto-invalidation is enabled, the error is corrected by re-writing all three dirty bits to 1. This implies that a single or multi-bit error that sets one or more dirty bits from an initially cleared state causes the line to appear dirty. This should not cause a functional issue, however, because the only result is that a clean but coherent line may be pushed on a flush or replacement in correction/auto-invalidation mode.

Regardless of the error action mode indicated by DCEA/ICEA, lock bit parity errors do not signal an exception for normal hits without a tag parity error. If correction/auto-invalidation is enabled, on each cache lookup operation, a single-bit lock error that is detected in one or more ways is corrected by rewriting all lock bits to the correct state. Uncorrectable lock errors remain unchanged. For cache hits without a tag parity/EDC error, all lock parity errors are ignored. Lock parity errors on a cacheable miss (after a correction attempt if correction/auto-invalidation is enabled) result in the line(s) being invalidated if clean and a machine check being generated. A new line is not allocated, and the lock bits are not updated on the invalidation. Lock bit parity errors are ignored for non-cacheable accesses.

Signaling of a parity error or EDC error may cause a Machine Check exception to occur and one or more syndrome bits to be set in the machine check syndrome register. However, it may instead result in a correction/auto-invalidation operation and not in an exception being signaled. Both may also occur, depending on the error action control setting in the appropriate cache control register. Refer to Section 9.8.1, “Cache Error Action Control,” for details of the cache error action controls. Refer to Section 7.6.2, “Machine Check Interrupt (IVOR1),” and to Section 2.4.7, “Machine Check Syndrome Register (MCSR),” for a description of Machine Check conditions.

9.8.1 Cache Error Action Control

The L1CSR0[DCEA] and L1CSR1[ICEA] control fields allow the selection of several policies to apply when errors are detected during a cache lookup. They are described in the following subsections.

9.8.1.1 L1CSR0[DCEA]/L1CSR1[ICEA] = 00, Machine Check Generation on Error

Selection of the machine check generation on error policy allows all errors to be processed by software. Parity or EDC errors that may result in incorrect operation cause a machine check condition. To be recoverable, the machine check handler must not incur another parity or EDC error during the initial portion of the machine check handler. Parity/EDC errors do not generate a machine check exception for cache-inhibited accesses.

If machine check generation on error is enabled (L1CSR0[DCEA]/ L1CSR1[ICEA] = 00) and an uncorrectable parity or EDC error is detected on any portion of the accessed tags for a cacheable load or store access, a machine check is reported, regardless of whether a cache hit or miss occurs. If a cache hit occurs and a parity or EDC error is detected on any portion of the accessed double word of data for a load or an instruction access, a machine check is also reported. For store accesses, data parity errors are ignored. Lock or dirty parity errors on a cacheable miss cause a machine check to be reported, indicating a lock error and/or a tag parity error. Dirty parity errors on a cache hit for a reservation instruction (**lwarx**, **stwcx.**, etc.) result in a machine check and indicate a tag parity error. If a miss occurs and a tag parity/EDC error is detected on a lookup for a cacheable reservation instruction (**lwarx**, **stwcx**., etc.), it is ignored if the line is clean. If the line is dirty or a dirty parity error occurs, a machine check is generated and the reservation access is not run externally. Cache inhibited reservation accesses ignore all parity/EDC errors.

9.8.1.2 L1CSR0[DCEA]/L1CSR1[ICEA] = 01, Correction/Auto-invalidation on Error

The correction/auto-invalidation on error policy attempts to cause most parity and EDC errors to be transparently handled by correcting lines with single-bit tag errors, invalidating lines with uncorrectable tag errors or with data errors, and then causing cache refills to reload correct data from memory, without generation of exceptions. Exceptions are only generated when invalidations could cause or would cause a change in correct behavior, such as changing the locked status of a line, or invalidating potentially dirty data. Parity/EDC errors do not generate invalidations that could cause a machine check exception for cache-inhibited accesses, however.

When using EDC protection for the cache tags (L1CSR0[DCEDT]/L1CSR1[ICEDT] = 01), single-bit tag errors are corrected by the cache hardware during a correction/auto-invalidation cycle. Clean unlocked lines with multi-bit errors are invalidated on cache hits, with no machine check signaled. Clean locked lines with uncorrectable tag errors are invalidated on cache misses, and a machine check is signaled. When operating with only parity protection for the cache tags (L1CSR0[DCEDT]/L1CSR1[ICEDT] = 00), clean unlocked cache entries with detectable tag errors are invalidated rather than corrected by the cache hardware during a correction/auto-invalidation cycle.

Note that since the data arrays have a higher probability of incurring an error than the tag arrays, due to the relative storage capacities, most errors are transparently corrected, even if they are double-bit or multi-bit errors. Using write-through mode for critical data ensures that invalidation or refills are able to recover from errors transparently in most cases.

9.8.1.2.1 Instruction Cache Errors

If correction/auto-invalidation on error is enabled (L1CSR1[ICEA] = 01) and an error is detected on any portion of the accessed tags or data for an access, a correction/auto-invalidation cycle is inserted, regardless of whether a cache hit or miss occurs. During this cycle, any tag entry with a single-bit tag or lock error is corrected if possible (correction is not possible when operation with only parity protection for the tags), and re-written to correct the stored error. Tag entries with uncorrectable errors are invalidated if unlocked or are invalidated if a cache miss occurs after a correction/auto-invalidation cycle regardless of locked status. If a locked line is invalidated, a machine check occurs, no replacement occurs, and the locked status remains set for the invalidated line(s) to assist software in determining the location of the error(s).

Following the correction/auto-invalidation cycle, a re-lookup is performed for the access. If a cache hit occurs on a way without a tag parity/EDC error, and a parity or EDC error is detected on any portion of the accessed double word of data, a miss is forced, and the same line is refilled from system memory, retaining the existing lock status. The replacement pointer for the cache is not updated in these circumstances. If a cache hit occurs on a way without a tag parity/EDC error, parity or EDC errors on all other lines are ignored, and no invalidations for those lines occurs.

For all cases of invalidations, if any line which was locked or incurred a lock error was invalidated, a machine check also occurs, even though auto-invalidation is selected. Invalidation is not blocked for locked lines or lines with lock parity errors on cache misses. The lock bits remain unmodified by the invalidation operation to allow for potential software recovery.

If a refill of a locked line due to a data parity/EDC error encounters an external bus error during the linefill, a machine check is generated, the line is invalidated, and the lock bits remain set.

9.8.1.2.2 Data Cache Errors

If correction/auto-invalidation on error is enabled (L1CSR0[DCEA] = 01) and an error is detected on any portion of the accessed tags, or if a lock or dirty parity error is detected, an invalidation/correction cycle is inserted, regardless of whether a cache hit or miss occurs. Following the invalidation/correction cycle, a re-lookup is performed for the access. During the correction/auto-invalidation cycle, any tag entry with a tag or lock error is corrected if possible, and re-written to correct the stored error. Tag entries with uncorrectable errors are invalidated if the line is clean and unlocked, or if the line is clean and a miss will occur after the re-lookup, regardless of lock status. Dirty parity errors are corrected by setting all dirty bits to ‘1’. Dirty lines and lines with a dirty parity error are not invalidated.

Following the correction/auto-invalidation cycle, a re-lookup is performed for the access. If a cache hit occurs on a way without a tag parity/EDC error, and a parity error is detected on any portion of the accessed double word of data for a load, if the line is clean, a miss is forced and the line is refilled from system memory, retaining the existing lock status. The replacement pointer for the cache is not updated in these circumstances. All other clean unlocked lines with uncorrectable tag errors will have been invalidated during the correction/auto-invalidation cycle if one was initially needed. Tag parity/EDC errors on lines which were not invalidated earlier due to lock or dirty status will be ignored since a cache hit occurs. For stores, parity errors on data are ignored, and no invalidation or refill of any lines will occur on a hit to a way without a tag parity/EDC error.

Note that since the data arrays have a higher probability of incurring an error than the tag arrays, due to the relative storage capacities, most errors will be transparently corrected. Using write-through mode for critical data will ensure that invalidation or refills are able to recover from errors transparently in most cases.

If a cache hit occurs on a way without a tag parity/EDC error, and a parity error is detected on any portion of the accessed double word of data for a load, and the line is dirty or a dirty error occurs, no refill of the cache line will occur, the line will not be invalidated, and a machine check will also occur, even if auto-invalidation is selected. All other clean unlocked lines with uncorrectable tag errors will also have been invalidated during the correction/auto-invalidation cycle if one was initially needed. Tag parity/EDC errors on lines which were not invalidated earlier due to lock or dirty status will be ignored

If a cache hit occurs only on a line(s) with an uncorrectable tag parity/EDC error after a invalidation /correction cycle has been performed, since the line is dirty or has a dirty parity error (it would have been invalidated otherwise), a machine check is generated, and no linefill is performed.

If a cache miss occurs and any line with an uncorrectable tag parity/EDC error is dirty or has a dirty parity error, the line is not invalidated, a machine check is generated, and no linefill is performed. All clean lines with tag errors will have been invalidated/corrected on a cache miss, regardless of locked status.

For all cases of invalidations, if any line which was locked or incurred a lock error was invalidated, a machine check will also occur, even though auto-invalidation is selected. Invalidation on a miss is not blocked for locked lines or lines with lock parity errors unless the access is cache-inhibited or is dirty. The lock bits will remain unmodified by the invalidation operation to allow for potential software recovery.

If a refill of a locked line due to a data parity error encounters an external bus error during the linefill, a machine check will be generated, the line will be invalidated, and the lock bits will remain set.

9.8.1.2.3 Data cache line flush or invalidation due to reservation instructions (l[b,h,w]arx, st[b,h,w]cx.)

Normally, when executing a load and reserve, or a store conditional instruction, a cache line hit results in the line being pushed (if dirty) and marked clean, and the reservation access performed as a single-beat access. Certain parity or EDC errors may cause other actions however.

If a cache hit to a line with no tag parity/EDC error occurs when performing a lookup for a load or store reservation access, the line will be pushed if dirty, or if a dirty parity error occurs, and will be marked as clean. Locked status will not be changed. A push parity error may occur during the push if a data parity error is encountered, and a machine check will be generated. In this case the reservation access will not be performed. Otherwise, a load reservation access is then performed as a single-beat access, ignoring the cache data. A store reservation access is performed as a writethrough single-beat write access on the bus, regardless of whether it is marked as writethrough required. If the write access completes without error and succeeds (no ERROR or XFAIL response from the bus), then the cache is updated with the store data, but the line is left in a clean state. Uncorrectable tag errors on other clean unlocked lines will cause invalidation of those lines without signaling a machine check. Uncorrectable tag errors on other cache lines which are locked or are dirty will be ignored.

Otherwise, if any line has an uncorrectable tag parity/EDC error and is dirty or has a dirty parity error, a machine check is generated, and the line(s) remains unchanged. Clean unlocked lines with tag parity/EDC errors will be invalidated or corrected, but locked lines or lines with a lock error will not be invalidated on a cache miss, since no new cache line will be allocated.

9.8.2 Parity/EDC Error Handling for Cache Control Operations and Instructions

Parity/EDC errors are not signaled when the respective L1CSR0[DCECE] and L1CSR1[ICECE] cache error checking enable bits are cleared. The following sections describe error handling for cache control operations and cache control instructions when set.

9.8.2.1 L1FINV0/L1FINV1 Operations

For invalidation operations via the L1FINV0/L1FINV1 control registers, uncorrectable tag parity or EDC errors result in the specified line being invalidated. No error is reported, regardless of the setting of the DCEA/ICEA bit. Data parity or EDC errors and dirty errors are ignored. Parity or EDC errors on all other ways not specified by the CWAY value for L1FINV0/L1FINV1 are ignored, regardless of the settings of the DCEA/ICEA bit.

For flush and flush with invalidate operations via the L1FINV0 control register, if no uncorrectable tag parity/EDC error occurs on the specified line, it is flushed to memory if dirty or if a dirty parity error occurs and then invalidated for flush with invalidate operations. No machine check is signaled for dirty parity errors. If an uncorrectable tag parity/EDC error occurs on the specified line, and the line is dirty or a dirty error is encountered, no flush or invalidation is performed. The line remains unchanged, and a machine check is generated. For flush operations, an uncorrectable tag parity or EDC error on a clean line is ignored, and no error is reported. For flush with invalidate operations, an uncorrectable tag parity or EDC error on a clean line results in the specified line being invalidated, and no error is reported. Lock status is ignored for these operations.

Data parity errors may result in a push parity error and a machine check being generated. However, the line is still flushed to memory if not prevented due to an uncorrectable tag parity/EDC error. If a push parity error occurs, the line is left unaffected for flush with invalidate operations. Lock status is cleared on an invalidation or flush with invalidation that does not result in a machine check.

9.8.2.2 Cache touch instructions (dcbt, dcbtst, icbt)

Parity errors are not signaled on a lookup for a **dcbt**, **dcbtst**, or **icbt** instruction. For those instructions, an uncorrectable tag parity or EDC error results in a No-op and no error is reported, regardless of error checking being enabled. No invalidations occur.

9.8.2.3 icbi instructions

For **icbi** instructions, on a hit to any locked or unlocked line without an uncorrectable tag parity/EDC error (with or without a lock parity error), or on a hit to an unlocked line with an uncorrectable tag parity/EDC error, the line(s) is invalidated, regardless of the setting of L1CSR1[ICEA]. No machine check is generated. If L1CSR1[ICEA] = 01, if any line has a tag parity/EDC error, a correction/invalidation cycle is inserted to correct tags with single-bit errors and to invalidate unlocked lines with multi-bit errors. Locked lines with uncorrectable tag errors which miss are unaffected. No machine check is generated.

If a hit occurs to a line with a tag parity/EDC error (after a correction for L1CSR1[ICEA] = 01) that is locked or has a lock parity error, the line is left unaffected. No machine check is generated, regardless of the setting of L1CSR1[ICEA].

If a miss occurs, all parity/EDC errors are ignored, the lines are left unaffected. No machine check is generated, regardless of the setting of L1CSR1[ICEA].

All data parity or EDC errors are ignored regardless of L1CSR1[ICEA].

9.8.2.4 dcbi instructions

For **dcbi** instructions, on a hit to a line without a tag parity/EDC error, the line is invalidated, regardless of the setting of L1CSR0[DCEA]. For this case, data, lock, and dirty parity errors are ignored. When

L1CSR0[DCEA] = 00, tag parity/DC errors on other lines are ignored. When L1CSR0[DCEA] = 01, uncorrectable tag parity/EDC errors on other lines also cause clean unlocked lines to be invalidated, regardless of hit or miss. No machine check is generated regardless of the setting of L1CSR0[DCEA].

For **dcbi** instructions that hit to a line with a tag parity/EDC error, the line(s) is invalidated if clean and unlocked and no machine check is generated, regardless of the setting of L1CSR0[DCEA]. Uncorrectable tag parity/EDC errors will cause other clean unlocked lines to be invalidated when L1CSR0[DCEA] = 01, regardless of hit or miss. If a hit occurs to a line with an uncorrectable tag parity/EDC error and the line is dirty, or is locked or has a lock parity error, the line is left unaffected, and no machine check is generated, regardless of the setting of L1CSR0[DCEA].

For **dcbi** instructions that miss in all ways, when L1CSR0[DCEA] = 00, no invalidation is performed regardless of tag parity/EDC errors and no machine check is signaled. Uncorrectable tag parity/EDC errors cause clean unlocked lines to be invalidated when L1CSR0[DCEA] = 01, and no machine check is signaled. All other lines are left unchanged.

9.8.2.5 dcbst instructions

For **dcbst** instructions, on a hit to any line without a tag parity or EDC error, if the line is dirty, or has a dirty bit error, the line is flushed. Lock errors are ignored. When L1CSR0[DCEA] = 00, tag parity/EDC errors on other lines are ignored. When L1CSR0[DCEA] = 01, uncorrectable tag parity/EDC errors on other lines also cause clean unlocked lines to be invalidated, regardless of hit or miss. No machine check is generated regardless of the setting of L1CSR0[DCEA]. For **dcbst**, lock and dirty errors are ignored on a hit. Data parity errors will not prevent the line from being flushed, but will cause a machine check to be generated due to a push parity error.

For cacheable **dcbst** instructions that hit only to a line with a tag parity/EDC error or that miss in all ways, a machine check will be generated if L1CSR0[DCEA] = 00 and any line with a tag parity or EDC error is dirty. Lock errors are ignored. If L1CSR0[DCEA] = 01, clean unlocked lines with an uncorrectable tag parity/EDC error are invalidated, and no errors are signaled unless any line with an uncorrectable tag parity/EDC error is also dirty or has a dirty parity error. If any line with an uncorrectable tag parity/EDC error is dirty or has a dirty parity error, the line is not flushed and a machine check is generated, regardless of the settings of L1CSR0[DCEA].

9.8.2.6 dcbf Instructions

For **dcbf** instructions, on a hit to any line without a tag parity or EDC error, if the line is dirty or has a dirty bit error, the line is flushed and invalidated. Lock errors are ignored. When L1CSR0[DCEA] = 00, tag parity/EDC errors on other lines are ignored. When L1CSR0[DCEA] = 01, uncorrectable tag parity/EDC errors on other lines also cause clean unlocked lines to be invalidated, regardless of hit or miss. No machine check is generated regardless of the setting of L1CSR0[DCEA]. For **dcbf**, data parity errors do not prevent the line from being flushed, but cause a machine check to be generated due to a push parity error.

For cacheable **dcbf** instructions that hit only to a line with a tag parity/EDC error or that miss in all ways, a machine check is generated if L1CSR0[DCEA] = 00 and any line with a tag parity or EDC error is dirty, locked, or has a dirty parity error or a lock parity error. If L1CSR0[DCEA] = 01, clean unlocked lines with an uncorrectable tag parity/EDC error are invalidated, and no errors are signaled unless any line with an uncorrectable tag parity/EDC error is also dirty, locked, or has a dirty parity error or a lock parity error. If any line with an uncorrectable tag parity/EDC error is dirty or has a dirty parity error, the line is not flushed and a machine check is generated. If any line with an uncorrectable tag parity/EDC error is locked or has a lock parity error, the line is not invalidated, and a machine check is generated.

9.8.2.7 dcbz Instructions

For **dcbz** instructions, on a hit to any line without a tag parity/EDC error, the line is zeroed and set to dirty. Data errors, lock errors, and dirty errors are ignored. When L1CSR0[DCEA] = 00, tag parity/EDC errors on other lines are ignored. When L1CSR0[DCEA] = 01, uncorrectable tag parity/EDC errors on other lines also cause clean unlocked lines to be invalidated, regardless of hit or miss. No machine check is generated regardless of the setting of L1CSR0[DCEA]. For **dcbz**, lock errors are ignored on a hit.

For cacheable **dcbz** instructions that hit only to a line with a tag parity/EDC error or that miss in all ways, a machine check is generated if L1CSR0[DCEA] = 00 and any line has a tag parity/EDC or lock error. If L1CSR0[DCEA] = 01, all line(s) with an uncorrectable tag parity/EDC error are invalidated if clean. If a clean line which was locked or had a lock parity error was invalidated, a machine check is generated. If any line with an uncorrectable tag parity/EDC error is dirty or has a dirty parity error, the line is not affected, and a machine check is generated, regardless of the settings of L1CSR0[DCEA]. If a machine check is generated, no **dcbz** operation will be performed.

9.8.2.8 Cache Locking Instructions (dcbtls, dcbtstls, dcblc, icbtls, icblc)

For **dcbtls**, **dcbtstls**, **dcblc**, **icbtls**, and **icblc** instructions, on a hit to any line without a tag parity or EDC error, the lock bits are set or cleared appropriately, and data, lock, and dirty bit parity or EDC errors are ignored. When L1CSR0[DCEA]/L1CSR1[ICEA] = 00, tag parity/EDC or lock errors on other lines are ignored. When L1CSR0[DCEA]/L1CSR1[ICEA] = 01, uncorrectable tag parity/EDC errors on other lines also cause clean unlocked lines to be invalidated, regardless of hit or miss. No machine check is generated regardless of the setting of L1CSR0[DCEA]/L1CSR1[ICEA]$_A$.

For cacheable **dcbtls**, **dcbtstls**, and **icbtls** instructions that hit only to a line with a tag parity/EDC error or which miss in all ways, a machine check is generated if L1CSR0[DCEA]/L1CSR1[ICEA] = 00 and any line has a tag parity/EDC error or a lock error. If L1CSR0[DCEA]/L1CSR1[ICEA] = 01, clean lines with an uncorrectable tag parity/EDC error are invalidated and if a clean line which was locked or had a lock parity error was invalidated, a machine check is generated. If any line with an uncorrectable tag parity/EDC error is dirty or has a dirty parity error, the line is not affected and a machine check is generated, regardless of the settings of L1CSR0[DCEA]/L1CSR1[ICEA].

For cacheable **dcblc** and **icblc** instructions that hit only to a line with a tag parity/EDC error or that miss in all ways, a machine check is generated if L1CSR0[DCEA]/L1CSR1[ICEA] = 00 and any line with a tag parity/EDC error is locked or has a lock parity error. L1CSR0[DCEA]/L1CSR1[ICEA] = 01, lock and dirty parity errors do not cause a machine check on their own, but clean lines with an uncorrectable tag parity/EDC error are invalidated. If a clean line that was locked or had a lock parity error was invalidated, a machine check is generated. If any locked line with an uncorrectable tag parity/EDC error is dirty or has a dirty parity error, the line is not affected and a machine check is generated, regardless of the settings of L1CSR0[DCEA]/L1CSR1[ICEA].

9.8.3 Cache Inhibited Accesses and Parity/EDC Errors

For non-cacheable access misses, no cache parity/EDC exceptions are signaled. When operating with correction/auto-invalidation disabled, tag parity/EDC errors cause misses for cache-inhibited accesses, and no machine check is generated. When correction/auto-invalidation mode is enabled, a correction/auto-invalidation cycle is run to correct/auto-invalidate tag, dirty, and lock errors, but invalidations are only performed for uncorrectable tag errors on clean unlocked lines. If a cache-inhibited load or instruction fetch access hit occurs to a line with no tag parity/EDC error, and the requested double word of data has no parity/EDC error, the access is treated as a cache hit and the CI status is ignored. Otherwise, if the requested double word of data has a parity/EDC error, the access is treated as a cache-inhibited cache miss and the cache data is ignored, even if dirty. No machine check is generated in this case. A cache-inhibited store hit to a line with no tag parity/EDC error causes the data to be written to the cache, as well as to memory if the store is a write-through store, and all data parity errors are ignored. If a cache hit occurs to a line with an uncorrectable tag error, the hit is ignored, the access is performed as a cache-inhibited cache miss, and the cache data is ignored, even if dirty. No machine check is generated in this case.

For cache control instructions such as **dcbf**, **dcbi**, **icbi**, and **dcbst**, which are performed to addresses marked as cache-inhibited, no machine checks are generated. The operations are only performed on/for lines which would not cause exceptions for the non-CI cases.

9.8.4 Snoop Operations and Parity/EDC Errors

For snoop command lookups in which a hit occurs to a cache line with no tag parity/EDC error, tag parity/EDC errors in other lines are ignored, and no error condition is signaled.

Otherwise, for snoop command lookups in which a tag parity/EDC error occurs and no hit occurs to a tag entry without a parity/EDC error, no correction attempt for the tags with errors is made regardless of L1CSR0[DCEA]. The snoop response indicates an error condition. When such a tag parity/EDC error occurs on a snoop invalidate command, the invalidation does not occur, and the error results in a machine check. The snoop queue continues to be serviced, and the machine check will not necessarily be recoverable. A checkstop condition does not occur however. In this respect, it is treated similarly to a non-maskable interrupt, and MSR[RI] should be used accordingly by software.

9.8.5 EDC Checkbit/Syndrome Coding Scheme Generation—Icache

When operating with EDC enabled (L1CSR1[ICEDT] = 01), double bit error detection codes are used to protect the tag and data portions of an instruction cache line. Each tag entry utilizes six check bits to cover the tag + valid bit, and each double word of data in the data arrays utilizes eight check bits. These same bits are used for parity coding when the L1CSR1[ICEDT] control field selects parity mode. The specific coding schemes are shown in Table 9-7 and Table 9-8. The lock bits utilize bit-level redundancy, thus are independently protected.

Table 9-7 shows the checkbit coding for each tag entry. A ‘*’ in the table indicates the bit is XORed to form the final checkbit value.

Table 9-7. Tag Checkbit Generation

Checkbits p_tchk[0:5]	Tag Bit																						
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	V
0	*	*	*	*	*	*			*	*	*		*	*		*							
1		*	*				*	*		*	*	*	*				*	*				*	*
2				*	*			*	*	*					*	*	*	*	*		*		*
3				*		*					*	*	*	*	*	*	*			*	*	*	*
4	*	*		*	*		*	*			*			*					*	*	*	*	*
5	*		*	*		*	*		*		*	*			*			*	*	*			*

Table 9-8 shows the checkbit coding for each double word data entry. A ‘*’ in the table indicates the bit is XORed to form the final checkbit value.

Table 9-8. Data Checkbit Generation

Checkbits p_dchk[0:7]	Data Bit																															
	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
0	*	*	*	*	*	*	*	*			*			*	*			*			*			*	*			*				
1	*	*	*						*	*	*	*	*	*	*	*			*			*	*			*			*			*
2				*	*	*			*	*	*						*	*	*	*	*	*	*	*			*			*	*	
3				*			*	*				*	*	*			*	*	*						*	*	*	*	*	*	*	*
4				*								*			*	*				*	*	*			*	*	*					
5	*			*								*								*			*	*				*	*	*		
6		*			*			*	*			*								*								*			*	*
7			*			*	*			*			*			*	*			*								*				

Checkbit	Data Bit																															
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
0				*								*			*	*				*	*	*			*	*	*					
1	*			*								*								*			*	*				*	*	*		
2		*			*			*	*			*								*								*			*	*
3			*			*	*			*			*			*	*			*								*				
4	*	*	*	*	*	*	*	*			*			*	*			*			*			*	*			*				
5	*	*	*						*	*	*	*	*	*	*	*			*			*	*			*			*			*

Table 9-8. Data Checkbit Generation (continued)

Checkbits p_dchk[0:7]	Data Bit																															
	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
6				*	*	*			*	*	*						*	*	*	*	*	*	*	*			*			*	*	
7				*			*	*				*	*	*			*	*	*						*	*	*	*	*	*	*	*

9.8.6 EDC Checkbit/Syndrome Coding Scheme Generation—Dcache

When operating with EDC enabled (L1CSR0[DCEDT] = 01), double bit error detection codes are used to protect the tag portion of a data cache line. The data array continues to utilize single-bit parity protection. Each data cache tag entry utilizes six check bits to cover the tag + valid bit. Two of these same bits are used for parity coding when the L1CSR0[DCEDT] control field selects parity mode. The specific coding scheme for the tag array is the same as is used for the Icache, and is shown in Table 9-7. The dirty and lock bits utilize bit-level redundancy, thus are independently protected. Three dirty bits are provided to support single-bit and double-bit error detection. Correction is performed by setting the dirty bits to 1 if a dirty parity error occurs and correction/auto-invalidation is enabled. Four lock bits are provided to support single-bit error correction and double-bit error detection.

9.8.7 Cache Error Injection

Cache error injection provides a way to test error recovery by intentionally injecting parity errors into the instruction and/or data cache.

Error injection into the instruction cache operates as follows:

- If L1CSR1[ICEI] is set and the L1CSR1[ICEDT] = 00, any instruction cache line fill to the instruction cache data has all of the associated parity bits inverted in the instruction cache data array for each double word loaded.
- If L1CSR1[ICEI] is set and L1CSR1[ICEDT] = 01, any instruction cache line fill to the instruction cache data has the associated two most significant parity check bits inverted in the instruction cache data array for each double word loaded.

Error injection for the data cache operates as follows:

- If L1CSR0[DCEI] is set, any cache line fill to the data cache data array has all of the associated parity bits inverted in the data array for each double word loaded. Additionally, inverted parity bits are generated for any bytes stored into the data cache data array on a store hit.

Cache parity error injection is not performed for cache debug write accesses, since parity bit values written can be directly controlled (See Section 9.19.3, “Cache Debug Access Control Register (CDACNTL)”).

In order to clear the parity errors, a cache invalidation or an invalidation of the lines which could have had an injected parity error may be performed. Line invalidation may be performed by an **icbi/dcbi** instruction, or an L1FINV[0,1] invalidation operation.

9.8.8 Cache Error Cross-Signaling

Cache error cross-signaling provides a way to support multiple cores running in lock-step when one of the CPUs encounters a parity/EDC error in the instruction and/or data cache. Refer to Section 11.2.13, "Cache Error Cross-signaling Signals," and Section 11.3.4, "Cache Error Cross-signaling Operation," for more details of operation.

9.9 Push and Store Buffers

The push buffer reduces latency for requested new data on a data cache miss by temporarily holding displaced dirty data while the new data is fetched from memory. The push buffer contains 32 bytes of storage (one displaced cache line).

If a data cache miss displaces a dirty line, the linefill request is forwarded to the external bus. While waiting for the response, the current contents of the dirty cache line are placed into the push buffer. Once the linefill transaction (burst read) completes, the cache controller can generate the appropriate burst write bus transaction to write the contents of the push buffer into memory.

The store buffer contains a FIFO that can defer pending write misses or writes marked as write-through in order to maximize performance. The store buffer can buffer as many as eight words (32 bytes) for this purpose. The store buffer may be disabled for debug purposes. Operation of the store buffer is independent of L1CSR0[DCE]. When the store buffer is enabled, non-allocating store operations which miss the cache or which are marked as writethrough are placed in the store buffer, and the CPU access is terminated. Each store buffer entry contains 32-bits of physical address, 32-bits of data, size information, and 3 bits of access attribute information (W, G, and S/U) in order to properly drive the **attribute** output signals on a buffered store access. Cache-inhibited guarded stores are not buffered however, and are delayed from being performed until the push and store buffers have been emptied.

Once the push or store buffer has valid data, the internal bus controller uses the next available external bus cycle to generate the appropriate write cycles. In the event that another data cache fill is required (e.g., cache load or store w/allocate miss to process) during the continued instruction execution by the processor pipeline, an alias check is performed between the linefill address and all valid entries in the store and push buffer using the index portion of the access address. If no match is found, the linefill may bypass pending stores in the store or push buffer. Otherwise, if an alias exists (index matches any valid store buffer entry), the data cache pipeline will stall until the aliased entries have been flushed from the store and push buffer before generating the required external bus transaction for the linefill.

Single-beat read transactions will not bypass pending stores in the push or store buffer.

The push buffer is always emptied prior to queued store buffer entries to avoid memory consistency issues. Once the push buffer has been loaded with dirty data to be written back to memory, a subsequent store may be buffered, but will not be written to memory until the push has completed.

For cache-inhibited load accesses or cache-inhibited guarded store accesses, the processor termination is withheld until the store buffer has been flushed of all entries, the push buffer has been emptied, and the access has completed to memory.

A write to the L1CSR0 register may be used to force the push and store buffers to empty before proceeding with the actual L1CSR0 update. Additionally, the **msync** and **mbar** instructions also cause these buffers to be emptied prior to completion.

If an external transfer ERROR response occurs while emptying the store buffer, a machine check exception is signaled to the CPU, and a store for the next entry to be written (if any) is initiated. If a transfer error occurs for a push buffer transaction, the push of the remaining portion of the cache line is aborted, and a machine check exception is signaled to the CPU. This is also the case for a cache control operation that causes a line to be pushed. Following the transfer error, the line is marked invalid. If it is possible for a transfer error to be returned by the system on a push or a buffered store, and this could cause a problem, the address must be marked guarded and cache inhibited.

External termination errors that occur on any push of a dirty cache line results in a machine check condition.

9.10 Cache Management Instructions

This section describes the implementation of cache management instructions in the e200z7.

9.10.1 Instruction Cache Block Invalidate (icbi) Instruction

If the cache line containing the byte addressed by the EA associated with this instruction is present in the instruction cache, it is invalidated, regardless of lock status. If an instruction cache linefill is in progress and the linefill data corresponds to the EA associated with a **icbi**, the instruction cache is not updated with linefill data.

See the *EREF* for the full description of **icbi**.

9.10.2 Instruction Cache Block Touch (icbt) Instruction

If HID0[NOPTI] is set, this instruction is treated as a no-op. See the *EREF* for the full description of **icbt**.

9.10.3 Data Cache Block Allocate (dcba) Instruction

This instruction is treated as a no-op. See the *EREF* for the full description of **dcba**.

9.10.4 Data Cache Block Flush (dcbf) Instruction

If the cache line containing the byte addressed by the EA associated with this instruction is present in the data cache, it is copied back to memory if dirty. The line is subsequently invalidated regardless of whether it was copied back or locked. If a data cache linefill is in progress and the linefill data corresponds to the EA associated with a **dcbf**, the data cache is not updated with linefill data.

This instruction is treated in the following way:

- As a load for the purposes of access protection
- As a no-op if the data cache is disabled

See the *EREF* for the full description of **dcbf**.

9.10.5 Data Cache Block Invalidate (dcbi) Instruction

If the cache line containing the byte addressed by the EA associated with this instruction is present in the data cache, it is invalidated, regardless of lock status. No copyback occurs if the line is present in the data cache and dirty. If a data cache linefill is in progress and the linefill data corresponds to the EA associated with a **dcbi**, the data cache is not updated with linefill data.

This instruction is privileged. It is treated in the following way:

- As a store for the purposes of access protection
- As a no-op in supervisor mode if the data cache is disabled

See the *EREF* for the full description of **dcbi**.

9.10.6 Data Cache Block Store (dcbst) Instruction

If the cache line containing the byte addressed by the EA associated with this instruction is present in the data cache, it is copied back to memory if dirty. The line is subsequently marked clean, and the lock status is unchanged. The following conditions apply:

- This instruction is treated as a load for the purpose of access protection.
- If the data cache is disabled, this instruction is treated as a no-op.

See the *EREF* for the full description of **dcbst**.

9.10.7 Data Cache Block Touch (dcbt) Instruction

If HID0[NOPTI] is set, this instruction is treated as a no-op. See the *EREF* for the full description of **dcbt**.

9.10.8 Data Cache Block Touch for Store (dcbtst) Instruction

If HID0[NOPTI] is set, this instruction is treated as a no-op. See the *EREF* for the full description of **dcbtst**.

9.10.9 Data Cache Block set to Zero (dcbz) Instruction

If the cache line containing the byte addressed by the EA associated with this instruction is present in the data cache, all bytes in the line are zeroed, the line is marked as modified and remains valid. Lock status remains unchanged. If the cache line is not present and the address is cacheable, it is established in the data cache (without fetching from memory), all bytes in the line are zeroed, and the line is marked as modified and valid.

This instruction is treated as a store for the purposes of access protection.

dcbz causes an alignment exception if the EA is marked by the MMU as cache-inhibited and a data cache miss occurs, if the EA is marked by the MMU as write through required, if the data cache is disabled or is operating in writethrough mode, or if an overlocking condition prevents the allocation of a line into the data cache.

See the *EREF* for the full description of **dcbz**.

9.11 Touch Instructions

Due to the limitations of using the **icbt**, **dcbt**, and **dcbtst** instructions, a program that uses these instructions improperly may actually see a degradation in performance from their use. To avoid this, the e200z7 provides the HID0[NOPTI] control bit to cause these instructions to be treated as no-ops.

9.12 Cache Line Locking/Unlocking Unit

This section has the following structure:

- Section 9.12.1, "Overview," provides an overview of the Freescale EIS cache line locking/unlocking unit.
- Section 9.12.2, "Instruction Details," describes the instructions shown in Table 9-9.

Table 9-9. Cache Line Locking/Unlocking Unit Instructions

Acronym	Definition	Cross-Reference
dcbtls	Data cache block touch and lock set	9-35
dcbtstls	Data cache block touch for store and lock set	9-37
dcblc	Data cache block lock clear	9-38
icbtls	Instruction cache block touch and lock set	9-39
icblc	Instruction cache block lock clear	9-41

- Section 9.12.3, "Effects of Other Cache Instructions on Locked Lines" identifies which instructions have no effect on the state of the lock bit and which instructions flush/invalidate and unlock a cache line.
- Section 9.12.4, "Flash Clearing of Lock Bits" explains how the e200z7 supports flash clearing of lock bits.

9.12.1 Overview

The e200z7 supports the Freescale EIS cache line locking unit which defines user-mode instructions to perform cache locking/unlocking. Three of the instructions are for data cache locking control (**dcblc**, **dcbtls**, **dcbtstls**) and two instructions are for instruction cache locking control (**icblc**, **icbtls**).

The **dcbtls**, **dcbtstls**, and **dcblc** lock instructions are treated as reads for checking access permissions when translated by the TLB, and exceptions are taken for data TLB errors or data storage interrupts. The **icbtls** and **icblc** instructions require either execute (X) or read (R) permission when translated by the TLB. Exceptions are taken using data TLB errors (DTLB) or data storage interrupts (DSI), not ITLB or ISI.

The user-mode cache lock enable MSR[UCLE] may be used to restrict user-mode cache line locking. If MSR[UCLE] is clear, any cache lock instruction executed in user-mode takes a cache-locking DSI exception (unless no-oped) and set either ESR[DLK] or ESR[ILK]. If MSR[UCLE] is set, cache-locking instructions can be executed in user-mode and they will not take a DSI for cache-locking. However, they may still cause a DSI for access violations or cause machine checks for external termination errors.

The following list identifies cases where attempting to set a lock fail, even when no DSI or DTLB exceptions occur.

- The target address is marked cache-inhibited and a cache miss occurs.
- The cache is disabled or all ways of the cache are disabled for replacement.
- The cache target indicated by the CT field (bits 6–10) of the instruction is not 0.

In these cases, the lock set instruction is treated as a no-op, and the cache unable to lock L1CSR{0,1}[CUL] bit is set.

Assuming no exception conditions occur (DSI or DTLB error), for **dcbtls**, **dcbtstls**, and **icbtls** an attempt is made to lock the corresponding cache line. If a miss occurs, and all of the available ways (ways enabled for a particular access type) are already locked in a given cache set, an attempt to lock another line in the same set will result in an overlocking situation. In this case, the cache overlock bit L1CSR{0,1}[CLO] is set to indicate that an overlocking situation occurred. This does not cause an exception condition. The new line is conditionally placed in the cache, displacing a previously locked line depending on the setting of the appropriate L1CSR0,1[CLOA].

The CUL conditions have priority over the CLO condition.

If multiple no-op or exception conditions arise on a cache lock instruction, the results are determined by the order of precedence described in Table 9-11.

It is possible to lock all ways of a given cache set. If an attempt is made to perform a non-locking line fill for a new address in the same cache set, the new line is not put into the cache. It is satisfied on the bus using a single beat transfer instead of normal burst transfers. If a **dcbz** instruction is executed, and all ways available for allocation have been locked, an alignment exception is generated and no line is put into the cache.

Cache line locking interacts with the ability to control replacement of lines in certain cache ways via the L1CSR0 WID and WDD control bits. If any cache line locking instruction (**icbtls**, **dcbtls**, **dcbtstls**) is allowed to execute and finds a matching line already present in the cache, the line's lock bit is set regardless of the settings of the WID and WDD fields. In this case, no replacement has been made. However, for cache misses that occur while executing a cache line lock set instruction, the only candidate lines available for locking are those that correspond to ways of the cache that have not been disabled for the particular type of line locking instruction (controlled by WDD for **dcbtls** and **dcbtstls**, controlled by WID for **icbtls**). Thus, an overlocking condition may result even though fewer than four lines with the same index are locked.

The cache-locking DSI handler must decide whether or not to lock a given cache line based upon available cache resources. If the locking instruction is a set lock instruction, and if the handler decides to lock the line, it should do the following:

- Add the line address to its list of locked lines.
- Execute the appropriate set lock instruction to lock the cache line.

- Modify save/restore register 0 to point to the instruction immediately after the locking instruction that caused the DSI.
- Execute an **rfi**.

If the locking instruction is a clear lock instruction, and if the handler decides to unlock the line, it should do the following:

- Remove the line address from its list of locked lines.
- Execute the appropriate clear lock instruction to unlock the cache line.
- Modify save/restore register 0 to point to the instruction immediately after the locking instruction that caused the DSI.
- Execute an **rfi**.

9.12.2 Instruction Details

This section provides details for the instructions shown in Table 9-10:

Table 9-10. Cache Line Locking/Unlocking Unit Instructions

Acronym	Definition	Cross-Reference
dcbtls	Data cache block touch and lock set	9-35
dcbtstls	Data cache block touch for store and lock set	9-37
dcblc	Data cache block lock clear	9-38
icbtls	Instruction cache block touch and lock set	9-39
icblc	Instruction cache block lock clear	9-41

dcbtls dcbtls

Data Cache Block Touch and Lock Set

dcbtls CT, RA, RB (E=0) Form X

31	CT	RA	RB	0 0 1 0 1 0 0 1 1 0	0
0 5	6 10	11 15	16 20	21 30	31

Description:

```
if RA=0 then a $\leftarrow {}^{64}0$ else a $\leftarrow$ GPR(RA)
EA <- $^{32}0$ || (a + GPR(RB))$_{32:63}$
PrefetchDataCacheBlockLockSet(CT, EA)
```

If CT = 0, the cache line corresponding to EA is loaded and locked into the level 1 data cache.

If CT = 0 and the line already exists in the data cache, **dcbtls** locks the line without refetching it from external memory.

Exceptions:

If the MSR[UCLE] (user-mode cache lock enable) bit is set, **dcbtls** may be performed while in user mode (MSR[PR] = 1). If MSR[UCLE] is clear, an attempt to perform these instructions in user mode causes a data cache locking error DSI unless the CT field or other conditions otherwise no-op the instruction.

The e200z7 only supports CT = 0. If CT is some value other than 0, the **dcbtls** is no-oped and the L1CSR0[DCUL] bit is set indicating an unable-to-lock condition occurred. No other exceptions are reported. If the data cache is disabled, the **dcbtls** is no-oped and L1CSR0[DCUL] is set indicating an unable-to-lock condition occurred. No other exceptions are reported.

The **dcbtls** instruction is treated as a load with respect to translation and causes a DSI interrupt for access violations, as well as causing a Data TLB error interrupt if the target address cannot be translated.

If the block corresponding to EA is cache-inhibited and a data cache miss occurs, the instruction is no-oped, (no DSI is taken due to the cache-inhibited status), and L1CSR0[DCUL] is set, indicating an unable-to-lock condition occurred.

Other registers altered:

- L1CSR0 (see below)

When a **dcbtls** is performed to an index, and a way can not be locked, L1CSR0[DCUL] is set, indicating an unable-to-lock condition occurred. This also occurs whenever the **dcbtls** must be no-oped.

When a **dcbtls** is performed to an index in the data cache that already has all the ways locked, this is referred to as an over-locking situation. There is no exception generated by an over-locking situation. Instead, L1CSR0[DCLO] is set, indicating an over-lock condition occurred. A line is allocated and locked in the cache depending on the setting of the L1CSR0[DCLOA] control bit. If system software wants to precisely determine if an overlock condition has happened, it must perform the following code sequence:

```
dcbtls
msync
```

```
mfspr (L1CSR0)
       (check L1CSR0[DCUL] bit for cache index unable-to-lock condition)
       (check L1CSR0[DCLO] bit for cache index over-lock condition)
```

dcbtstls dcbtstls

Data Cache Block Touch for Store and Lock Set

dcbtstls CT, RA, RB (E=0) Form X

31	CT	RA	RB	0 0 1 0 0 0 0 1 1 0	0
0 5	6 10	11 15	16 20	21 30	31

Description:

```
if RA=0 then a ← $^{64}0$ else a ← GPR(RA)
EA <- $^{32}0$ || (a + GPR(RB))$_{32:63}$
PrefetchDataCacheBlockLockSet(CT, EA)
```

The e200z7 treats the **dcbtstls** instruction identically to the **dcbtls** instruction because no hardware coherency mechanisms are implemented for the cache.

dcblc dcblc

Data Cache Block Lock Clear

dcblc CT, RA, RB (E=0) Form X

31	CT	RA	RB	0 1 1 0 0 0 0 1 1 0	0
0 5	6 10	11 15	16 20	21 30	31

Description:

```
if RA=0 then a ← $^{64}0$ else a ← GPR(RA)
EA <- $^{32}0$ || (a + GPR(RB))$_{32:63}$
     DataCacheClearLockBit(CT, EA)
```

If CT = 0, and the line is present in the L1 data cache, the lock bit for that line is cleared, making that line eligible for replacement.

Exceptions:

If the MSR[UCLE] (user-mode cache lock enable) bit is set, **dcblc** may be performed while in user mode (MSR[PR] = 1). If MSR[UCLE] is clear, an attempt to perform this instructions in user mode causes a DSI, unless the CT field or other conditions otherwise no-op the instruction.

The e200z7only supports CT = 0. If CT is some value other than 0, the **dcblc** is no-op’ed. No other exceptions are reported. If the data cache is disabled, the **dcblc** is no-op’ed. No other exceptions are reported.

The **dcblc** instruction is treated as a load with respect to translation and causes a DSI interrupt for access violations, as well as a Data TLB error interrupt if the target address cannot be translated.

icbtls icbtls

Instruction Cache Block Touch and Lock Set

icbtls CT, RA, RB (E=0) Form X

31	CT	RA	RB	0 1 1 1 1 0 0 1 1 0	0
0 5	6 10	11 15	16 20	21 30	31

Description:

```
if RA=0 then a ← $^{64}0$ else a ← GPR(RA)
EA <- $^{32}0$ || (a + GPR(RB))$_{32:63}$
    PrefetchInstructionCacheBlockLockSet(CT, EA)
```

If CT = 0, the cache line corresponding to EA is loaded and locked into the level 1 instruction cache.

If CT = 0 and the line already exists in the instruction cache, **icbtls** locks the line without refetching it from external memory.

Exceptions:

If MSR[UCLE] (user-mode cache lock enable) is set, **icbtls** may be performed while in user mode (MSR[PR] = 1). If MSR[UCLE] is clear, an attempt to perform these instructions in user mode causes an Instruction cache locking error DSI unless the CT field or other conditions otherwise no-op the instruction.

The e200z7 only supports CT = 0. If CT is some value other than 0, the **icbtls** is no-op'ed and L1CSR1[ICUL] is set, indicating an unable-to-lock condition occurred. No other exceptions are reported. If the instruction cache is disabled, the **icbtls** is no-op'ed and L1CSR1[ICUL] is set, indicating an unable-to-lock condition occurred. No other exceptions are reported.

The **icbtls** instruction requires either execute or read (X or R) permissions with respect to translation and cause a DSI interrupt for access violations as well as a Data TLB error interrupt if the target address cannot be translated.

If the block corresponding to EA is cache-inhibited and an instruction cache miss occurs, the instruction is no-op'ed, (no DSI is taken due to the cache-inhibited status), and the L1CSR1[ICUL] bit is set indicating an unable-to-lock condition occurred.

Other registers altered:

- L1CSR1 (see below)

When **icbtls** is performed to an index and a way can not be locked, the L1CSR1[ICUL] bit is set indicating an unable-to-lock condition occurred. This also occurs whenever **icbtls** must be no-op'ed.

When **icbtls** is performed to an index in the instruction cache that already has all the ways locked, this is referred to as an overlocking situation. There is no exception generated by an overlocking situation. Instead L1CSR1[ICLO] is set, indicating an overlock condition occurred. A line is allocated and locked in the cache depending on the setting of the L1CSR1[ICLOA] control bit. If system software wants to precisely determine whether an overlock condition has happened, it must perform the following code sequence:

```
icbtls
msync
mfspr (L1CSR1)
       (check L1CSR1[ICUL] bit for cache index unable-to-lock condition)
       (check L1CSR1[ICLO] bit for cache index over-lock condition)
```

icblc icblc

Instruction Cache Block Lock Clear

icblc CT, RA, RB (E=0) Form X

31	CT	RA	RB	0 0 1 1 1 0 0 1 1 0	0
0 5	6 10	11 15	16 20	21 30	31

Description:

```
if RA=0 then a ← $^{64}0$ else a ← GPR(RA)
EA <- $^{32}0$ || (a + GPR(RB))$_{32:63}$
    InstCacheClearLockBit(CT, EA)
```

If CT = 0, and the line is present in the instruction cache, the lock bit for that line is cleared, making that line eligible for replacement.

Exceptions:

If the MSR[UCLE] (user-mode cache lock enable) bit is set, **icblc** may be performed while in user mode (MSR[PR]=1). If the MSR[UCLE] bit is clear, an attempt to perform these instructions in user mode causes an instruction cache locking error DSI unless the CT field or other conditions otherwise no-op the instruction.

The e200z7 only supports CT = 0. If CT is some value other than 0, the **icblc** is no-op'ed. No other exceptions are reported. If the instruction cache is disabled, the **icblc** is no-op'ed. No other exceptions are reported.

The **icblc** instruction requires either execute or read (X or R) permissions with respect to translation and cause a DSI interrupt for access violations as well as a Data TLB error interrupt if the target address cannot be translated.

9.12.3 Effects of Other Cache Instructions on Locked Lines

The following cache instructions have no effect on the state of a cache line's lock bit: **icbt**, **dcba**, **dcbz**, **dcbst**, **dcbt,** and **dcbtst**.

The following cache instructions flush/invalidate and unlock a cache line in the respective L1 caches: **dcbf, dcbi,** and **icbi**.

9.12.4 Flash Clearing of Lock Bits

The e200z7 supports flash clearing of cache lock bits under software control by using the CFCL (cache flash clear locks) control bit in the L1CSR{0,1} register.

Lock bits are not cleared automatically upon power-up (**m_por**) or normal reset (**p_reset_b**). Software must use the CLFC control bit to clear the lock bits after a reset. Proper use of this bit is to determine that it is clear and then set it with a pair of **mfspr mtspr** operations. A 0-to-1 transition on CLFC causes a flash clearing of the lock bits to be initiated which lasts for multiple (approx. 134) CPU cycles. Once set, the

CLFC bit will be cleared by hardware after the operation is complete. It remains set during the clearing interval and may be tested by software to determine when the operation has completed. An **mtspr** operation to L1CSR{0,1}, which attempts to change the state of L1CSR{0,1}[CLFC] during invalidation, does not affect the state of that bit.

During the process of performing the flash clearing, the cache does not respond to accesses and remains busy. Interrupts may still be recognized and processed, potentially aborting the flash clearing operation. When this occurs, L1CSR{0,1}[ABT] is set to indicate unsuccessful completion of the operation. Software should read the L1CSR{0,1} register to determine that the operation has completed (L1CSR{0,1}[CLFC] cleared) and then check the status of L1CSR{0,1}[ABT] to determine completion status.

NOTE

Note that while most implementations of the e200z7 stall further instruction execution during this flash clearing interval, this is not guaranteed across all implementations. Thus, software should be written using these guidelines.

9.13 Cache Instructions and Exceptions

All cache management instructions (except **icbt**, **dcba**, **dcbt**, and**dcbtst**) can generate TLB miss exceptions if the effective address cannot be translated, or may generate DSI exceptions due to permission violations. In addition, **dcbz** may generate an alignment interrupt as described in Section 9.10.9, “Data Cache Block set to Zero (dcbz) Instruction.”

The cache locking instructions **dcblc, dcbtls**, **dcbtstls**, **icblc** and **icbtls** generate DSI exceptions if MSR[UCLE] is clear and the locking instruction is executed in user mode (MSR[PR] = 1). Data cache locking instructions that result in a DSI exception for this reason set ESR[DLK] (documented as DLK0 in the Power ISA embedded category), and instruction cache locking instructions that result in a DSI exception for this reason set ESR[ILK] (documented as DLK1 in the Power ISA embedded category).

9.13.1 Exception Conditions for Cache Instructions

If multiple no-op or exception conditions arise on a cache instruction, the results are determined by the order of precedence described in Table 9-11.

Table 9-11. Special Case Handling

Operation	CT ≠ 0	Cache Disabled	TLB Miss	User & UCLE= 0	Protection Violation	WT or Cache in Write-through mode	Cache Parity Error	CI and miss in cache	All Available ways locked	External Termination Error
icbt, dcbt, dcbtst	No-op	No-op	No-op	—	No-op	—	No-op	No-op	No-op	No-op
dcbtls **dcbtstls** **dcblc**	DCUL DCUL No-op	DCUL DCUL No-op	DTLB DTLB DTLB	DLK DLK DLK	DSI DSI DSI	— — —	MC MC MC	DCUL DCUL —	DCLO DCLO —	MC MC —

Table 9-11. Special Case Handling (continued)

Operation	CT ≠ 0	Cache Disabled	TLB Miss	User & UCLE= 0	Protection Violation	WT or Cache in Write-through mode	Cache Parity Error	CI and miss in cache	All Available ways locked	External Termination Error
icbtls **icblc**	ICUL No-op	ICUL No-op	DTLB DTLB	ILK ILK	DSI DSI	— —	MC MC	ICUL —	ICLO —	MC —
dcbz	—	ALI	DTLB	—	DSI	ALI	MC	ALI	ALI	—
dcbf, dcbst	—	No-op	DTLB	—	DSI	—	MC	—	—	MC
icbi, dcbi	—	No-op	DTLB	—	DSI	—	—	—	—	—
Atomic load or store.	— —	— —	DTLB DTLB	— —	DSI DSI	— —	MC MC	— —	— —	MC MC
load store	— —	— —	DTLB DTLB	— —	DSI DSI	— —	MC MC	— —	— —	MC MC

Notes:
- Priority decreases from left to right
- Cache operations that do not set or clear locks ignore the value of the CT field
- “dash” indicates executes normally
- “NOP” indicates treated as a no-op
- DSI = data storage interrupt; ALI = alignment interrupt; DTLB = data TLB interrupt
- DCUL, ICUL = no-op, and set L1CSR0[CUL]
- DCLO, ICLO = no-op, and set L1CSR0[CLO]
- DLK, ILK = data storage interrupt (DSI) and set ESR[DLK] or ESR[ILK]
- MC = Machine Check and update MCAR

9.13.2 Transfer Type Encodings for Cache Management Instructions

Transfer type encodings are used to indicate to the cache whether a normal access, atomic access, cache management control access, or MMU management control access is being requested. These attribute signals are driven with addresses when an access is requested. Table 9-12 shows the definitions of the **p_d_ttype[0:5]** encodings.

Table 9-12. Transfer Type Encoding

p_d_ttype[0:5][1]	Transfer Type	Instruction
00000e	Normal	Normal loads/stores
000010	Atomic	**lwarx**, **stwcx., lharx, sthcx., lbarx, stbcx.**
00010e	Flush Data Block	**dcbst**
00011e	Flush and Invalidate Data Block	**dcbf**
00100e	Allocate and Zero Data Block	**dcbz**
001010	Invalidate Data Block	**dcbi**
00110e	Invalidate Instruction Block	**icbi**

Table 9-12. Transfer Type Encoding (continued)

p_d_ttype[0:5][1]	Transfer Type	Instruction
001110	multiple word load/store	**lmw, stmw**
010000	TLB Invalidate	**tlbivax**
010010	TLB Search	**tlbsx**
010100	TLB Read entry	**tlbre**
010110	TLB Write entry	**tlbwe**
011000	Touch for Instruction	**icbt**
011010	Lock Clear for Instruction	**icblc**
011100	Touch for Instruction and Lock Set	**icbtls**
011110	Lock Clear for Data	**dcblc**
10000e	Touch for Data	**dcbt**
10001e	Touch for Data Store	**dcbtst**
100100	Touch for Data and Lock Set	**dcbtls**
100110	Touch for Data Store and Lock Set	**dcbtstls**

[1] p_ttype[5] ’e’ is set to set to 0.

9.14 Sequential Consistency

The Power ISA embedded category architecture requires that all memory operations executed by a single processor be sequentially self-consistent. This means that all memory accesses appear to be executed in the order that is specified by the program with respect to exceptions and data dependencies. The e200 CPU achieves this effect by operating a single pipeline to the cache/MMU. All memory accesses are presented to the MMU in the exact order that they appear in the program, and therefore exceptions are determined in order.

9.15 Self-Modifying Code Requirements

The following sequence of instructions synchronizes the instruction stream.

1. dcbf
2. icbi
3. msync
4. isync

This sequence ensures that the operation is correct for Power ISA embedded category processors that implement separate instruction and data caches, as well as for multi-processor cache-coherent systems.

9.16 Page Table Control Bits

The Power ISA embedded category architecture allows certain memory characteristics to be set on a page and on a block basis. These characteristics include write through (using the W-bit), cacheability (using the I-bit), coherency (using the M-bit), guarded memory (using the G-bit), and endianness (using the E-bit). Incorrect use of these bits may create situations where coherency paradoxes are observed by the processor. In particular, this can happen when the state of these bits are changed without appropriate precautions being taken (that is, flushing the pages that correspond to the changed bits from the cache) or when the address translations of aliased real addresses specify different values for any of the WIMGE bits.

Certain mixing of WIMG settings are allowed by the Power ISA embedded category architecture. However, others may present cache coherence paradoxes and are considered programming errors.

9.16.1 Write-through Stores

A write-through store (WIMGE = b’1xxxx’) may normally hit to a valid cache line. In this case, the cache line remains in its current state, the store data is written into the cache, and the store goes out on the bus as a single beat write.

9.16.2 Cache-Inhibited Accesses

When the cache-inhibited attribute is indicated by translation (WIMGE = b’x1xxx’) and a cache miss occurs, all accesses are performed as single beat transactions on the system bus with a size indicator corresponding to the size of the load, store, or prefetch operation.

Note that cache inhibited status is ignored on all cache hits.

9.16.3 Memory Coherence Required

For the e200z7, the “memory coherence required” storage attribute (WIMGE = b’xx1xx’) is reflected on the **p_d_gbl** output during each external data access, to indicate to external coherency logic that memory coherence is required. This bit is ignored for instruction accesses.

9.16.4 Guarded Storage

For the e200z7, the guarded storage attribute (WIMGE = b’xxx1x’) is used to determine if a second outstanding data cache miss may proceed to the system interface prior to the termination of the first outstanding miss. If the second address is marked as guarded, it is not presented to the external interface until the previous miss has been completed without error.

9.16.5 Misaligned Accesses and the Endian (E) Bit

Misaligned load or store accesses that cross page boundaries can cause data corruption if the two pages do not have the same endianness (that is, one page is big endian while the other page is little endian). If this occurs, the processor would not get all the bytes, or would get some of them out of order, resulting in garbled data. To protect against data corruption, the e200 core takes a DSI exception and sets the BO (byte ordering) bit in the exception syndrome register whenever this situation occurs.

9.17 Reservation Instructions and Cache Interactions

The e200 core treats reservation instruction (**lbarx**, **lharx**, **lwarx**, **stbcx.**, **sthcx.**, and **stwcx.**) accesses as though they were cache inhibited, regardless of page attributes. Additionally, a cache line corresponding to the address of a reservation instruction access is flushed to memory if dirty, and then invalidated (even if marked as locked) prior to the reservation access being issued to the bus. This allows external reservation logic to be built which properly signals a reservation failure. The bus access is treated as a single-beat transfer.

9.18 Effect of Hardware Debug on Cache Operation

Hardware debug facilities utilize normal CPU instructions to access register and memory contents during a *debug session*. This may have the unavoidable side-effect of causing the store and push buffers to be flushed. During hardware debug, the MMU page attributes are controllable by the debug firmware via settings of the OnCE control register (OCR).

Cache snoop operations continue to be serviced during debug sessions.

Refer to Section 13.4.6.3, “e200 OnCE Control Register (OCR).”

9.19 Cache Memory Access For Debug/Error Handling

The cache memory provides resources needed to do foreground accesses via **mtdcr** instructions executed by the processor, or background accesses through the JTAG/OnCE port to read and write the cache SRAM arrays. Accesses are supported via a pair of device control registers (DCRs) which are also mapped into OnCE-accessible registers. These resources are intended for use by special debug tools and by debug or specialized error recovery exception software, not by general application code.

Access to the cache memory SRAM arrays using **mtdcr** instructions may be performed by supervisor-level software after appropriate synchronization has been performed with **msync**, **isync** instruction pairs. Access to the cache memory SRAM arrays using the JTAG port is conditional on the CPU being in debug mode. The CPU must be placed in debug state prior to initiation of a read or write access via OnCE.

This facility allows access only to the SRAM arrays used for cache tag and data storage. This function is available even when the cache is disabled. The cache linefill buffer, push buffer, store buffer, and late write buffer are all outside of the SRAM arrays and are not accessible. However, before a debug memory access request is serviced, the push and store buffers will be written to external memory, and the late write and linefill buffers will be written to the cache arrays.

9.19.1 Cache Memory Access via Software

Cache debug access control and data information are accessed by executing **mfdcr** and **mtdcr** instructions to the Cache Debug Access control and data registers CDACNTL and CDADATA (see Table 9-13 and Table 9-14). Accesses are performed one word (32 bits) at a time.

For a Cache write access, software must first write the CDADATA register with the desired tag and status flags, or data values. The second step is to write the CDACNTL register with desired tag or data location and parity values, and assert the R/W and GO bits in CDACNTL.

Note that writing a 64-bit value for data requires two passes, one for the even word (A29 = 0) and one for the odd word (A29 = 1). Each 32-bit write will update all of the parity/check bits, so in general, if only a single 32-bit write is performed, it should be preceded by a read of the data which is not being modified in order to properly compute or store all 8 parity/check bits when the modified 32-bit data is written. Tag writes are accomplished in a single pass.

For a Cache read access, software must first access and write the CDACNTL register with desired tag or data location, and assert the R/W and GO bits in CDACNTL. The second step is to read the CDADATA register for the tag or data and read the CDACNTL register for parity information.

Completion of any operation can be determined by reading the CDACNTL register. Operations are indicated as complete when CDACNTL[30:31] = 00. Software should poll the CDACNTL register to determine when an access has been completed prior to assuming validity of any other information in the CDACNTL or CDADATA registers.

Note that no parity errors are generated as a result of **mtdcr**/**mfdcr** instructions involving the CDACNTL or CDADATA registers.

To ensure proper cache write operation, the following program sequence is recommended:

```
        msync
        isync
        mtdcr cdadata, rS1 // set up write data
        mtdcr cdacntl, rS2 // write control to initiate write
        msync
        isync
loop:   mfdcr rN, cdacntl // check for done
        andi. rT, rN, #3
        bne loop
        .
        .
```

To ensure proper cache read operation, the following program sequence is recommended:

```
        msync
        isync
        mtdcr cdacntl, rS2 // write control to initiate read
        msync
        isync
loop:   mfdcr rN, cdacntl // check for done
        andi. rT, rN, #3
        bne loop
        mfdcr rT, cdadata // return data
        .
        .
```

Conflict conditions with snoop accesses to the same cache line cannot be resolved in a manner that guarantees a value read will not change state before a subsequent value written. No interlocking is performed, so a cache entry read as being valid or written to a valid state may become invalid at any time.

9.19.2 Cache Memory Access Through JTAG/OnCE Port

Cache debug access control and data information are serially accessed through the OnCE controller and access the Cache Debug Access control and data registers CDACNTL and CDADATA (see Table 9-13 and Table 9-14). Accesses are performed one word (32 bits) at a time.

For a Cache write access, the user must first write the CDADATA register with the desired tag or data values. The second step is to write the CDACNTL register with desired tag or data location, parity and dirty information (for data writes only), and assert the R/W and GO bits in CDACNTL.

For a Cache read access, the user must first access and write the CDACNTL register with desired tag or data location, and assert the R/W and GO bits in CDACNTL. The second step is to access and read the CDADATA register for the tag or data and read the CDACNTL register for parity.

Completion of any operation can be determined by reading the CDACNTL register. Operations are indicated as complete when CDACNTL[30:31] = 00. Debug firmware should poll the CDACNTL register to determine when an access has been completed prior to assuming validity of any other information in the CDACNTL or CDADATA registers.

Conflict conditions with snoop accesses to the same cache line cannot be resolved in a manner that guarantees a value read will not change state before a subsequent value written. No interlocking is performed, so a cache entry read as being valid or written to a valid state may become invalid at any time.

9.19.3 Cache Debug Access Control Register (CDACNTL)

The Cache Debug Access Control Register (CDACNTL) contains location information (T/D, CWAY, CSET, and WORD), and control (R/W and GO) needed to access the Cache Tag or Data SRAM arrays. Also included here are the SRAM parity bit values which must be supplied by the user for write accesses, and which will be supplied by the cache for read accesses. The CDACNTL register is shown in Figure 9-10.

DCR 351 Access: Read/Write

	0	1	2 3	4 5	6 12	13 15	16 19 20 23	24 27	28	29	30 31
R W	T/D	—	CWAY	—	CSET	WORD	PARITY	—	CACHE	R/W	GO
Reset	All zeros										

Figure 9-10. Cache Debug Access Control Register (CDACNTL)

Table 9-13 describes the CDACNTL bits.

Table 9-13. CDACNTL Field Descriptions

Bit	Name	Description
0	T/D	Tag/Data: 0 Data array selected 1 Tag array selected
1	—	Reserved[1]

Table 9-13. CDACNTL Field Descriptions (continued)

Bit	Name	Description
2–3	CWAY	Cache Way Specifies the cache way to be selected
4–5	—	Reserved[1]
6–12	CSET	Cache Set: Specifies the cache set to be selected
13–15	WORD	Word (Data array access only, I or D cache) Specifies one of eight words of selected set
16–23	PARITY/EDC CHECK BITS	Parity check bits[2] (I or D cache) Parity Mode (L1CSR[0,1][D,I]CEDT = 00): Data array: Byte parity bits. One bit per data byte. bit 16: Parity for byte 0, bit 17: Parity for byte 1.... bit 23: Parity for byte 7. Tag Array: parity check bits for tag. Bit 16 corresponds to parity of tag[0:11]. Bit 17 corresponds to parity of tag[12:21]+V. bits 18:23 reserved. EDC Mode (L1CSR[0,1][D,I]CEDT = 01): Dcache Data array: Byte parity bits. One bit per data byte. bit 16: Parity for byte 0, bit 17: Parity for byte 1.... bit 23: Parity for byte 7. Icache Data Array: parity check bits for data. Bits 16:23 correspond to p_dchk[0:7] (See Table 9-8). Tag Array: parity check bits for tag. Bits 16:21 correspond to p_tchk[0:5] (See Table 9-7). bits 22:23 reserved.
24–27	—	Reserved[1]
28	CACHE	Cache Select Specifies the cache to be selected 0 Selects the data cache for the operation. 1 Selects the instruction cache for the operation.
29	R/W	Read / Write: 0 Selects write operation. Write the data in the CDADATA register to the location specified by this CDACNTL register. 1 Selects read operation. Read the cache memory location specified by this CDACNTL register and store the resulting data in the CDADATA register and store the parity bits in this CDACNTL register.
30–31	GO	GO command bits 00 Inactive or complete (no action taken) hardware sets GO=00 when an operation is complete 01 Read or write cache memory location specified by this CDACNTL register. 1x Reserved

---

[1] These bits are not implemented and should be written zero for future compatibility.

[2] Cache parity checkers assume odd parity when using parity protection. EDC coding is used otherwise.

9.19.3.1 Cache Debug Access Data Register (CDADATA)

The cache debug access data register (CDADATA) contains the SRAM data for a debug access. The same register is used for Tag and Data SRAM read and write operations for both caches. Note that a single 32-bit word is accessed. Accessing an entire 64-bit double word requires two passes. The CDADATA register is shown in Figure 9-11.

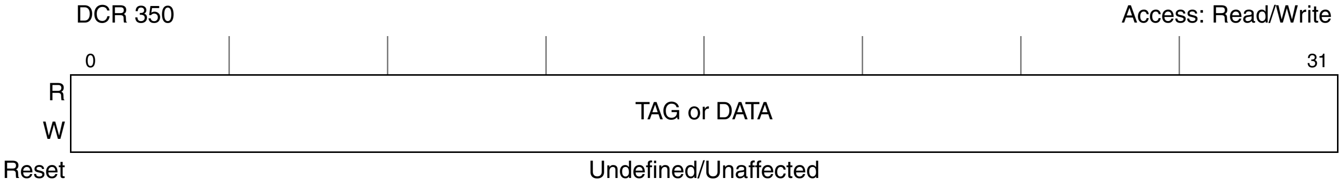


Figure 9-11. Cache Debug Access Data Register (CDADATA)

Table 9-14 describes the CDADATA bits.

Table 9-14. CDADATA Field Descriptions

Bit	Name	Description
0–31	TAG	TAG Array Access Data When accessing the tag array of either cache, it has the following values: 0–21 Tag compare bits 22 Reserved 23 Valid bit 24–27 Lock bits. These four bits should have the same value: 1 Locked 0 Unlocked. 28–30 Dirty bits (data cache only). These three bits should have the same value: 1 Dirty 0 Clean
	DATA	DATA Array Access Data (Bytes 0–3 of the selected word) When accessing the data array of either cache, it has the following values: 0–7 byte 0 8–15 byte 1 16–23 byte 2 24–31 byte 3

9.20 Hardware Debug (Cache) Control Register 0

Hardware debug control register 0 is used to disable certain cache features for hardware debug purposes. This register is not intended for normal user use. The HDBCR0 register is accessed using a **mfspr** or **mtspr** instruction. The SPR number for HDBCR0 is 976 in decimal. The HDBCR0 register is shown in Figure 9-11.

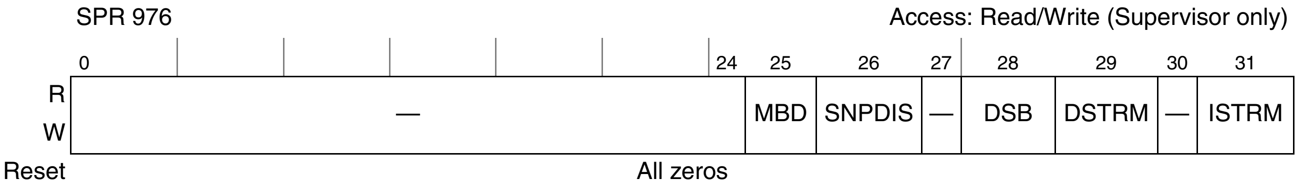


Figure 9-12. Hardware Debug Control Register 0 (HDBCR0)

Table 9-15 describes the HDBCR0 bits.

Table 9-15. HDBCR0 Field Descriptions

Bits	Name	Description
0-24	—	Reserved[1]
25	MBD	Msync/Mbar Broadcast Disable 0 - msync/mbar broadcasting is enabled. **p_sync_req_out** asserted normally and **p_sync_ack_in** is used to terminate msync and mbar MO=0,1 instruction execution 1 - msync/mbar broadcasting is disabled. **p_sync_req_out** remains negated, and **p_sync_ack_in** is ignored and not used to terminate msync and mbar MO=0,1 instruction execution. Note: MBD settings have no effect on the operation of **p_sync_req_in** and **p_sync_ack_out**. Normal handshaking and completion of the synchronization request input will be performed.
26	SNPDIS	Snoop Disable 0 - Snooping is not disabled. Snoops are processed normally according to the settings of L1CSR0$_{DCE}$. 1 - Snoop lookups are disabled. Snoops are processed in the same manner as when the data cache is disabled, i.e null responses are generated and no snoop lookups are performed.
27	—	Reserved[1]
28	DSB	Disable Store Buffer 0 - Store buffer enabled 1 - Store buffer disabled
29	DSTRM	Disable Data Cache Streaming 0 - DCache streaming is enabled 1 - DCache streaming is disabled
30	—	Reserved[1]
31	ISTRM	Disable Instruction Cache Streaming 0 - ICache streaming is enabled 1 - ICache streaming is disabled

[1] These bits are not implemented and should be written with zero for future compatibility.

9.21 Hardware Debug (Cache) Coherency

Hardware cache coherency is supported to allow for dual-core or CPU + I/O coherency. The cache must operate in writethrough mode for those pages of memory requiring coherency operations. Coherency is maintained by the use of snoop invalidation commands provided to the CPU through a dedicated snoop interface port. Snooping is only performed while the data cache is enabled (L1CSR0[DCE] = 1). Figure 9-13 shows an abstract block diagram of the structure.

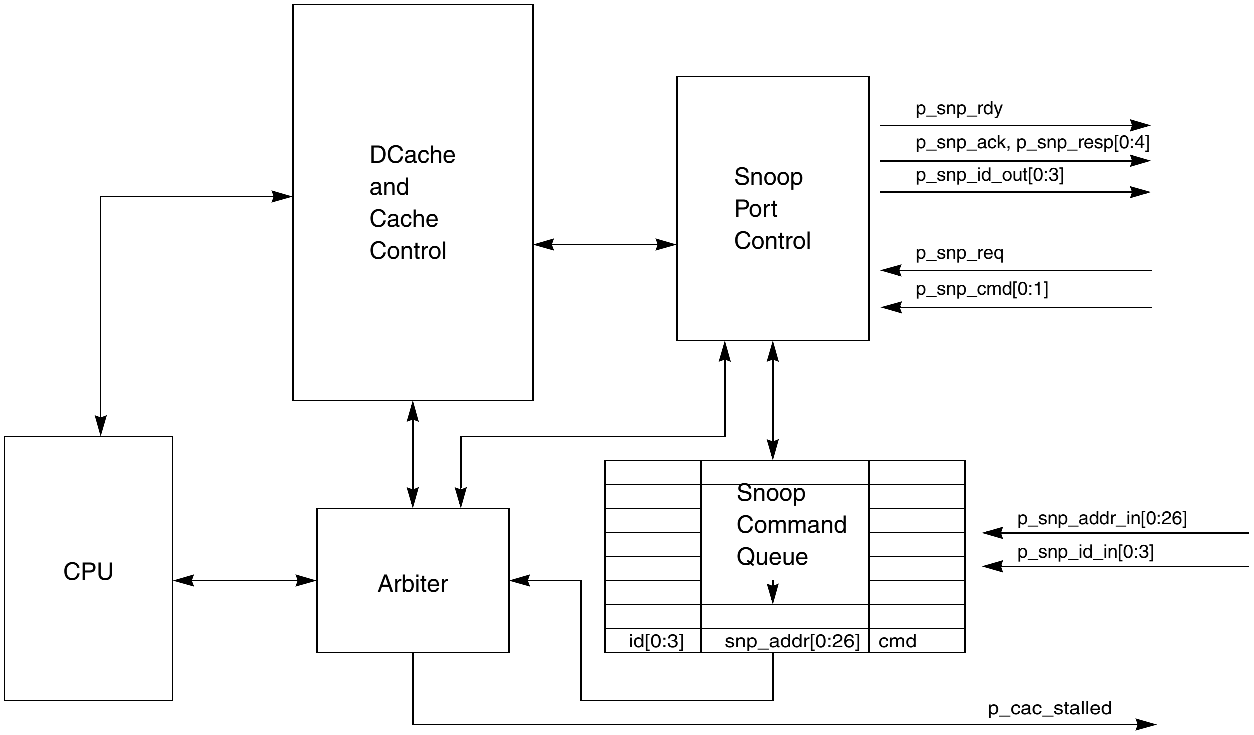


Figure 9-13. Snoop Command Port

9.21.1 Coherency Protocol

The cache operates in a 2-state protocol for coherency purposes. The only state a coherent cache line should assume is Valid or Invalid. No Modified or Shared state is supported for coherent cache lines (although modified state is available for non-coherent lines), thus no snoop copyback or intervention operations are required. A snoop invalidation signaling port is provided to receive coherency requests. Snoop invalidation requests are received at the snoop invalidation port, and arbitrate with the CPU for access to the data cache tags for lookup and cache line invalidation. External coherency logic provides snoop invalidation requests to the snoop invalidation port based on the bus activity of other coherent bus masters, and these invalidation requests are later processed and a response provided. Memory regions which require coherency operations must be marked as “memory coherence required” (page’s M bit set) and as “writethrough” (page’s W bit set).

External data accesses by the CPU reflect the value of the M bit of the accessed page on the **p_d_gbl** output. Typically, external coherency logic will monitor external accesses by a CPU (or other agent), and will request invalidation operations to other coherent entities for write accesses which also have **p_d_gbl** asserted. Non-shared data should be placed into pages with the M bit cleared, thus avoiding unnecessary coherency operations.

9.21.2 Snoop Command Port

The snoop command port provides the signaling mechanism between external coherency logic and the snoop request queue. Command requests are received on the **p_snp_cmd[0:1]**, **p_snp_id_in[0:3]**, and **p_snp_addr_in[0:26]** inputs when the **p_snp_req** signal is properly asserted, and responses to snoop command requests are provided on the **p_snp_ack**, **p_snp_resp[0:4]**, and **p_snp_id_out[0:3]** outputs.

Snoop invalidation requests provide the physical address of the data to be invalidated (**p_snp_addr_in[0:26]**), along with a four-bit ID field (**p_snp_id_in[0:3]**) which flows through the command pipeline and is returned on the **p_snp_id_out[0:3]** output port along with the completion status provided on **p_snp_resp[0:4]** when **p_snp_ack** is asserted.

The **p_snp_rdy** output signal provides a handshaking mechanism for flow control of snoop requests to prevent overflow of the internal snoop queue which buffers incoming snoop requests from the snoop command port prior to cache tag lookups and updates. Negation of **p_snp_rdy** indicates that another snoop command port request will not be accepted due to resource constraints in the snoop pipeline.

Refer to Section 11.2.9, “Coherency Control Signals,” for details on the operating protocol of the snoop command port.

The command value is stored in the snoop queue along with the snoop address and snoop ID value. Table 9-16 shows the definitions of the **p_snp_cmd[0:1]** encodings.

Table 9-16. p_snp_cmd[0:1] Snoop Command Encoding

p_snp_cmd[0:1]	Response Type
00	Null—no status bit operation performed; lookup is performed
01	INV—invalidate matching cache entry
10	SYNC—synchronize snoop queue
11	Reserved

The NULL command is used for testing interface handshaking and other status gathering purposes. The NULL command performs a snoop lookup operation, but performs no actual cache tag or status modifications (even in the presence of tag parity or EDC errors). The INV command causes a snoop lookup and subsequent invalidation of a matching cache line. The SYNC command causes the snoop queue to be emptied with highest priority relative to CPU requests.

Table 9-16 shows the definitions of the **p_snp_resp[0:4]** encodings.

Table 9-17. p_snp_resp[0:4] Snoop Response Encoding

p_snp_resp[0:4][1]	Response Type
000cc	NULL—no operation performed or no matching cache entry
001cc	AutoInv—AutoInvalidation performed on clean unlocked lines with tag parity errors

Table 9-17. p_snp_resp[0:4] Snoop Response Encoding (continued)

p_snp_resp[0:4][1]	Response Type
010cc	ERROR—Error in processing a snoop request due to TAG parity error. For NULL commands, a tag parity error occurred and no hit to a tag without error occurred. No modification of cache entries, no machine check generated internally. For INV commands, a) possible invalidation of locked line with tag parity error occurred, or b) dirty line left valid with tag parity error, or c) no true hit occurred, and one or more lines reported tag parity errors. Machine check generated internally.
01100	SYNC—Sync completed, snoop queue synchronized
100cc	HIT Clean- matching unlocked cache entry found
101cc	HIT Dirty- matching unlocked dirty cache entry found
110cc	HIT Locked—matching clean locked cache entry found
111cc	HIT Dirty Locked—matching dirty locked cache entry found

[1] cc = # collapsed requests
00 = no collapsing
01 = two requests combined
10 = three requests combined
11 = four requests combined

The NULL response indicates that either there was no matching cache entry found for a null or invalidate command or the cache was disabled when the request was originally made. The HIT responses indicate that a matching cache entry was found. The SYNC response indicates all previous entries in the snoop queue were emptied. The ERROR response indicates that an error occurred in processing a snoop request due to a cache tag parity error. The AutoInv response indicates that one or more cache lines with tag parity errors was/were invalidated.

Responses for a null command are either NULL, HIT, or ERROR. Responses for an INV command are either NULL (no hit occurred or cache is disabled), HIT (a matching entry was found and invalidated), or ERROR (a tag parity error was found and left valid, no guarantee of the command success). Responses for a Sync command are SYNC completed.

9.21.3 Snoop Request Queue

The snoop request queue provides a queueing mechanism between the snoop command port and the cache. As requests are accepted from the snoop invalidate port, they are queued into an 8-deep FIFO queue for arbitration to the cache for tag and status lookup and conditional status clearing.

Snoops can be collapsed within the queue under certain circumstances to minimize the number of invalidation lookups performed. When two consecutive snoop requests refer to the same cache line, they are collapsed (timing permitting) into a single snoop invalidation cycle. Collapsed entries are indicated complete via an encoding of the **p_snp_resp[0:4]** status outputs.

Snoop invalidation requests have a lower priority than CPU data accesses or change of flow accesses when only a single queue entry is occupied. This allows for some optimization in cycle-stealing of the tag array from the CPU in an attempt to minimize CPU stalls. The snoop invalidation request priority is raised when a snoop sync command is received on the snoop command port or when a sync request is generated on the synchronization port (**p_sync_req_in**), regardless of the number of active queue entries.

9.21.4 Snoop Lookup Operation

Entries in the snoop request queue are processed in-order after arbitrating for the cache tag and status bit arrays. Once the CPU has been stalled from performing further tag accesses, the snoop request queue is processed by performing a tag lookup and a subsequent status bit write to clear the valid bit of a matching valid entry. Invalidation hits require two tag array accesses to read and then update the valid bit. A subsequent snoop lookup may be pipelined while the first lookup of a pair of lookups is being processed to determine a hit/miss condition. In this manner, a pair of hitting invalidation requests block the CPU for a total of 5 cycles. A single snoop lookup requires 3 cycles of latency on a miss and 4 cycles on a hit prior to allowing the CPU to resume cache accesses. If the snoop queue contains enough entries, snoop read and write accesses to the cache tag are pipelined, and the total blockage is 3 × number_of_hits + number_of_misses + 1. In certain cases where the CPU has pipelined one or more cache misses, initial snoop accesses are interlaced with CPU tag accesses prior to assuming highest priority in order to allow for proper operation of linefill and copyback operations initiated by the CPU.

As entries are removed from the queue and the invalidation lookups are performed, the results of the lookups are provided on the response output signals, along with the original request ID.

9.21.5 Snoop Errors

Errors can occur during snoop lookup operations and are signaled on the snoop response output port. Tag parity errors that prevent an accurate hit/miss determination on the snoop request address may result in an error response signaled via **p_snp_resp[0:4]**. They may also result in a machine check to the CPU for the INV command if a locked line was invalidated, if a line was dirty and not invalidated, or if a tag parity error occurred and no hit occurred to a line without error. When such a tag parity error occurs, the invalidation does not occur to the line(s) with error. The snoop queue continues to be serviced, and the machine check is not necessarily recoverable. A checkstop condition does not occur, however. In this respect, it is treated similarly to a non-maskable interrupt, and MSR[RI] should be used accordingly by software.

9.21.6 Snoop Collisions

Snoop requests may collide with an outstanding or pending cache linefill.

Because there is no particular guarantee of the precise time an actual snoop invalidation lookup occurs relative to a cache linefill request, in some instances the CPU may be in the process of filling a line corresponding to a snoop invalidate request. In this case, the snoop causes the linefills to be marked such that they are not loaded into the cache. However, load miss operations that are in progress may use the data as it returns. The responses for these collisions is based on the state that the cache line would have taken if the linefill had completed successfully.

Snoop requests should not collide with dirty line copyback or flush operations because the coherent pages must be marked as write through required. These snoop collisions are ignored.

9.21.7 Snoop Synchronization

Synchronization of the snoop queue occurs under two conditions: a synchronization port sync request and a snoop command port sync request.

9.21.7.1 Synchronization Port Request

Assertion of the **p_sync_req_in** signal causes the snoop queue to assume highest priority and be flushed. It is assumed that the system stops generating snoop requests during a synchronization of the queue to allow it to drain. However, if snoop requests continue to be received, the acknowledgement of the synchronization request is delayed until the queue finally drains to the point that all queue entries that were present prior to the recognition of the sync request have been serviced.

In general, the synchronization port is expected to be utilized to handshake execution of **msync** instructions from an alternate CPU. Additional snoop requests do not typically occur until the synchronization handshake is complete, since no further bus writes will be requested by the alternate CPU. However, if additional coherency traffic occurs due to another alternate master, it follows the normal queueing process and does not block the eventual assertion of the **p_sync_ack_out** signal.

9.21.7.2 Snoop Command Port Request

Receiving a snoop command port snoop sync request encoded via the **p_snp_cmd[0:1]** inputs causes the snoop queue to assume highest priority and to be flushed to the point the command has reached the head of the queue and been acknowledged. After the command has been completed, snoop queue priority reverts to normal operation, unless another snoop sync command has been received and placed into the queue, in which case snoop queue priority remains elevated until all snoop sync commands have been processed from the queue.

9.21.8 Starvation Control

To avoid starvation of a higher priority CPU due to a continuous stream of snoop requests from a lower priority master which block CPU forward progress, some form of starvation control is desired. This is implemented with a forward progress counter that tracks the number of contiguous cycles the CPU has been prevented from accessing the cache due to snoop command port access requests. Once the count has been exceeded, the CPU regains highest priority for one access cycle. A similar counter exists for the snoop queue to allow for periodic snoop request processing when the queue holds only a single entry. Each counter is 4 bits and causes a priority inversion to occur for tag access upon timeout.

The presence of one or more sync commands in the snoop queue when the counter expires delays the priority inversion until the queue has been emptied up to the point that the sync(s) have been completed. Subsequent syncs received while the starvation timeout is being postponed may also prevent the priority inversion after the original sync(s) have been completed if additional snoops have been queued during the sync command processing. This is not normally expected to occur in a typical system.

In addition, external logic may be used to implement additional safeguards by monitoring the **p_cac_stalled** output, which indicates that the CPU has a pending cache access request blocked due to snoop access activity.

9.21.9 Queue Flow Control

To avoid overflow of the snoop queue, the **p_snp_rdy** output is provided to indicate whether an additional snoop command port request will be accepted on the following clock cycle. When negated, no further command requests can be honored until a snoop queue entry becomes available.

To provide for flow control of CPU-generated snoop requests to another CPU's queue, the p_stall_bus_gwrite input is provided. This input suspends further bus activity that is requesting a global write cycle. Other bus traffic is not affected.

9.21.10 Snooping in Low Power States

If the clock is running, snooping remains enabled while in the waiting or halted states. Snoops should only be issued to the core complex while the core is in the normal, halted, or waiting states and both the **p_stop** and **p_stopped** signals are negated.

When a request is made to enter stop mode via the assertion of **p_stop**, the **p_snp_rdy** output is negated. While the core complex is in the stopped (power-down) state, bus snooping is disabled, and the **p_snp_rdy** output is held negated. Snoop requests are processed around the assertion of the stop mode entry request (assertion of **p_stop**) per the normal protocol associated with **p_snp_rdy** negation, including acceptance of a snoop request during a small interval around **p_snp_rdy** negation. Therefore, additional snoop operations may need to occur prior to entering the stopped state. All snoop queue entries are processed prior to the assertion of **p_stopped**.

Chapter 10
Memory Management Unit

10.1 Overview

The e200z7 memory management unit (MMU) is a 32-bit Power ISA embedded category compliant implementation, with the following feature set:

- Freescale EIS MMU architecture compliant
- Translates from 32-bit effective to 32-bit real addresses
- 64-entry fully associative TLB with support for twenty-three page sizes (1 KB, 2 KB, 4 KB, 8 KB, 16 KB, 32 KB, 64 KB, 128 KB, 256 KB, 512 KB, 1 MB, 2 MB, 4 MB, 8 MB, 16 MB, 32 MB, 64 MB, 128 MB, 256 MB, 512 MB, 1 GB, 2 GB, 4 GB)
- Hardware assist for TLB miss exceptions
- Software managed by **tlbre**, **tlbwe**, **tlbsx**, **tlbsync**, and **tlbivax** instructions
- Support for external control of entry matching for a subset of TID values to support non-intrusive runtime mapping modifications

10.2 Effective to Real Address Translation

This section describes effective to real address translation. It contains the following subsections:

- Section 10.2.1, "Effective Addresses"
- Section 10.2.2, "Address Spaces"
- Section 10.2.3, "Process ID"
- Section 10.2.4, "Translation Flow"
- Section 10.2.5, "Permissions"
- Section 10.2.6, "Restrictions on 1-KB and 2-KB Page Size Usage"

10.2.1 Effective Addresses

Instruction accesses are generated by sequential instruction fetches or due to a change in program flow (branches and interrupts). Data accesses are generated by load, store, and cache management instructions. The e200 instruction fetch, branch, and load/store units generate 32-bit effective addresses. The MMU translates this effective address to a 32-bit real address that is then used for memory accesses.

The Power ISA embedded category architecture divides the effective (virtual) and real (physical) address space into pages. The page represents the granularity of effective address translation, permission control, and memory/cache attributes. The MMU supports twenty-three page sizes (1 KB, 2 KB, 4 KB, 8 KB, 16

KB, 32 KB, 64 KB, 128 KB, 256 KB, 512 KB, 1 MB, 2 MB, 4 MB, 8 MB, 16 MB, 32 MB, 64 MB, 128 MB, 256 MB, 512 MB, 1 GB, 2 GB, 4 GB). For an effective to real address translation to exist, a valid entry for the page containing the effective address must be in a translation lookaside buffer (TLB). Addresses for which no TLB entry exists (a TLB miss) cause instruction or data TLB errors.

10.2.2 Address Spaces

Instruction accesses are generated by sequential instruction fetches or due to a change in program flow (branches and interrupts). Data accesses are generated by load, store, and cache management instructions.

The Power ISA embedded category architecture defines two effective address spaces for instruction accesses and two effective address spaces for data accesses. The current effective address space for instruction or data accesses is determined by the value of MSR[IS] and MSR[DS], respectively. The address space indicator (the value of either MSR[IS] or MSR[DS], as appropriate) is used in addition to the effective address generated by the processor for translation into a physical address by the TLB mechanism. Because MSR[IS] and MSR[DS] are both cleared to 0 when an interrupt occurs, an address space value of 0b0 can be used to denote interrupt-related address spaces (or possibly all system software address spaces), and an address space value of 0b1 can be used to denote non interrupt-related (or possibly all user address spaces) address spaces.

The address space associated with an instruction or data access is included as part of the virtual address in the translation process (AS). The **p_tc[1]** interface signal indicates the appropriate address space.

10.2.3 Process ID

The Power ISA embedded category architecture defines that a process ID (PID) value is associated with each effective address (instruction or data) generated by the processor. At the Power ISA embedded category level, a single PID register is defined as a 32-bit register, and it maintains the value of the PID for the current process. This PID value is included as part of the virtual address in the translation process (PID0). For the e200z7 MMU, the PID is 8 bits in length. The most-significant 24 bits are unimplemented and read as 0. The **p_pid0[0:7]** interface signals indicate the current process ID.

10.2.4 Translation Flow

The effective address, concatenated with the address space value of the corresponding MSR bit (MSR[IS] or MSR[DS], is compared to the appropriate number of bits of the EPN field (depending on the page size) and the TS field of TLB entries. If the contents of the effective address plus the address space bit matches the EPN field and TS bit of the TLB entry, that TLB entry is a candidate for a possible translation match. In addition to a match in the EPN field and TS, a matching TLB entry must match with the current Process ID of the access (in PID0), or have a TID value of ‘0’, indicating the entry is globally shared among all processes.

Figure 10-1 shows the translation match logic for the effective address plus its attributes, collectively called the virtual address, and how it is compared with the corresponding fields in the TLB entries.

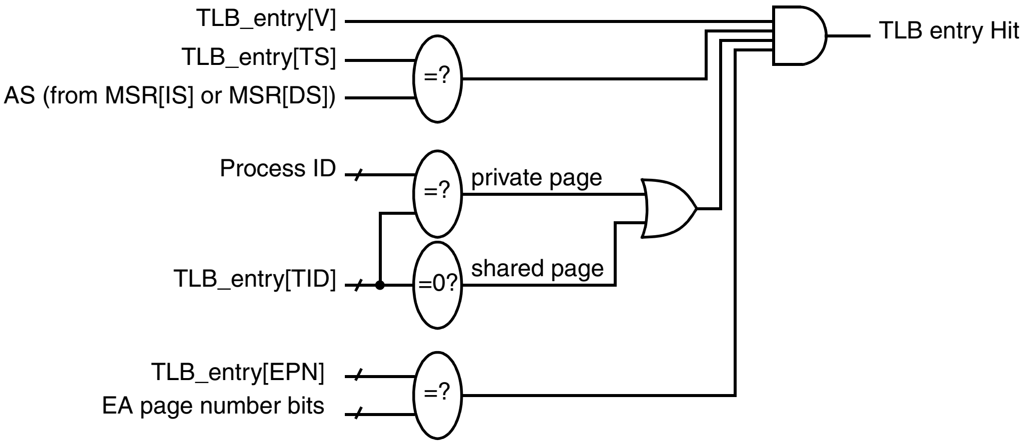


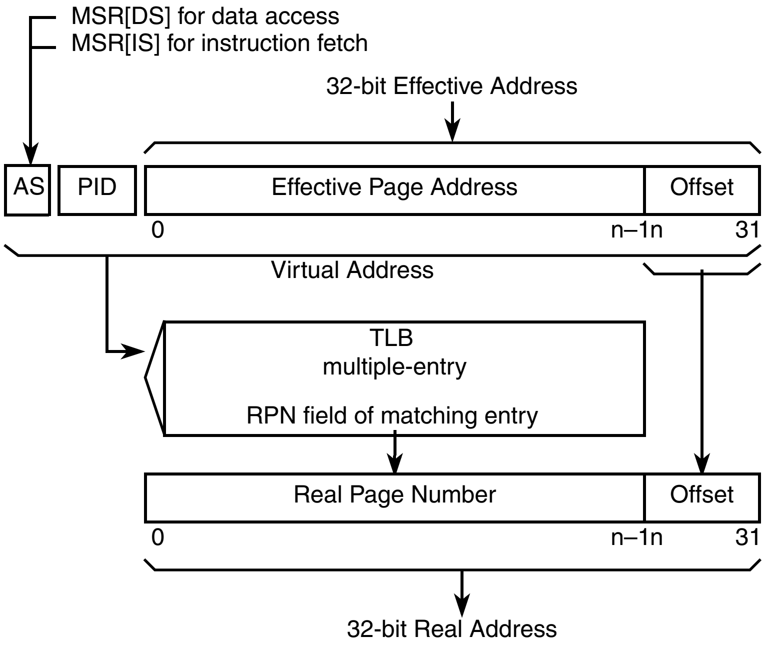
Figure 10-1. Virtual Address and TLB-Entry Compare Process

The page size defined for a TLB entry determines how many bits of the effective address are compared with the corresponding EPN field in the TLB entry as shown in Table 10-1. On a TLB hit, the corresponding bits of the real page number (RPN) field are used to form the real address.

Table 10-1. Page Size Field Encodings and EPN Field Comparison

SIZE Field	Page Size (2^{SIZE}KB)	EA to EPN Comparison
0b00000	1 KB	EA[0–21] =? EPN[0–21]
0b00001	2 KB	EA[0–20] =? EPN[0–20]
0b00010	4 KB	EA[0–19] =? EPN[0–19]
0b00011	8 KB	EA[0–18] =? EPN[0–18]
0b00100	16 KB	EA[0–17] =? EPN[0–17]
0b00101	32 KB	EA[0–16] =? EPN[0–16]
0b00110	64 KB	EA[0–15] =? EPN[0–15]
0b00111	128 KB	EA[0–14] =? EPN[0–14]
0b01000	256 KB	EA[0–13] =? EPN[0–13]
0b01001	512 KB	EA[0–12] =? EPN[0–12]
0b01010	1 MB	EA[0–11] =? EPN[0–11]
0b01011	2 MB	EA[0–10] =? EPN[0–10]
0b01100	4 MB	EA[0–9] =? EPN[0–9]
0b01101	8 MB	EA[0–8] =? EPN[0–8]
0b01110	16 MB	EA[0–7] =? EPN[0–7]
0b01111	32 MB	EA[0–6] =? EPN[0–6]
0b10000	64 MB	EA[0–5] =? EPN[0–5]
0b10001	128 MB	EA[0–4] =? EPN[0–4]
0b10010	256 MB	EA[0–3] =? EPN[0–3]
0b10011	512 MB	EA[0–2] =? EPN[0–2]
0b10100	1 GB	EA[0–1] =? EPN[0–1]
0b10101	2 GB	EA[0] =? EPN[0]
0b10110	4GB	(none)

On a TLB hit, the generation of the physical address occurs as shown in Figure 10-2.



NOTE: n = 32 – $\log_2$(page size)
n ≤ 22
n = 20 for 4-KB page size.

Figure 10-2. Effective to Real Address Translation Flow

10.2.5 Permissions

An operating system may restrict access to virtual pages by selectively granting permissions for user mode read, write, and execute, and supervisor mode read, write, and execute on a per page basis. These permissions can be set up for a particular system (for example, program code might be execute-only, data structures may be mapped as read/write/no-execute) and can also be changed by the operating system based on application requests and operating system policies.

The UX, SX, UW, SW, UR, and SR access control bits are provided to support selective permissions (access control):

- SR—Supervisor read permission. Allows loads and load-type cache management instructions to access the page while in supervisor mode (MSR[PR = 0]).
- SW—Supervisor write permission. Allows stores and store-type cache management instructions to access the page while in supervisor mode (MSR[PR = 0]).
- SX—Supervisor execute permission. Allows instruction fetches to access the page and instructions to be executed from the page while in supervisor mode (MSR[PR = 0]).
- UR—User read permission. Allows loads and load-type cache management instructions to access the page while in user mode (MSR[PR = 1]).

- UW—User write permission. Allows stores and store-type cache management instructions to access the page while in user mode (MSR[PR = 1]).
- UX—User execute permission. Allows instruction fetches to access the page and instructions to be executed from the page while in user mode (MSR[PR = 1]).

If the translation match was successful, the permission bits are checked as shown in Figure 10-3. If the access is not allowed by the access permission mechanism, the processor generates an Instruction or Data Storage interrupt (ISI or DSI). The current privilege level of an access is signaled to the MMU with the CPU's **p_tc[0]** output signal.

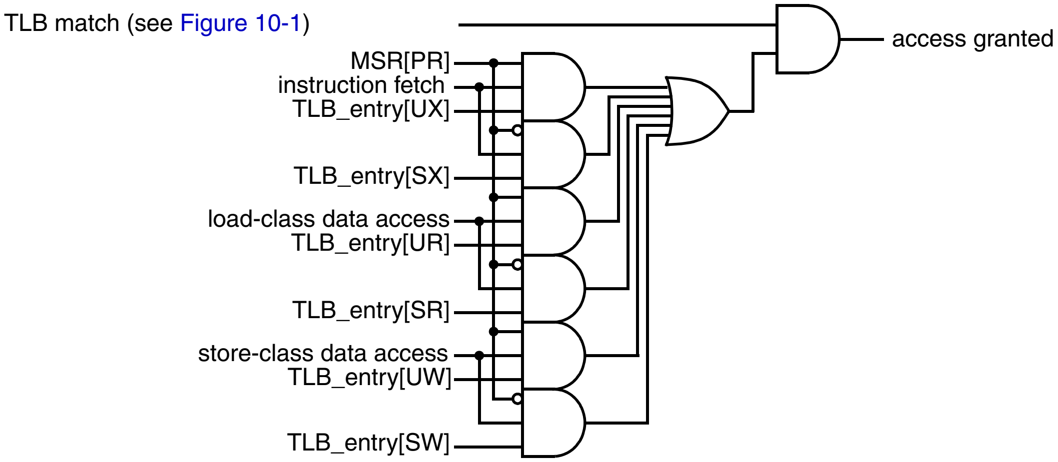


Figure 10-3. Granting of Access Permission

10.2.6 Restrictions on 1-KB and 2-KB Page Size Usage

Because of certain implementation limitations regarding coherency lookup operations (lookup is done by physical address), the low order virtual address bits used to index the cache must match the corresponding physical address bit value(s) if 1-KB or 2 KB pages are used. These bits are A[20–21] for 1-KB pages and A20 for 2-KB pages. For example, if logical page X maps to physical page P, then X and P must have the same values of A[20–21] for 1-KB pages, and A20 for 2-KB pages. This restriction must be followed for proper CPU operation.

10.3 Translation Lookaside Buffer

The Freescale EIS architecture defines support for zero or more TLBs in an implementation, each with its own characteristics, and provides configuration information for software to query the existence and structure of the TLB(s) through a set of special purpose registers: MMUCFG, TLB0CFG, TLB1CFG, etc. By convention, TLB0 is used for a set associative TLB with fixed page sizes, TLB1 is used for a fully associative TLB with variable page sizes, and TLB2 is arbitrarily defined by an implementation. The e200z7 MMU supports a TLB which is fully associative and supports variable page sizes, thus it corresponds to TLB1.

TLB1 consists of a 64-entry, fully associative CAM array with support for 23 page sizes. To perform a lookup, the CAM is searched in parallel for a matching TLB entry. The contents of this TLB entry are then concatenated with the page offset of the original effective address. The result constitutes the real (physical) address of the access.

A hit to multiple TLB entries is considered to be a programming error. If this occurs, the TLB generates an invalid address but an exception will not be reported.

Table 10-2 shows the TLB entry bit definitions.

Table 10-2. TLB Entry Bit Definitions

Field	Comments
V	Valid bit for entry
TS	Translation address space (compared against AS bit)
TID[0–7]	Translation ID (compared against PID0 or ‘0’)
EPN[0–21]	Effective page number (compared against effective address)
RPN[0–21]	Real page number (translated address)
SIZE[0–34]	Page size (see Table 10-1)
SX, SW, SR	Supervisor execute, write, and read permission bits
UX, UW, UR	User execute, write, and read permission bits
WIMGE	Translation attributes (write-through required, cache-inhibited, memory coherence required, guarded, endian)
U0–U3	User bits—used only by software
IPROT	Invalidation protect
VLE	VLE page indicator

10.4 Configuration Information

Information about the configuration for a given MMU implementation is available to system software by reading the contents of the MMU configuration SPRs. These SPRs describe the architectural version of the MMU, the number of TLB arrays, and the characteristics of each TLB array.

10.4.1 MMU Configuration Register (MMUCFG)

The MMU configuration register (MMUCFG), shown in Figure 10-4, is a 32-bit read-only register that provides information about the configuration of the e200z7 MMU design.

SPR 1015 Access: Read only

	0–7	8–14	15–16	17–20	21–25	26–27	28–29	30–31
R	—	RASIZE	—	NPIDS	PIDSIZE	—	NTLBS	MAVN
W								
Reset								

Figure 10-4. MMU Configuration Register (MMUCFG)

Table 10-3 describes the MMUCFG bits.

Table 10-3. MMUCFG Field Descriptions

Bits	Name	Function
0–7 [32–39]	—	Reserved
8–14 [40–46]	RASIZE	Number of Bits of Real Address supported 0100000 This version of the MMU implements 32 real address bits
15–16 [47–48]	—	Reserved
17–20 [49–52]	NPIDS	Number of PID Registers 0001 This version of the MMU implements one PID register (PID0)
21–25 [53–57]	PIDSIZE	PID Register Size 00111 PID registers contain 8 bits in this version of the MMU
26–27 [58–59]	—	Reserved
28–29 [60–61]	NTLBS	Number of TLBs 01 This version of the MMU implements two TLB structures: a null TLB0 and a fully-associative TLB for TLB1
30–31 [62–63]	MAVN	MMU Architecture Version Number 00 This version of the MMU implements version 1.0 of the Freescale EIS MMU architecture

10.4.2 TLB0 Configuration Register (TLB0CFG)

The TLB0 configuration register (TLB0CFG) is a 32-bit read-only register that provides information about the configuration of TLB0. Because the e200z7 MMU design does not implement TLB0, this register reads as all ‘0’. It is supplied to allow software to query it in a fashion compatible with other Freescale EIS designs. The TLB0CFG register is shown in Figure 10-5.

SPR 688 Access: Read only

	0 7	8 11	12 15	16	17	18	19	20 31
R	ASSOC	MINSIZE	MAXSIZE	IPROT	AVAIL	P2PSA	—	NENTRY
W								
Reset	All zeros							

Figure 10-5. TLB0 Configuration Register (TLB0CFG)

The TLB0CFG bits are described in Table 10-4.

Table 10-4. TLB0CFG Field Descriptions

Bits	Name	Function
0–7 [32–39]	ASSOC	Associativity 0
8–11 [40–43]	MINSIZE	Minimum Page Size 0

Table 10-4. TLB0CFG Field Descriptions (continued)

Bits	Name	Function
12–15 [44–47]	MAXSIZE	Maximum Page Size 0
16 [48]	IPROT	Invalidate Protect Capability 0 Not present in TLB0
17 [49]	AVAIL	Page Size Availability 0 No variable page sizes available
18 [50]	P2PSA	Power-of-2 Page Size Availability 0 No odd powers of 2 page sizes are supported
19 [51]	—	Reserved[1]
20–31 [52–63]	NENTRY	Number of Entries 0 TLB0 contains 0 entries

[1] These bits are not implemented and will be read as zero.

10.4.3 TLB1 Configuration Register (TLB1CFG)

The TLB1 configuration register (TLB1CFG) is a 32-bit read-only register that provides information about the configuration of TLB1 in the e200z7 MMU. Figure 10-6 shows the TLB1CFG register.

SPR 689 Access: Read only

	0–7	8–11	12–15	16	17	18	19	20–31
R	ASSOC	MINSIZE	MAXSIZE	IPROT	AVAIL	P2PSA	—	NENTRY
W								
Reset								

Figure 10-6. TLB1 Configuration Register (TLB1CFG)

The TLB1CFG bits are described in Table 10-5.

Table 10-5. TLB1CFG Field Descriptions

Bits	Name	Function
0–7 [32–39]	ASSOC	Associativity 0x40 Indicates that TLB1 associativity is 64
8–11 [40–43]	MINSIZE	Minimum Page Size 0x0 Smallest page size is 1 KB
12–15 [44–47]	MAXSIZE	Maximum Page Size 0xB Largest page size is 4 GB
16 [48]	IPROT	Invalidate Protect Capability 1 Invalidate protect capability is supported in TLB1
17 [49]	AVAIL	Page Size Availability 1 All page sizes between MINSIZE and MAXSIZE are supported

Table 10-5. TLB1CFG Field Descriptions (continued)

Bits	Name	Function
18 [50]	P2PSA	Power-of-2 Page Size Availability 1 All odd powers of 2 page sizes between MINSIZE and MAXSIZE are supported (2K, 8K, 32K, etc.)
19 [51]	—	Reserved
20–31 [52–63]	NENTRY	Number of Entries 0x40 Indicates that TLB1 contains 64 entries

10.5 Software Interface and TLB Instructions

The TLB is accessed indirectly through several 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 TLB. Data is read from the TLB into the MAS registers with a **tlbre** (TLB read entry) instruction. Data is written to the TLB from the MAS registers with a **tlbwe** (TLB write entry) instruction.

Certain fields of the MAS registers are also written by hardware when an Instruction TLB Error or Data TLB Error interrupt occurs.

On a TLB Error interrupt, the MAS registers are written by hardware with the proper EA, default attributes (TID, WIMGE, permissions, etc.), and TLB selection information, and an entry in the TLB to replace. Software manages this entry selection information by updating a replacement entry value during TLB miss handling. Software must provide the correct RPN and permission information in one of the MAS registers before executing a **tlbwe** instruction.

On taking a DSI or ISI interrupt, software should update the search PID (SPID) and search address space (SAS) fields in the MAS registers using PID0, and appropriate MSR[IS] or MSR[DS] values which were used when the DSI or ISI exception was recognized. During the interrupt handler, software can issue a TLB search instruction (**tlbsx**), which uses the SPID field along with the SAS field, to determine the entry related to the DSI or ISI exception. (It is possible that the entry which caused the DSI or ISI interrupt no longer exists in the TLB by the time the search occurs if a TLB invalidate or replacement removes the entry between the time the exception is recognized and when the **tlbsx** is executed.)

The **tlbre**, **tlbwe**, **tlbsx**, **tlbivax**, and **tlbsync** instructions are privileged.

10.5.1 TLB Read Entry Instruction (tlbre)

The TLB read entry instruction causes the content of a single TLB entry to be placed in the MMU assist registers. The entry is specified by the TLBSEL and ESEL fields of the MAS0 register. The entry contents are placed in the MAS1, MAS2, and MAS3 registers. See Table 10-15 for details on how MAS register fields are updated.

tlbre

tlb read entry

31	0	1 1 1 0 1 1 0 0 1 0	0
0 5	6 20	21 30	31

```
tlb_entry_id = MAS0(TLBSEL, ESEL)
result = MMU(tlb_entry_id)
MAS1, MAS2, MAS3 = result
```

10.5.2 TLB Write Entry Instruction (tlbwe)

The TLB write entry instruction causes the contents of certain fields within the MMU assist registers MAS1, MAS2, and MAS3 to be written into a single TLB entry in the MMU. The entry written is specified by the TLBSEL, and ESEL fields of the MAS0 register.

tlbwe

tlb write entry

31	0	1 1 1 1 0 1 0 0 1 0	0
0 5	6 20	21 30	31

```
tlb_entry_id = MAS0(TLBSEL, ESEL)
MMU(tlb_entry_id) = MAS1, MAS2, MAS3
```

10.5.3 TLB Search Instruction (tlbsx)

The TLB search instruction updates the MMU assist registers conditionally based on success or failure of a lookup of the TLB. The lookup is controlled by an effective address provided by GPR[RB] as specified in the instruction encoding, as well as by the SAS and SPID search fields in MAS6. The values placed into MAS0, MAS1, MAS2, and MAS3 differ depending on a successful or unsuccessful search. See Table 10-15 for details on how MAS register fields are updated.

tlbsx

TLB Search Indexed

tlbsx RA,RB Form X

31	0	RA	RB	1 1 1 0 0 1 0 0 1 0	0
0 5	6 10	11 15	16 20	21 30	31

```
if RA!=0 then EA = GPR(RA) + GPR(RB)
else EA = GPR(RB)
ProcessIDs = MAS6(SPID), 8’b00000000
AS = MAS6(SAS)
VA = AS || ProcessIDs || EA
if Valid_TLB_matching_entry_exists(VA)
then result = see Table 10-15, column labelled “tlbsx hit”
else result = see Table 10-15, column labelled “tlbsx miss”
MAS0, MAS1, MAS2, MAS3 = result
```

10.5.4 TLB Invalidate (tlbivax) Instruction

The TLB invalidate operation is performed whenever a TLB Invalidate Virtual Address Indexed (**tlbivax**) instruction is executed. This instruction invalidates TLB entries which correspond to the virtual address calculated by this instruction. The address is detailed in Table 10-6. No other information except for that shown in Table 10-6 is used for the invalidation (entry AS and TID values are don’t-cared).

Additional information about the targeted TLB entries is encoded in two of the lower bits of the effective address calculated by the **tlbivax** instruction. Bit 28 of the **tlbivax** effective address is the TLBSEL field. This bit should be set to ‘1’ to ensure TLB1 is targeted by the invalidate.Bit 29 of the **tlbivax** effective address is the INV_ALL field. If this bit is set, it indicates that the invalidate operation needs to completely invalidate all entries of TLB1 which are not marked as invalidation protected (IPROT bit of entry set to 1).

The bits of EA used to perform the **tlbivax** invalidation of TLB1 are bits 0–21.

Table 10-6. tlbivax EA Bit Definitions

Bits	Field
0–21	EA[0–21]
22–27	Reserved[1]
28	TLBSEL(1 = TLB1) Should be set to 1 for future compatibility.
29	INV_ALL
30–31	Reserved[1]

---

[1] These bits should be zero for future compatibility. They are ignored.

tlbivax tlbivax

TLB Invalidate Virtual Address Indexed

tlbivax RA,RB Form X

31	0	RA	RB	1 1 0 0 0 1 0 0 1 0	0
0 5	6 10	11 15	16 20	21 30	31

```
if RA!=0 then EA = GPR(RA) + GPR(RB)
else EA = GPR(RB)
VA = EA
```

```
if (Valid_TLB_matching_entry_exists(VA) or INV_ALL) and Entry_IPROT_not_set
then Invalidate entry
```

10.5.5 TLB Synchronize Instruction (tlbsync)

The TLB synchronize instruction is treated as a privileged no-op by the e200z7.

tlbsync tlbsync

TLB Synchronize

tlbsync

31	0	1000110110	0
0 5	6 10 11 15 16 20	21 30	31

10.6 TLB Operations

This section discusses the TLB operations. It consists of the following subsections:

- Section 10.6.1, “Translation Reload”
- Section 10.6.2, “Reading the TLB”
- Section 10.6.3, “Writing the TLB”
- Section 10.6.4, “Searching the TLB”
- Section 10.6.5, “TLB Miss Exception Update”
- Section 10.6.6, “IPROT Invalidation Protection”
- Section 10.6.7, “TLB Load on Reset”
- Section 10.6.8, “The G bit”

10.6.1 Translation Reload

The TLB reload function is performed in software with some hardware assist. This hardware assist consists of the following:

- Five 32-bit MMU assist registers (MAS0–4, MAS6) for support of the **tlbre**, **tlbwe**, and **tlbsx** TLB management instructions.
- Loading of MAS0–2 based upon defaults in MAS4 for TLB miss exceptions. This automatically generates most of the TLB entry.
- 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, or Data Storage Interrupt.
- The **tlbwe** instruction. When **tlbwe** is executed, the new TLB entry contained in MAS0-MAS2 is written into the TLB.

10.6.2 Reading the TLB

The TLB array can be read by first writing the necessary information into MAS0 using **mtspr** and then executing the **tlbre** instruction. To read an entry from the TLB, the TLBSEL field in MAS0 must be set to 01, and the ESEL bits in MAS0 must be set to point to the desired entry. After executing the **tlbre** instruction, MAS1–MAS3 is updated with the data from the selected TLB entry.

10.6.3 Writing the TLB

The TLB1 array can be written by first writing the necessary information into MAS0–MAS3 using **mtspr** and then executing the **tlbwe** instruction. To write an entry into the TLB, the TLBSEL field in MAS0 must be set to 01, and the ESEL bits in MAS0 must be set to point to the desired entry. When the **tlbwe** instruction is executed, the TLB entry information stored in MAS1–MAS3 is written into the selected TLB entry.

10.6.4 Searching the TLB

The TLB can be searched using the **tlbsx** instruction by first writing the necessary information into MAS6. The **tlbsx** instruction searches using EPN[0–21] from the GPR selected by the instruction, SAS (search AS bit) in MAS6, and SPID in MAS6. If the search is successful, the given TLB entry information is loaded into MAS0–MAS3. The valid bit in MAS1 is used as the success flag. If the search is successful, the valid bit in MAS1 is set; if unsuccessful it is cleared. The **tlbsx** instruction is useful for finding the TLB entry that caused a DSI or ISI exception.

10.6.5 TLB Miss Exception Update

When a TLB miss exception occurs, MAS0–MAS3 are updated with the defaults specified in MAS4, and the AS and EPN[0–21] of the access that caused the exception. In addition, the ESEL bits are updated with the replacement entry value.

This sets up all the TLB entry data necessary for a TLB write except for the RPN[0–21], the U0–U3 user bits, and the UX/SX/UW/SW/UR/SR permission bits, all of which are stored in MAS3. Thus, if the defaults stored in MAS4 are applicable to the TLB entry to be loaded, the TLB miss exception handler will only have to update MAS3 via **mtspr** before executing **tlbwe**. If the defaults are not applicable to the TLB entry being loaded, the TLB miss exception handler must update MAS0–MAS2 before performing the TLB write.

10.6.6 IPROT Invalidation Protection

The IPROT bit is used to protect TLB entries from invalidation. TLB entries with IPROT set are not invalidated by a **tlbivax** instruction (even when INV_ALL is indicated), nor by the MMUCSR0[TLB1_FI] control function. The IPROT bit is used to protect interrupt vectors/handlers because the instruction fetch of those vectors must be guaranteed to never take a TLB miss exception.

10.6.7 TLB Load on Reset

During reset, all TLB entries except entry 0 are invalidated. TLB entry 0 is loaded with the values in Table 10-7:

Table 10-7. TLB Entry 0 Values after Reset

Field	Reset Value	Comments
VALID	1	Entry is valid
TS	0	Address space 0
TID[0–7]	0x00	TID value for shared (global) page
EPN[0–21]	value of **p_rstbase[0–21]**	Page address present on **p_rstbase[0:29]**. See Section 11.2.2.5, “Reset Base (p_rstbase[0:29]).”
RPN[0–21]	value of **p_rstbase[0–21]**	Page address present on **p_rstbase[0:29]**. See Section 11.2.2.5, “Reset Base (p_rstbase[0:29]).”
SIZE[0–4]	00010	4 KB page size
SX/SW/SR	111	Full supervisor mode access allowed
UX/UW/UR	111	Full user mode access allowed
WIMG	0100	Cache inhibited, non-coherent
E	value of **p_rst_endmode**	Value present on **p_rst_endmode**. See Section 11.2.2.6, “Reset Endian Mode (p_rst_endmode).”
U0–U3	0000	User bits
IPROT	1	Page is protected from invalidation
VLE	the value of **p_rst_vlemode**	Value present on **p_rst_vlemode signal.**See Section 11.2.2.7, “Reset VLE Mode (p_rst_vlemode).”

10.6.8 The G bit

The G-bit provides protection from bus accesses that can be cancelled due to an exception on a prior uncompleted instruction.

If G = 1 (guarded), these types of accesses must stall (if they miss in the cache) until the exception status of the instruction(s) in progress is known. If G = 0 (unguarded), these accesses may be issued to the bus regardless of the completion status of other instructions. Since the e200z7 does not make requests to the bus for load or store instructions which miss in the cache until it is known that prior instructions will complete without exceptions, proper operation always occurs to guarded storage.

10.7 MMU Control Registers

This section discusses the following registers:

- Section 10.7.1, “Data Exception Address Register (DEAR)”
- Section 10.7.2, “MMU Control and Status Register 0 (MMUCSR0)”
- Section 10.7.3, “MMU Assist Registers (MAS)”

- Section 10.7.4, “MAS Register Updates”

10.7.1 Data Exception Address Register (DEAR)

The data exception address register (DEAR), shown in Figure 10-7, is loaded with the effective address of the data access that results in an Alignment, Data TLB Miss, or DSI exception.

SPR 61 Access: Read/Write

	0–31
R W	Effective Page Address
Reset	Unaffected

Figure 10-7. Data Exception Address Register

The DEAR register can be read or written using the **mfspr** and **mtspr** instructions.

10.7.2 MMU Control and Status Register 0 (MMUCSR0)

The MMU control and status register 0 (MMUCSR0), shown in Figure 10-8, controls the state of the MMU.

SPR 1012 Access: Read/Write

	0–29	30	31
R W	—	TLB1_FI	—
Reset	All zeros		

Figure 10-8. MMU Control and Status Register 0 (MMUCSR0)

The MMUCSR0 bits are described in Table 10-8.

Table 10-8. MMUCSR0—MMU Control and Status Register 0

Bits	Name	Description
0–29 [32–61]	—	Reserved
30 [62]	TLB1_FI	TLB1 flash invalidate 0 No flash invalidate 1 TLB1 invalidation operation When written to a 1, a TLB1 invalidation operation is initiated by hardware. Once complete, this bit is reset to 0. Writing a 1 while an invalidation operation is in progress will result in an undefined operation. Writing a 0 to this bit while an invalidation operation is in progress will be ignored. TLB1 invalidation operations require 3 cycles to complete.
31 [63]	—	Reserved

10.7.3 MMU Assist Registers (MAS)

The e200z7 uses six special purpose registers (MAS0, MAS1, MAS2, MAS3, MAS4, and MAS6) to facilitate reading, writing, and searching the TLBs. The MAS registers can be read or written using the **mfspr** and **mtspr** instructions. The e200z7 does not implement the MAS5 register, present in other Freescale EIS designs, because the **tlbsx** instruction only searches based on a single SPID value.

Figure 10-9 shows the MAS0 register.

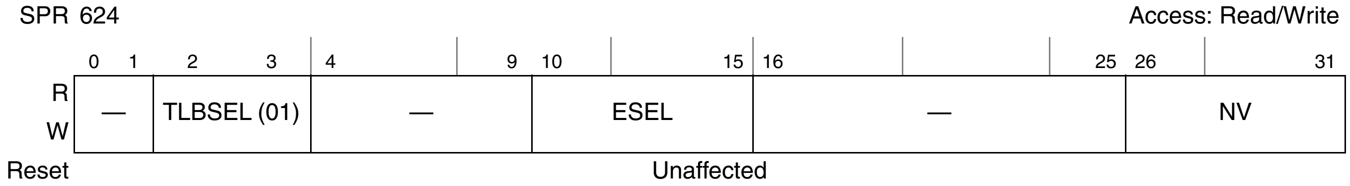


Figure 10-9. MMU Assist Register 0 (MAS0)

Table 10-9 describes the fields.

Table 10-9. MAS0 —MMU Read/Write and Replacement Control

Bit	Name	Comments, or Function when Set
0–1 [32–33]	—	Reserved
2–3 [34–35]	TLBSEL	selects TLB for access: 00=TLB0, 01=TLB1 (ignored by the e200, should be written to 01 for future compatibility)
4–9 [36–41]	—	Reserved
10–15 [42–47]	ESEL	Entry select for TLB.
16–25 [48–57]	—	Reserved
26–31 [58–63]	NV	Next replacement victim for TLB1 (software managed) Software updates this field; it is copied to the ESEL field on a TLB Error (see Table 10-15)

The MAS1 register is shown in Figure 10-10.

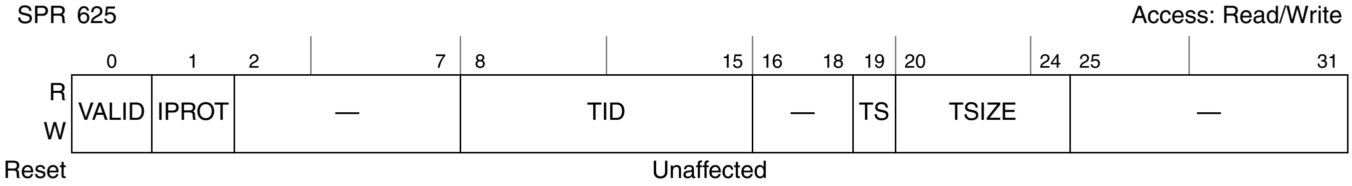


Figure 10-10. MMU Assist Register 1 (MAS1)

Table 10-10 describes the fields.

Table 10-10. MAS1—Descriptor Context and Configuration Control

Bit	Name	Comments, or Function when Set
0 [32]	VALID	TLB Entry Valid 0 This TLB entry is invalid 1 This TLB entry is valid
1 [33]	IPROT	Invalidation Protect 0 Entry is not protected from invalidation 1 Entry is protected from invalidation as described in Section 10.6.6, "IPROT Invalidation Protection." Protects TLB entry from invalidation by **tlbivax** (TLB1 only), or flash invalidates through MMUSCR0[TLB1_FI].
2–7 [34–39]	—	Reserved
8–15 [40–47]	TID	Translation ID bits This field is compared with the current process IDs of the effective address to be translated. A TID value of 0 defines an entry as global and matches with all process IDs.
16–18 [48–50]	—	Reserved
19 [51]	TS	Translation address 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.
20–24 [52–56]	TSIZE	Entry's page size Supported page sizes are: 0b00000–1 KB 0b00001–2 KB 0b00010–4 KB 0b00011–8 KB 0b00100–16 KB 0b00101–32 KB 0b00110–64 KB 0b00111–128 KB 0b01000–256 KB 0b01001–512 KB 0b01010–1 MB 0b01011–2 MB 0b01100–4 MB 0b01101–8 MB 0b01110–16 MB 0b01111–32 MB 0b10000–64 MB 0b10001–128 MB 0b10010–256 MB 0b10011–512 MB 0b10100–1 GB 0b10101–2 GB 0b10110–4 GB All other values are undefined
25–31 [57–63]	—	Reserved

Figure 10-11 shows the MAS2 register.

SPR 626

Access: Read/Write

	0–21	22–25	26	27	28	29	30	31
R W	EPN	—	VLE	W	I	M	G	E
Reset	Unaffected							

Figure 10-11. MMU Assist Register 2 (MAS2)

Table 10-11 describes the fields.

Table 10-11. MAS2—EPN and Page Attributes

Bit	Name	Comments, or Function when Set
0–21 [32–53]	EPN	Effective page number [0–21]
22–25 [54–57]	—	Reserved[1]
26 [58]	VLE	Power ISA VLE 0 This page is a standard Power ISA page 1 This page is a Power ISA VLE page This bit will always read as zero and writes will be ignored if **p_vle_present** is negated.
27 [59]	W	Write-through Required 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 to main memory
28 [60]	I	Cache Inhibited 0 This page is considered cacheable 1 This page is considered cache-inhibited
29 [61]	M	Memory Coherence Required 0 Memory Coherence is not required 1 Memory Coherence is required
30 [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 (i.e. they are known to be required) e200z7 uses the guarded attribute as described in Section 9.16, “Page Table Control Bits ,” for more information.
31 [63]	E	Endianness 0 The page is accessed in big-endian byte order. 1 The page is accessed in true little-endian byte order. Determines endianness for the corresponding page. Refer to Section 11.2.5, “Byte Lane Specification,” for more information

[1] These bits are not implemented, will be read as zero, and writes are ignored.

The MAS3 register is shown in Figure 10-12.

SPR 627 Access: Read/Write

	0–21	22	23	24	25	26	27	28	29	30	31
R W	RPN	U0	U1	U2	U3	UX	SX	UW	SW	UR	SR
Reset	Unaffected										

Figure 10-12. MMU Assist Register 3 (MAS3)

Table 10-12 describes the fields.

Table 10-12. MAS3—RPN and Access Control

Bit	Name	Comments, or Function when Set
0–21 [32–53]	RPN	Real page number [0–21] Only bits that correspond to a page number are valid. Bits that represent offsets within a page are ignored and should be zero.
22–25 [54–57]	U0-U3	User bits [0–3] for use by system software
26–31 [58–63]	PERMIS	Permission bits (UX, SX, UW, SW, UR, SR)

The MAS4 register is shown in Figure 10-13.

SPR 628 Access: Read/Write

	0–1	2–3	4–13	14–15	16–19	20–23	24–25	26	27	28	29	30	31
R W	—	TLBSELD (01)	—	TIDSELD	—	TSIZED	—	VLED	WD	ID	MD	GD	ED
Reset	Unaffected												

Figure 10-13. MMU Assist Register 4 (MAS4)

Table 10-13 describes the fields.

Table 10-13. MAS4—Hardware Replacement Assist Configuration Register

Bit	Name	Comments, or Function when Set
0–1 [32–33]	—	Reserved
2–3 [34–35]	TLBSELD	Default TLB selected 00 TLB0 01 TLB1
4–13 [36–45]	—	Reserved
14–15 [46–47]	TIDSELD	Default PID# to load TID from 00 PID0 01 Reserved, do not use 10 Reserved, do not use 11 TIDZ (0x00)) (Use all zeros, the globally shared value)

Table 10-13. MAS4—Hardware Replacement Assist Configuration Register (continued)

Bit	Name	Comments, or Function when Set
16–19 [48–51]	—	Reserved
20–24 [52–56]	TSIZED	Default TSIZE value
–25 [–57]	—	Reserved
26 [58]	VLED	Default VLE value
27–31 [59–63]	DWIMGE	Default WIMGE values

The MAS6 register is shown in Figure 10-14.

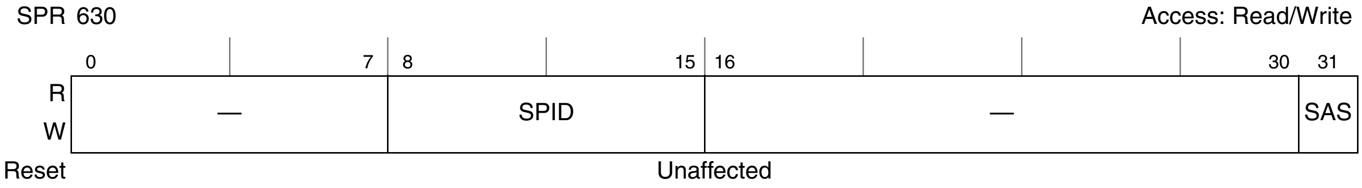


Figure 10-14. MMU Assist Register 6 (MAS6)

Table 10-14 describes the fields.

Table 10-14. MAS6—TLB Search Context Register 0

Bit	Name	Comments, or Function when Set
0–7 [32–39]	—	Reserved
8–15 [40–47]	SPID	PID value for searches
16–30 [48–62]	—	Reserved
31 [63]	SAS	AS value for searches

10.7.4 MAS Register Updates

Table 10-15 details the updates to each MAS register field for each update type.

Table 10-15. MMU Assist Register Field Updates

Bit/Field	MAS affected	Instr/Data TLB Error	tlbsx hit	tlbsx miss	tlbre	tlbwe	ISI/DSI
TLBSEL	0	TLBSELD	‘Hitting TLB’	TLBSELD	NC	NC	NC
ESEL	0	NV	matched entry	NV	NC	NC	NC
NV	0	NC	NC	NC	NC	NC	NC
VALID	1	1	1	0	V(array)	NC	NC
IPROT	1	0	Matched IPROT if TLB1 hit, else 0	0	IPROT(array) if TBL1, else 0	NC	NC
TID[0–7]	1	TIDSELD (pid0,TIDZ)	TID(array)	SPID	TID(array)	NC	NC
TS	1	MSR(IS/DS)	SAS	SAS	TS(array)	NC	NC
TSIZE[0–4]	1	TSIZED	TSIZE(array)	TSIZED	TSIZE(array)	NC	NC
EPN[0–21]	2	I/D EPN	EPN(array)	**tlbsx** EPN	EPN(Array)	NC	NC
VWIMGE	2	Default values	VWIMGE(array)	Default values	VWIMGE(array)	NC	NC
RPN[0–21]	3	Zeroed	RPN(Array)	Zeroed	RPN(Array)	NC	NC
ACCESS (PERMISS + U0:U3)	3	Zeroed	Access(Array)	Zeroed	Access(Array)	NC	NC
TLBSELD	4	NC	NC	NC	NC	NC	NC
TIDSELD[0–1]	4	NC	NC	NC	NC	NC	NC
TSIZED[0–4]	4	NC	NC	NC	NC	NC	NC
Default VWIMGE	4	NC	NC	NC	NC	NC	NC
SPID	6	PID0	NC	NC	NC	NC	NC
SAS	6	MSR(IS/DS)	NC	NC	NC	NC	NC

10.8 TLB Coherency Control

The e200 core allows invalidation of a TLB entry as described in the Power ISA embedded category architecture. The **tlbivax** instruction invalidates local TLB entries only. No broadcast is performed, as no hardware-based coherency support is provided.

The **tlbivax** instruction invalidates by effective address only. This means that only the TLB entry’s EPN bits are used to determine if the TLB entry should be invalidated. It is therefore possible for a single **tlbivax** instruction to invalidate multiple TLB entries, since the AS and TID fields of the entries are ignored.

10.9 Core Interface Operation for MMU Control Instructions

MMU control instructions utilize the normal CPU interface to perform MMU control instructions. The address bus is driven with the effective address value calculated by the instruction (if any). The access is treated as a Supervisor Data word-size write, and the Transfer Type encodings are used to distinguish these operations from other load and store operations. These transfers do not cause debug data address compare matches to occur regardless of the effective address that is driven.

10.9.1 Transfer Type Encodings for MMU Control Instructions

Transfer type encodings are used to indicate whether a normal access, atomic access, cache management control access, or MMU management control access is being requested. These attribute signals are driven with addresses when an access is requested. Table 10-16 shows the definitions of the **p_d_ttype[0:5]** encodings.

Table 10-16. Transfer Type Encoding

p_d_ttype[0:5][1]	Transfer Type	Instruction
00000e	Normal	Normal loads/stores
000010	Atomic	**lbarx, lharx, lwarx,** **stbcx., sthcx.,** and **stwcx.**
00010e	Flush Data Block	**dcbst**
00011e	Flush and Invalidate Data Block	**dcbf**
00100e	Allocate and Zero Data Block	**dcbz**
001010	Invalidate Data Block	**dcbi**
00110e	Invalidate Instruction Block	**icbi**
001110	multiple word load/store	**lmw, stmw**
010000	TLB Invalidate	**tlbivax**
010010	TLB Search	**tlbsx**
010100	TLB Read entry	**tlbre**
010110	TLB Write entry	**tlbwe**
011000	Touch for Instruction	**icbt**
011010	Lock Clear for Instruction	**icblc**
011100	Touch for Instruction and Lock Set	**icbtls**
011110	Lock Clear for Data	**dcblc**
10000e	Touch for Data	**dcbt**
10001e	Touch for Data Store	**dcbtst**
100100	Touch for Data and Lock Set	**dcbtls**
100110	Touch for Data Store and Lock Set	**dcbtstls**

[1] p_ttype[5] 'e' is set to set to 0.

10.10 Effect of Hardware Debug on MMU Operation

Hardware debug facilities utilize normal CPU instructions to access register and memory contents during a debug session. If desired during a debug session, the debug firmware may disable the translation process and may substitute default values for the access protection (UX, UR, UW, SX, SR, SW) bits, and values obtained from the OnCE control register for page attribute (VLE, W, I, M, G, E) bits normally provided by a matching TLB entry. In addition, no address translation is performed, and instead, a 1:1 mapping of effective to real addresses is performed. When disabled during the debug session, no TLB miss or TLB Access Protection related DSI conditions occur. If the debugger wants to use the normal translation process, the MMU can be left enabled in the OnCE OCR, and normal translation (including the possibility of a TLB Miss or DSI) remains in effect.

Refer to Section 13.4.6.3, “e200 OnCE Control Register (OCR),” for more detail on controlling MMU operation during debug sessions.

10.11 External Translation Alterations for Realtime Systems

To support realtime systems in which dynamic mapping of calibration or other data types is needed, the MMU provides special capabilities on a subset of TLB entries. These capabilities allow external hardware to dynamically select one of multiple mappings to one or more physical pages by the same logical address. This capability provides an inexpensive way of dynamically overlaying selected RAM pages on top of read-only memory during runtime. The particular physical page a given logical page maps to can be dynamically altered by means of the **p_extpid[6:7]** inputs. This capability is only provided for TLB1 entries 0–15, and only for a restricted subset of PID values.

The **p_extpid_en** control input controls the enabling of the dynamic mapping capability. This input is sampled with the rising edge of the clock, and when asserted, allows the use of the dynamic remapping capability.

When one or more of TLB1 entries 0–15 is programmed with a TID value of 0b1111xxxx, special entry-specific logic is enabled for the entry. This logic causes the sampled values of the **p_extpid[6:7]** inputs to be used in place of PID0[6–7] for the purposes of comparison of this entry with the current PID0 register contents to determine an entry hit condition.

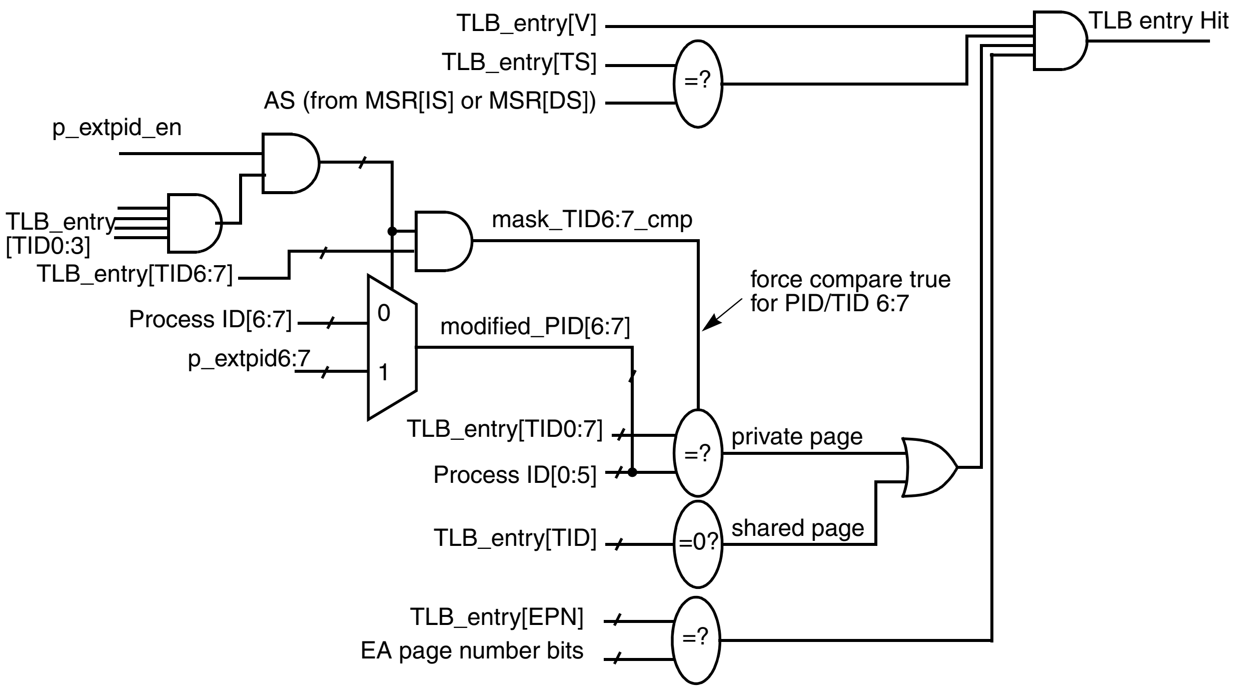
In addition, for those entries within entries 0–15 programmed with a TID value of 0b1111xx11, the comparison of TID[6–7] to PID0[6–7] for a match is always forced true. This means that the hit condition for these entries is independent of the sampled values of the **p_extpid[6:7]** inputs.

Entries within entries 0–15 programmed with a TID value of 0b1111nm00 match a PID0 value of 0b1111nmxx when **p_extpid[6:7]** inputs are 00. Those programmed with a TID value of 0b1111nm01 match a PID0 value of 0b1111nmxx when **p_extpid[6:7]** inputs are 01. Those programmed with a TID value of 0b1111nm10 match a PID0 value of 0b1111nmxx when **p_extpid[6:7]** inputs are 10. Those entries within entries 0–15 programmed with a TID value of 0b1111nm11 match a PID0 value of 0b1111nmxx regardless of the sampled values of the **p_extpid[6:7]** inputs.

This logic allows application software of this type to set up to three independent mappings for a set of calibration pages and for external hardware to select between one of the three based on the driven values of the **p_extpid[6:7]** inputs. The other pages are mapped with a common set of entries with stored TID values of 1111xx11, which match for all sets of calibration page selections. This specialized software must use PID values in the range of 111100xx to 111111xx.

Software is responsible for coordinating the modification to the **p_extpid[6:7]** inputs to ensure they only change when there is no possibility of an error induced by simultaneous use.

Figure 10-15 shows the equivalent logical operation of the capability.



Note: Functionality available for entry #0–15 only

Figure 10-15. External Translation Alteration TLB Entry Compare Process

Chapter 11
External Core Complex Interfaces

This chapter describes the external interfaces to the e200z7 core complex. This chapter also documents signal descriptions and data transfer protocols.

The external interfaces encompass control and data signals supporting instruction and data transfers as well as support for interrupts, including vectored interrupt logic, reset support, power management interface signals, debug event signals, time base control and status information, processor state information, Nexus 1/3/OnCE/JTAG interface signals, and a test interface.

The memory portion of the e200 core interface consists of a pair of 64-bit wide standard AHB system buses, one for instructions and the other for data. The data memory interface supports read and write transfers of 8, 16, 24, 32, and 64 bits, supports misaligned transfers, supports true big- and little-endian operating modes, and operates in a pipelined fashion. The instruction memory interface supports read transfers of 16, 32, and 64 bits, supports misaligned transfers, supports true big- and little-endian operating modes, and operates in a pipelined fashion.

The memory interface supported by the BIUs is based on the AHB 2.v6 definition. Additional sideband signals have been added to support additional control functions.

NOTE

The AHB bit and byte ordering reflect a natural little-endian ordering, as used by the AMBA documentation. The e200z7 BIU automatically performs the necessary byte lane conversions to support big-endian transfers. Memories and peripheral devices/interfaces should be wired according to byte lane addresses defined in Section 11.2.5, “Byte Lane Specification,” and Table 11-10.

Single-beat and misaligned transfers are supported for cache-inhibited read and write cycles and write-buffer writes. Burst transfers (double-word aligned) of four double words are supported for cache linefill and copyback operations.

Misaligned accesses are supported with one or more transfers to an interface. If an access is misaligned, but is contained within an aligned 64-bit double word, the core performs a single transfer, and the memory interface is responsible for delivering (reads) or accepting (writes) the data corresponding to the size and byte enable signals aligned according to the low order three address bits. If an access is misaligned and crosses a 64-bit boundary, the BIU performs a pair of transfers beginning at the effective address for the first transfer, along with appropriate byte enables, and for the second transfer the address is incremented to the next 64-bit boundary, and the size and byte enable signals are driven to correspond to the number of remaining bytes to be transferred.

11.1 Signal Index

This section contains an index of the e200 signals. The following prefixes are used for the e200 signal mnemonics:

- **m** denotes master clock and reset signals
- **p** denotes processor or core-related signals
- **j** denotes JTAG mode signals
- **jd** denotes JTAG and Debug mode signals
- **ipt** denotes Scan and Test Mode signals
- **nex** denotes Nexus signals

NOTE

The “_b” suffix denotes an active low signal. Signals without the active-low suffix are active high.

Figure 11-1 and Figure 11-2 group core bus and control signals by function.

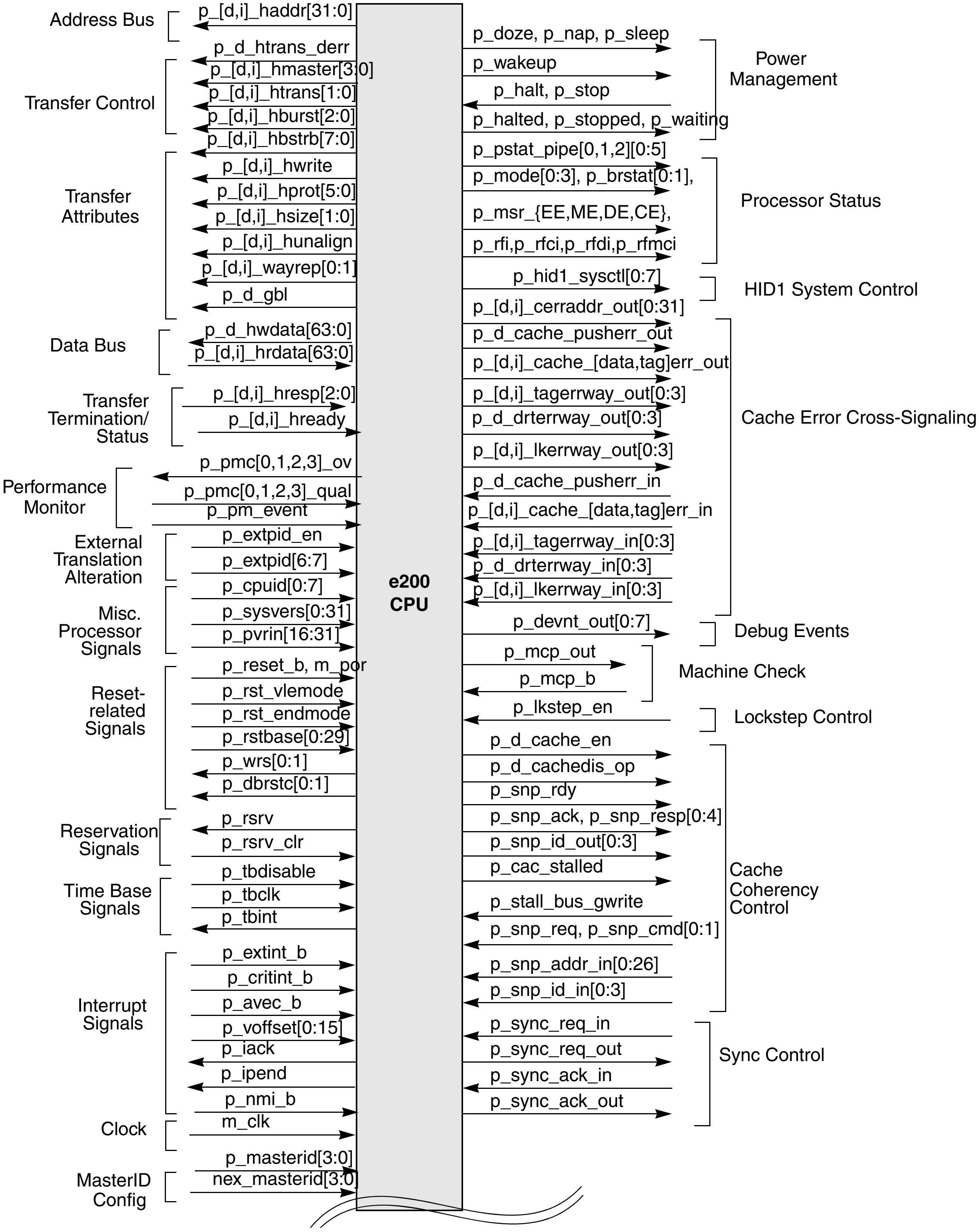
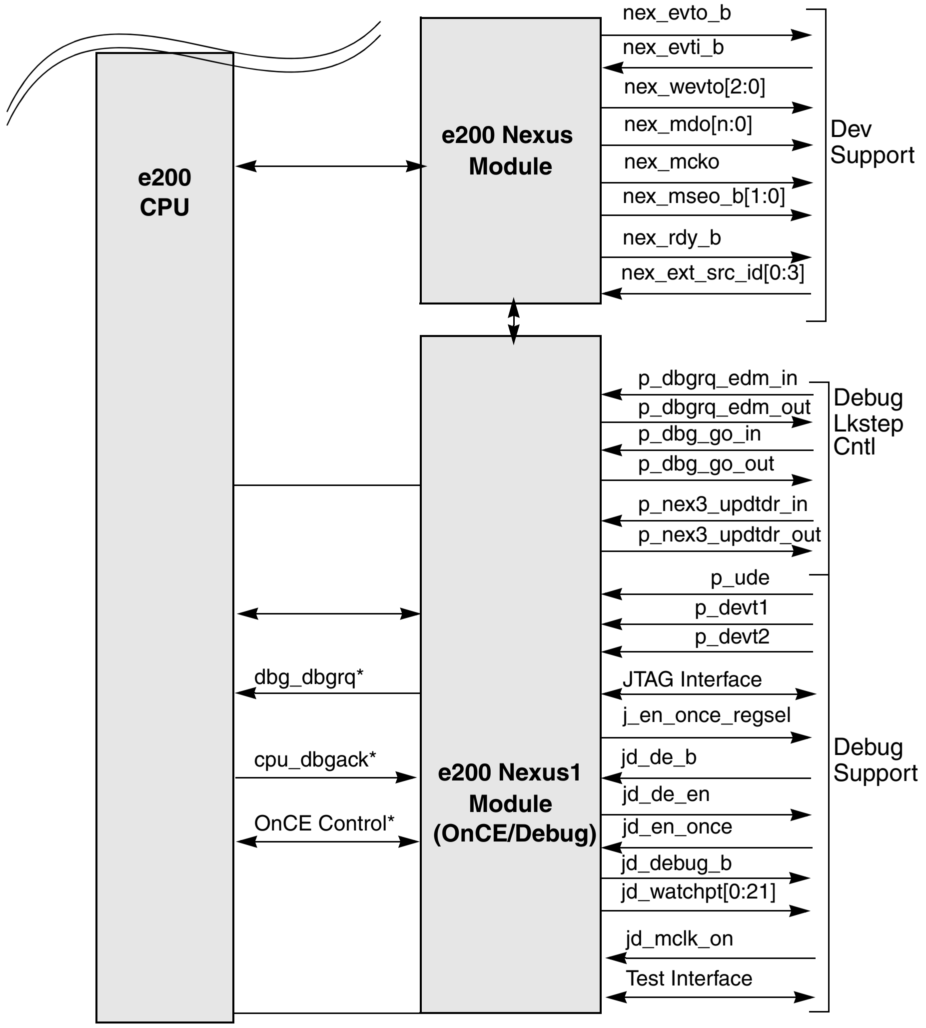


Figure 11-1. e200 Signal Groups—part 1



Note:
* = internal core signal

Figure 11-2. e200 Signal Group—part 2

Table 11-1 shows e200 signal function and type, signal definition, and reset value. Signals are presented in functional groups.

Table 11-1. Interface Signal Definitions

Signal Name	Type	Reset Value	Definition
Clock and Reset-related Signals			
m_clk	I	—	Global system clock
m_por	I	—	Power-on reset
p_reset_b	I	—	Processor reset input
p_wrs[0:1]	O	—	Processor watchdog reset status outputs

Table 11-1. Interface Signal Definitions (continued)

Signal Name	Type	Reset Value	Definition
p_dbrstc[0:1]	O	—	Processor debug reset control outputs
p_rstbase[0:29]	I	—	Reset exception handler base address
p_rst_endmode	I	—	Reset endian mode select
p_rst_vlemode	I	—	Reset VLE mode select, value to be loaded into TLB entry 0 on reset.
Memory Interface Signals			
p_d_hmaster[3:0], p_i_hmaster[3:0]	O	—	Master ID
p_d_haddr[31:0], p_i_haddr[31:0]	O	—	Address buses
p_d_hwrite, p_i_hwrite*	O	0	Write signal (always driven low for p_i_hwrite)
p_d_hprot[5:0], p_i_hprot[5:0]	O	—	Protection Codes
p_d_htrans[1:0], p_i_htrans[1:0]	O	—	Transfer Type
p_d_htrans_derr	O	—	Transfer Data Parity error indicator (push errors)
p_d_hburst[2:0], p_i_hburst[2:0]	O	—	Burst Type
p_d_hsize[1:0], p_d_hsize[1:0]	O	—	Transfer Size
p_d_hunalign, p_i_hunalign	O	—	Indicates the current data access is a misaligned access.
p_d_gbl	O	—	Indicates the current access is marked as a globally coherent access.
p_d_hbstrb[7:0], p_i_hbstrb[7:0]	O	0	Byte strobes
p_d_hrdata[63:0], p_i_hrdata[63:0]	I	—	Read data buses
p_d_hwdata[63:0]	O	—	Write data bus
p_d_hready, p_i_hready	I	—	Transfer Ready
p_d_hresp[2:0], p_i_hresp[1:0]	I	—	Transfer Response
p_d_wayrep[0:1] p_i_wayrep[0:1]	0	—	Way replacement Indicates the cache way being replaced by a burst read linefill.
p_d_ahb_clken, p_i_ahb_clken	I	—	AHB Clock enable
Master ID Configuration Signals			
p_masterid[3:0]	I	—	CPU Master ID configuration
nex_masterid[3:0]	I	—	Nexus Master ID configuration
Sync Control Interface Signals			
p_sync_req_in	I	—	Sync Request Input
p_sync_ack_in	I	—	Sync Acknowledge Input
p_sync_req_out	O	0	Sync Request Output

Table 11-1. Interface Signal Definitions (continued)

Signal Name	Type	Reset Value	Definition
p_sync_ack_out	O	0	Sync Acknowledge Output
Coherency Control Interface Signals			
p_snp_req	I	—	Snoop Request
p_snp_cmd[0:1]	I	—	Snoop Command
p_snp_addr_in[0:31]	I	—	Snoop Address Input (bit 0 is MSB)
p_snp_id_in[0:3]	I	—	Snoop ID Input
p_stall_bus_gwrite	I	—	Stall External Bus Global Writes
p_snp_rdy	O	0	Snoop Ready
p_snp_ack	O	0	Snoop Acknowledge
p_snp_resp[0:4]	O	0	Snoop Response
p_snp_id_out[0:3]	O	—	Snoop ID Output
p_cac_stalled	O	0	CPU cache access Stalled
p_d_cache_en	O	0	Data cache enabled/disabled state
p_d_cachedis_op	O	0	Data cache disable operation in progress
Interrupt Interface Signals			
p_extint_b	I	—	External Input interrupt request
p_critint_b	I	—	Critical Input interrupt request
p_nmi_b	I	—	Non-Maskable Interrupt input request
p_avec_b	I	—	Autovector request Use internal interrupt vector offset
p_voffset[0:15]	I	—	Interrupt vector offset for vectored interrupts
p_iack	O	0	Interrupt Acknowledge. Indicates an interrupt is being acknowledge.
p_ipend	O	0	Interrupt Pending. Indicates an interrupt is pending internally.
p_mcp_b	I	—	Machine Check input request
CPU Lockstep Enable Signal			
p_lkstep_en	I	—	CPU Lockstep Enable input
Cache Error Cross-Signaling Signals			
p_[d,i]_cache_tagerr_in	I	—	Cache tag error input
p_[d,i]_cache_dataerr_in	I	—	Cache data error input
p_d_pusherr_in	I	—	Cache data push error input

Table 11-1. Interface Signal Definitions (continued)

Signal Name	Type	Reset Value	Definition
p_[d,i]_tagerrway_in[0:3]	I	—	Cache tag error ways input
p_d_drterrway_in[0:3]	I	—	Cache dirty error ways input
p_[d,i]_lkerrway_in[0:3]	I	—	Cache lock error ways input
p_[d,i]_cerraddr_out[0:31]	O	—	Cache error address output
p_[d,i]_cache_tagerr_out	O	0	Cache tag error output
p_[d,i]_cache_dataerr_out	O	0	Cache data error output
p_d_pusherr_out	O	0	Cache data push error output
p_[d,i]_tagerrway_out[0:3]	O	—	Cache error ways output
p_d_drterrway_out[0:3]	O	—	Cache dirty error ways output
p_[d,i]_lkerrway_out[0:3]	O	—	Cache error ways output
External Translation Alteration Signals			
p_extpid_en	I	—	External PID enable input
p_extpid[6:7]	I	—	External PID[6:7] input
Time Base Signals			
p_tbint	O	0	Time Base Interrupt
p_tbdisable	I	—	Time Base Disable input
p_tbclk	I	—	Time Base Clock input
Misc. CPU Signals			
p_cpuid[0:7]	I	—	CPU ID input
p_sysvers[0:31]	I	—	System Version inputs (for SVR)
p_pvrin[16:31]	I	—	Inputs for PVR
p_pid0[0:7]	O	0	PID0[24:31] outputs
p_pid0_updt	O	0	PID0 update status
p_hid1_sysctl[0:7]	O	0	HID1[16:23] outputs
CPU Reservation Signals			
p_rsrv	O	0	Reservation status
p_rsrv_clr	I	—	Clear Reservation flag
CPU State Signals			
p_mode[0:3]	O	0	Indicates processor global status
p_pstat_pipe0[0:5], p_pstat_pipe1[0:5]	O	0	Indicates processor status for each pipe
p_brstat[0:1]	O	0	Indicates Branch prediction status

Table 11-1. Interface Signal Definitions (continued)

Signal Name	Type	Reset Value	Definition
p_msr_EE, p_msr_DE, p_msr_CE, p_msr_ME	O	0	Reflect the values of these MSR bits
p_rfi, p_rfci, p_rfdi, p_rfmci	O	0	Reflect the execution of the corresponding instruction
p_mcp_out	O	0	Indicates a machine check has occurred
p_doze	O	0	Indicates low-power doze mode of operation
p_nap	O	0	Indicates low-power nap mode of operation
p_sleep	O	0	Indicates low-power sleep mode of operation
p_wakeup	O	0	Indicates to external clock control module to enable clocks and exit from low-power mode
p_halt	I	—	CPU halt request
p_halted	O	0	CPU halted
p_stop	I	—	CPU stop request
p_stopped	O	0	CPU stopped
p_waiting	O	0	CPU waiting
CPU Performance Monitor Signals			
p_pm_event	I	—	Performance Monitor Event input
p_pmc0_ov	O	0	Performance Monitor Counter 0 OV bit
p_pmc1_ov	O	0	Performance Monitor Counter 1 OV bit
p_pmc2_ov	O	0	Performance Monitor Counter 2 OV bit
p_pmc3_ov	O	0	Performance Monitor Counter 3 OV bit
p_pmc0_qual	I	—	Performance Monitor Counter 0 trigger qualifier input
p_pmc1_qual	I	—	Performance Monitor Counter 1 trigger qualifier input
p_pmc2_qual	I	—	Performance Monitor Counter 2 trigger qualifier input
p_pmc3_qual	I	—	Performance Monitor Counter 3 trigger qualifier input
CPU Debug Event Signals			
p_ude	I	—	Unconditional Debug Event
p_devt1	I	—	Debug Event 1 input
p_devt2	I	—	Debug Event 2 input
p_devnt_out[0:7]	O	0	Debug Event outputs
Debug/Emulation Support Signals (Nexus 1/OnCE)			
jd_en_once	I	—	Enable full OnCE operation
jd_debug_b	O	1	Indicates processor has entered debug session

Table 11-1. Interface Signal Definitions (continued)

Signal Name	Type	Reset Value	Definition
jd_de_b	I	—	Debug request
jd_de_en	O	0	Active -high output enable for DE_b open-drain IO cell
jd_mclk_on	I	—	Indicates the system clock controller is actively toggling **m_clk**
jd_watchpt[0:29]	O	0	Indicate a watchpoint has occurred
Debug Lockstep Cross-Signaling Signals			
p_dbgrq_edm_in	I	—	Debug EDM debug request input
p_dbg_go_in	I	—	Debug OCMD go input
p_nex3_updtdr_in	I	—	Debug Nexus 3 synchronized update DR state in
p_dbgrq_edm_out	O	—	Debug EDM debug request output
p_dbg_go_out	O	—	Debug OCMD go output
p_nex3_updtdr_in	O	—	Debug Nexus 3 synchronized update DR state out
Development Support Signals (Nexus 3)			
nex_mcko	O	—	Nexus 3 Clock Output
nex_rdy_b	O	—	Nexus 3 Ready Output
nex_evto_b	O	—	Nexus 3 Event-Out Output
nex_wevto[3:0]	O	—	Nexus 3 Watchpoint Event-Out Output
nex_evti_b	I	—	Nexus 3 Event-In Input
nex_mdo[n:0]	O	—	Nexus 3 Message Data Output
nex_mseo_b[1:0]	O	—	Nexus 3 Message Start/End Output
JTAG-Related Signals			
j_trst_b	I	—	JTAG test reset from pad
j_tclk	I	—	JTAG test clock from pad
j_tms	I	—	JTAG test mode select from pad
j_tdi	I	—	JTAG test data input from pad
j_tdo	O	0	JTAG test data out to master controller or pad
j_tdo_en	O	0	Enables TDO output buffer
j_tst_log_rst	O	0	Indicates Test-Logic-Reset state of JTAG controller
j_capture_ir	O	0	Indicates Capture_IR state of JTAG controller
j_update_ir	O	0	Indicates Update_IR state of JTAG controller
j_shift_ir	O	0	Indicates Shift_IR state of JTAG controller

Table 11-1. Interface Signal Definitions (continued)

Signal Name	Type	Reset Value	Definition
j_capture_dr	O	0	Indicates parallel test data register load state of JTAG controller
j_shift_dr	O	0	Indicates the TAP controller is in shift DR state
j_update_gp_reg	O	0	Updates JTAG controller test data register
j_rti	O	0	JTAG controller run-test-idle state
j_key_in	I	—	Input for providing data to be shifted out during Shift_IR state when jd_en_once is negated
j_en_once_regsel	O	0	External Enable Once register select
j_nexus_regsel	O	0	External Nexus register select
j_lsrl_regsel	O	0	External LSRL register select
j_gp_regsel[0:9]	O	0	General-purpose external JTAG register select
j_id_sequence[0:1]	I	—	JTAG ID Register (2 MSBs of sequence field)
j_id_version[0:3]	I	—	JTAG ID Register Version Field
j_serial_data	I	—	Serial data from external JTAG registers
Test Primary Input/Output Signals			
Test Control Interface	—	—	Test Mode determination
Scan Test Interface	—	—	Scan Configuration and Testing
Memory BIST Interface	—	—	Memory BIST Configuration and Testing

11.2 Signal Descriptions

The following sections provide descriptions of the signals.

11.2.1 e200 Processor Clock (m_clk)

The **m_clk** input is the synchronous clock source for the e200 processor core. Because the e200 is designed for static operation, **m_clk** can be gated off to lower power dissipation, such as during low-power stopped states.

11.2.2 Reset-related Signals

The e200 supports several reset input signals for the CPU and JTAG/OnCE control logic: **m_por**, **p_reset_b**, and **j_trst_b**. The reset domains have been partitioned such that the CPU **p_reset_b** signal does not affect JTAG/OnCE logic and **j_trst_b** does not affect processor logic. It is possible and desirable to access OnCE registers while the processor is running or in reset. Alternatively, it is also possible and desirable to assert **j_trst_b** and clear the JTAG/OnCE logic without affecting the state of the processor.

The synchronization logic between the processor and debug module requires an assertion of either **j_trst_b** or **m_por** during initial processor power-up reset in order to ensure proper operation. If the pin associated with the **j_trst_b** input is designed with a pull-up resistor and left floating, then assertion of **m_por** is required during the initial power-on processor reset. Similarly, for those systems which do not have a power-on reset circuit and choose to tie **m_por** low, it is required to assert **j_trst_b** during processor power-up reset. Once a power-up reset has been achieved, the two resets can be asserted independently.

The watchdog reset status output signals **p_wrs[0:1]** are also provided which can be conditionally asserted by watchdog time-outs, and the debug reset control outputs **p_dbrstc[0:1]** can be asserted by debug control settings in DBCR0.

A set of input signals (**p_rstbase[0:29], p_rst_endmode, p_rst_vlemode**) are provided to relocate the reset exception handler to allow for flexible placement of boot code, and to select the default endian mode and VLE mode of the CPU out of reset.

These signals are described in detail in the following subsections.

11.2.2.1 Power-on Reset (m_por)

The **m_por** signal is the power-on reset input for the e200 processor. This signal serves the following purposes:

- **m_por** is "ORed" with the **j_trst_b** function and the resulting signal clears the JTAG TAP controller and associated registers as well as the OnCE state machine. This is an asynchronous clear with a short assertion time requirement.
- **m_por** is "ORed" with the **p_reset_b** function and the resulting signal clears certain CPU registers. This is an asynchronous clear with a short assertion time requirement.

11.2.2.2 Reset (p_reset_b)

The **p_reset_b** input is the active-low reset input for the e200 processor. **p_reset_b** is treated as an asynchronous input and is sampled by the clock control logic in the e200 debug module.

11.2.2.3 Watchdog Reset Status (p_wrs[0:1])

The **p_wrs[0:1]** outputs are active-high reset output status signals from the e200 core that reflect the value of the TSR[WRS] status field. **p_wrs[0:1]** are conditionally asserted by the watchdog timer (see Section 2.4.8, "Timer Control Register (TCR)," and Section 2.4.9, "Timer Status Register (TSR)").

11.2.2.4 Debug Reset Control (p_dbrstc[0:1])

The **p_dbrstc[0:1]** outputs are active-high reset output control signals from the e200 core that reflect the value of the DBCR0[RST] status field. **p_dbrstc[0:1]** are conditionally asserted by the debug control logic (Section 13.3.3.1, "Debug Control Register 0 (DBCR0)").

11.2.2.5 Reset Base (p_rstbase[0:29])

The **p_rstbase[0:29]** inputs are provided to allow system integrators to be able to specify/relocate the base address of the reset exception handler. These inputs are used to form the upper 30 bits of the instruction access following negation of reset which is used to fetch the initial instruction of the reset exception handler. These bits should be driven to a value corresponding to the desired boot memory device in the system. These inputs must remain stable in a window beginning two clocks prior to the negation of reset and extending into the cycle in which the reset vector fetch is initiated. These inputs are also used by the MMU during reset to form a default TLB entry 0 for translation of the reset vector fetch.

The initial instruction fetch will occur to the location [**p_rstbase[0:29]**] || 0b00.

11.2.2.6 Reset Endian Mode (p_rst_endmode)

The **p_rst_endmode** input is used by the MMU during reset to form the E bit of the default TLB entry 0 for translation of the reset vector fetch. A low logic level on this signal clears the resultant entry E bit, indicating a big-endian page. A high logic level on this signal sets the resultant entry E bit, indicating a little-endian page.

11.2.2.7 Reset VLE Mode (p_rst_vlemode)

The **p_rst_vlemode** input is used by the MMU during reset to form the VLE bit of the default TLB entry 0 for translation of the reset vector fetch. A low logic level on this signal clears the resultant entry VLE bit, indicating a standard Power ISA page. A high logic level on this signal sets the resultant entry VLE bit, indicating a VLE page.

11.2.2.8 JTAG/OnCE Reset (j_trst_b)

The **j_trst_b** signal (referred to in the IEEE Std 1149.1™ JTAG as the $\overline{\text{TRST}}$ signal) is an asynchronous reset with a short assertion time requirement. It is ORed with the **m_por** function and the resulting signal clears the OnCE TAP controller and associated registers as well as the OnCE state machine.

11.2.3 Address and Data Buses

Dual instruction and data interfaces are provided by the e200z7. They are described together, with appropriate differences denoted.

11.2.3.1 Address Bus (p_d_haddr[31:0], p_i_haddr[31:0])

These outputs provide the address for a bus transfer. Per the AHB definition, **p_[d,i]_haddr[31]** is the MSB and **p_[d,i]_haddr[0]** is the LSB.

11.2.3.2 Read Data Bus (p_d_hrdata[63:0], p_i_hrdata[63:0])

These inputs provide data to the e200z7 on read transfers. The read data bus can transfer 8, 16, 24, 32, or 64 bits of data per bus transfer. Instruction transfers do not use the 8-bit and 24-bit capability. Per the AHB definition, **p_[d,i]_hrdata[63]** is the MSB and **p_hrdata[0]** is the LSB. Table 11-2 shows the relationship of byte addresses to read data bus signals.

Table 11-2. p_hrdata[63:0] Byte Address Mappings

Memory Byte Address	Wired to p_[d,i]_hrdata Bits
000	7:0
001	15:8
010	23:16
011	31:24
100	39:32
101	47:40
110	55:48
111	63:56

11.2.3.3 Write Data Bus (p_d_hwdata[63:0])

These outputs transfer data from the e200z7 on write transfers. The write data bus can transfer 8, 16, 24, 32, or 64 bits of data per bus transfer. Per the AHB definition, **p_d_hwdata[63]** is the MSB and **p_d_hwdata[0]** is the LSB. Figure 11-3 shows the relationship of byte addresses to write data bus signals.

Table 11-3. p_d_hwdata[63:0] Byte Address Mappings

Memory Byte Address	Wired to p_d_hwdata Bits
000	7:0
001	15:8
010	23:16
011	31:24
100	39:32
101	47:40
110	55:48
111	63:56

11.2.4 Transfer Attribute Signals

The following paragraphs describe the transfer attribute signals, which provide additional information about the bus transfer cycle. Transfer attributes are driven with address at the beginning of a bus transfer.

11.2.4.1 Transfer Type (p_d_htrans[1:0], p_i_htrans[1:0])

The processor drives these signals to indicate the current transfer type. Table 11-4 shows **p_[d,i]_htrans[1:0]** encoding.

Table 11-4. p_[d,i]_htrans[1:0] Transfer Type Encoding

p_[d,i]_htrans[1]	p_[d,i]_htrans[0]	Access type
0	0	IDLE—no data transfer is required
0	1	BUSY—Master is busy, burst transfer continues. (encoding not used by e200z7)
1	0	NONSEQ—indicates the first transfer of a burst, or a single transfer. Address and control signals are unrelated to the previous transfer
1	1	SEQ—indicates the continuation of a burst. Address and control signals are related to the previous transfer. Control signals are the same, Address has been incremented by the size of the data transferred (optionally wrapped)

If the **p_[d,i]_htrans[1:0]** encoding is not IDLE or BUSY, a transfer is being requested. The e200z7 does not utilize the BUSY encoding and does not present this type of transfer to a bus slave. Slaves must terminate IDLE transfers with a zero wait-state OKAY response and ignore the (non-existent) transfer.

11.2.4.2 Write (p_d_hwrite, p_i_hwrite)

This output signal defines the data transfer direction for the current bus cycle. A high (logic one) level indicates a write cycle, and a low (logic zero) level indicates a read cycle. For **p_i_hwrite**, the signal is internally driven low for all instruction AHB transfers.

11.2.4.3 Transfer Size (p_d_hsize[1:0], p_i_hsize[1:0])

The **p_[d,i]_hsize[1:0]** signals indicate the data size for a bus transfer. Table 11-5 shows the definitions of the **p_[d,i]_hsize[1:0]** encodings. For misaligned transfers, the transfer size may indicate a size larger than the requested size to ensure that all asserted byte strobes are contained within the “container” defined by **p_[d,i]_hsize[1:0]**. Refer to Table 11-11 and Table 11-12 for **p_[d,i]_hsize[1:0]** encodings used for aligned and misaligned transfers.

Table 11-5. p_[d,i]_hsize[1:0] Transfer Size Encoding

p_[d,i]_hsize[1:0]	Transfer Size
00	Byte
01	Half word (2 bytes)
10	Word (4 bytes)
11	Double Word (8 bytes)

11.2.4.4 Burst Type (p_d_hburst[2:0], p_i_hburst[2:0])

The **p_[d,i]_hburst[2:0]** signals indicate the burst type for a bus transfer. Table 11-6 shows the definitions of the **p_[d,i]_hburst[2:0]** encodings.

Table 11-6. p_[d,i]_hburst[2:0] Burst Type Encoding

p_hburst[2:0]	Burst Type
000	SINGLE—No burst, single beat only
001	INCR—Incrementing burst of unspecified length—Unused
010	WRAP4—4-beat wrapping burst
011	INCR4—4-beat incrementing burst—Unused
100	WRAP8—8-beat wrapping burst—Unused
101	INCR8—8-beat incrementing burst—Unused
110	WRAP16—16-beat wrapping burst—Unused
111	INCR16—16-beat incrementing burst—Unused

The e200z7 will only utilize SINGLE and WRAP4 burst types. In addition, all WRAP4 bursts are of double word size aligned to double-word boundaries.

11.2.4.5 Protection Control (p_d_hprot[5:0], p_i_hprot[5:0])

The e200z7 drives the **p_[d,i]_hprot[5:0]** signals to indicate the type of access for the current bus cycle. **p_[d,i]_hprot[0]** indicates instruction/data, **p_[d,i]_hprot[1]** indicates user/supervisor. **p_[d,i]_hprot[5]** indicates whether the access is exclusive (i.e. for a **lbarx**, **lharx**, **lwarx**, **stbcx.**, **sthcx.**, or **stwcx.** instruction). **p_[d,i]_hprot[4:2]** (allocate, cacheable, bufferable) are used to indicate particular cache attributes for the access and are driven to default values based on settings in the memory management unit.

Table 11-7 shows the definitions of the **p_d_hprot[5:0]** signals.

Table 11-7. p_d_hprot[5:0] Protection Control Encoding

p_hprot[5]	p_hprot[4]	p_hprot[3]	p_hprot[2]	p_hprot[1]	p_hprot[0]	Transfer Type
—	—	—	—	0	1	User mode access
—	—	—	—	1	1	Supervisor mode access
—	0	0	0	—	1	Cache-Inhibited
—	0	0	1	—	1	Guarded, not Cache-Inhibited
—	0	1	0	—	1	Reserved
—	0	1	1	—	1	Reserved
—	1	0	0	—	1	Reserved
—	1	0	1	—	1	Reserved
—	1	1	0	—	1	Cacheable, Write through

Table 11-7. p_d_hprot[5:0] Protection Control Encoding

p_hprot[5]	p_hprot[4]	p_hprot[3]	p_hprot[2]	p_hprot[1]	p_hprot[0]	Transfer Type
—	1	1	1	—	1	Cacheable, Writeback
0	—	—	—	—	1	Not Exclusive
1	—	—	—	—	1	Exclusive Access

Table 11-8 shows the definitions of the **p_i_hprot[5:0]** signals.

Table 11-8. p_i_hprot[5:0] Protection Control Encoding

p_hprot[5]	p_hprot[4]	p_hprot[3]	p_hprot[2]	p_hprot[1]	p_hprot[0]	Transfer Type
0	-	—	—	0	0	User mode access
0	-	—	—	1	0	Supervisor mode access
0	0	0	0	—	0	Cache-Inhibited
0	0	0	1	—	0	Reserved
0	0	1	0	—	0	Reserved
0	0	1	1	—	0	Reserved
0	1	0	0	—	0	Reserved
0	1	0	1	—	0	Reserved
0	1	1	0	—	0	Cacheable
0	1	1	1	—	0	Reserved

Note that all signals are provided on both I and D ports, although they will not all change state. (ex. p_d_hprot0 is always high, etc.).

The e200z7 maps the Power ISA embedded category storage attributes to the AHB data port **hprot** signals in the manner described in Table 11-9.

Table 11-9. Mapping of Access attributes to p_d_hprot[4:2] Protection Control

[I]	[G]	[W]	p_hprot[4]	p_hprot[3]	p_hprot[2]	Transfer Type
0	0	0	1	1	1	Cacheable, write back
0	0	1	1	1	0	Cacheable, write through
0	1	—	0	0	1	Guarded, not cache-Inhibited
1	—	—	0	0	0	Cache-Inhibited
—	—	—	0	0	1	Buffered Store, page marked guarded
—	—	—	1	1	0	Buffered Store and page marked write through and non-guarded
—	—	—	1	1	1	Buffered Store and page marked copyback and non-guarded

For buffered stores, **p_d_hprot[1]** is driven with the user/supervisor mode attribute associated with the store at the time it was buffered.

11.2.4.6 Data Transfer Error (p_d_htrans_derr)

The **p_d_htrans_derr** control signal is driven during bus transfers on the data interface to indicate that a data cache data array parity error has occurred for a cache push (copyback) operation and that the data corresponding to this address is not valid due to a parity error. This signal is driven valid with address and attribute timing. System logic may monitor this output and perform any desired recovery activity. This signal is only asserted during a copyback operation for those beats for which the corresponding data has a parity error.

11.2.4.7 Globally Coherent Access—(p_d_gbl)

The **p_d_gbl** control signal is driven during bus transfers on the data interface to indicate whether the memory access is marked by the MMU 'M' page attribute as globally coherent. This signal is driven valid with address and attribute timing and remains valid for all beats of a burst access. This signal reflects the value of the M (memory coherence required) attribute for the page associated with the access, except for dirty line pushes to memory. For those accesses, it is negated.

11.2.4.8 Cache Way Replacement (p_d_wayrep[0:1], p_i_wayrep[0:1])

The **p_[d,i]_wayrep[0:1]** control signals are driven valid during cache linefills to indicate which way of the cache is being replaced. These signals are driven valid with address and attribute timing, and remain valid for all beats of the burst read. These signals are undefined on all other transfer types.

11.2.5 Byte Lane Specification

Read transactions transfer from 1 to 8 bytes of data on the **p_[d,i]_hrdata[63:0]** bus. The byte lanes involved in the transfer are determined by the starting byte number specified by the lower address bits in conjunction with the transfer size and byte strobes. Addressing of the byte lanes is shown big-endian (left to right) regardless of the endian mode of the e200 core. The byte of memory corresponding to address 0 is connected to B0 (**p_[d,i]_h{r,w}data[7:0]**) and the byte of memory corresponding to address 7 is connected to B7 (**p_[d,i]_h{r,w}data[63:56]**). The CPU internally permutes read data as required for the endian mode of the current access. Misaligned transfers are indicated with the **p_[d,i]_hunalign** signal to indicate that byte strobes do not correspond exactly to size and low-order address bits.

11.2.5.1 Unaligned Access (p_d_hunalign, p_i_hunalign)

The **p_[d,i]_hunalign** output signal indicates that the current access is a misaligned access. This signal is asserted for misaligned data accesses and for misaligned instruction accesses from VLE pages. Normal Power ISA instruction pages are always aligned. The timing of this signal is approximately the same as address timing. When **p_[d,i]_hunalign** is asserted, the **p_[d,i]_hbstrb[7:0]** byte strobe signals indicate the selected bytes involved in the current portion of the misaligned access, which may not include all bytes defined by the size and low-order address signals. Aligned transfers also assert the byte strobes, but in a manner corresponding to the size and low order address bits.

11.2.5.2 Byte Strobes (p_d_hbstrb[7:0], p_i_hbstrb[7:0])

The **p_[d,i]_hbstrb[7:0]** byte strobe signals indicate the selected bytes involved in the current transfer. For a misaligned access, the current transfer may not include all bytes defined by the size and low-order address signals. For aligned transfers, the byte strobe signals will correspond to the bytes defined by the size and low-order address signals.

Table 11-10 shows the relationship of byte addresses to the byte strobe signals.

Table 11-10. p_[d,i]_hbstrb[7:0] to Byte Address Mappings

Memory Byte Address	Wired to p_h{r,w}data Bits	Corresponding Byte Strobe Signal
000	7:0	p_[d,i]_hbstrb[0]
001	15:8	p_[d,i]_hbstrb[1]
010	23:16	p_[d,i]_hbstrb[2]
011	31:24	p_[d,i]_hbstrb[3]
100	39:32	p_[d,i]_hbstrb[4]
101	47:40	p_[d,i]_hbstrb[5]
110	55:48	p_[d,i]_hbstrb[6]
111	63:56	p_[d,i]_hbstrb[7]

Table 11-11 lists all of the data transfer permutations. Note that misaligned data requests that cross a 64-bit boundary are broken up into two separate bus transactions, and the address value and the size encoding for the first transfer is not modified. The table is arranged in a big-endian fashion, but the active lanes are the same regardless of the endian-mode of the access. The e200z7 performs the proper byte routing internally based on endianness.

Table 11-11. Byte Strobe Assertion for Transfers

Program Size and byte offset	A(2:0)	HSIZE [1:0]	Data Bus Byte strobes								HUNALIGN
			B0	B1	B2	B3	B4	B5	B6	B7	
Byte = 000	0 0 0	0 0	X	—	—	—	—	—	—	—	0
Byte = 001	0 0 1	0 0	—	X	—	—	—	—	—	—	0
Byte = 010	0 1 0	0 0	—	—	X	—	—	—	—	—	0
Byte = 011	0 1 1	0 0	—	—	—	X	—	—	—	—	0
Byte = 100	1 0 0	0 0	—	—	—	—	X	—	—	—	0
Byte = 101	1 0 1	0 0	—	—	—	—	—	X	—	—	0
Byte = 110	1 1 0	0 0	—	—	—	—	—	—	X	—	0
Byte = 111	1 1 1	0 0	—	—	—	—	—	—	—	X	0
Half = 000	0 0 0	0 1	X	X	—	—	—	—	—	—	0
Half = 001	0 0 1	1 0#	—	X	X	—	—	—	—	—	1

Table 11-11. Byte Strobe Assertion for Transfers (continued)

Program Size and byte offset	A(2:0)	HSIZE [1:0]	Data Bus Byte strobes								HUNALIGN
			B0	B1	B2	B3	B4	B5	B6	B7	
Half = 010	0 1 0	0 1	—	—	X	X	—	—	—	—	0
Half = 011	0 1 1	1 1$^{\#}$	—	—	—	X	X	—	—	—	1
Half = 100	1 0 0	0 1	—	—	—	—	X	X	—	—	0
Half = 101	1 0 1	1 0$^{\#}$	—	—	—	—	—	X	X	—	1
Half = 110	1 1 0	0 1	—	—	—	—	—	—	X	X	0
Half = 111 (2 bus transfers)	1 1 1 0 0 0	0 1* 0 0	— X	— —	— —	— —	— —	— —	— —	X —	1 0
Word = 000	0 0 0	1 0	X	X	X	X	—	—	—	—	0
Word = 001	0 0 1	1 1$^{\#}$	—	X	X	X	X	—	—	—	1
Word = 010	0 1 0	1 1$^{\#}$	—	—	X	X	X	X	—	—	1
Word = 011	0 1 1	1 1$^{\#}$	—	—	—	X	X	X	X	—	1
Word = 100	1 0 0	1 0	—	—	—	—	X	X	X	X	0
Word = 101 (2 bus transfers)	1 0 1 0 0 0	1 0* 0 0	— X	— —	— —	— —	— —	X —	X —	X —	1 0
Word = 110 (2 bus transfers)	1 1 0 0 0 0	1 0* 0 1	— X	— X	— —	— —	— —	— —	X —	X —	1 0
Word = 111 (2 bus transfers)	1 1 1 0 0 0	10* 1 0	— X	— X	— X	— —	— —	— —	— —	X —	1 1
Double Word = 000	0 0 0	1 1	X	X	X	X	X	X	X	X	0
Double Word = 001 (2 bus transfers)	0 0 1 0 0 0	1 1* 0 0	— X	X —	X —	X —	X —	X —	X —	X —	1 0
Double Word = 010 (2 bus transfers)	0 1 0 0 0 0	1 1* 0 1	— X	— X	X —	X —	X —	X —	X —	X —	1 0
Double Word = 011 (2 bus transfers)	0 1 1 0 0 0	1 1* 1 0#	— X	— X	— X	X —	X —	X —	X —	X —	1 1
Double Word = 100 (2 bus transfers)	1 0 0 0 0 0	1 1* 1 0	— X	— X	— X	— X	X —	X —	X —	X —	1 0
Double Word = 101 (2 bus transfers)	1 0 1 0 0 0	1 1* 1 1#	— X	— X	— X	— X	— X	X —	X —	X —	1 1
Double Word = 110 (2 bus transfers)	1 1 0 0 0 0	1 1* 1 1#	— X	— X	— X	— X	— X	— X	X —	X —	1 1

Table 11-11. Byte Strobe Assertion for Transfers (continued)

Program Size and byte offset	A(2:0)	HSIZE [1:0]	Data Bus Byte strobes								HUNALIGN
			B0	B1	B2	B3	B4	B5	B6	B7	
Double Word = 111 (2 bus transfers)	1 1 1 0 0 0	1 1* 1 1#	— X	— X	— X	— X	— X	— X	— X	X —	1 1

"X" indicates byte lanes involved in the transfer; Other lanes will contain driven but unused data.

These misaligned transfers drive size according to the size of the power of two aligned "container" in which the byte strobes are asserted.

* These misaligned cases drive request size according to the size specified by the load or store instruction.

Table 11-12 shows the final layout in memory for data transferred from a 64-bit GPR containing the bytes 'A B C D E F G H' to memory. Misaligned accesses that cross a double-word boundary are broken into a pair of accesses by the CPU.

Table 11-12. Big- and Little-Endian Storage

Program Size and Byte Offset	A(3:0)	HSIZE (1:0)	Even Double Word—0								0dd Double Word—1							
			B0	B1	B2	B3	B4	B5	B6	B7	B0	B1	B2	B3	B4	B5	B6	B7
Byte = 0000	0 0 0 0	0 0	H	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Byte = 0001	0 0 0 1	0 0	—	H	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Byte = 0010	0 0 1 0	0 0	—	—	H	—	—	—	—	—	—	—	—	—	—	—	—	—
Byte = 0011	0 0 1 1	0 0	—	—	—	H	—	—	—	—	—	—	—	—	—	—	—	—
Byte = 0100	0 1 0 0	0 0	—	—	—	—	H	—	—	—	—	—	—	—	—	—	—	—
Byte = 0101	0 1 0 1	0 0	—	—	—	—	—	H	—	—	—	—	—	—	—	—	—	—
Byte = 0110	0 1 1 0	0 0	—	—	—	—	—	—	H	—	—	—	—	—	—	—	—	—
Byte = 0111	0 1 1 1	0 0	—	—	—	—	—	—	—	H	—	—	—	—	—	—	—	—
Byte = 1000	1 0 0 0	0 0	—	—	—	—	—	—	—	—	H	—	—	—	—	—	—	—
Byte = 1001	1 0 0 1	0 0	—	—	—	—	—	—	—	—	—	H	—	—	—	—	—	—
Byte = 1010	1 0 1 0	0 0	—	—	—	—	—	—	—	—	—	—	H	—	—	—	—	—
Byte = 1011	1 0 1 1	0 0	—	—	—	—	—	—	—	—	—	—	—	H	—	—	—	—
Byte = 1100	1 1 0 0	0 0	—	—	—	—	—	—	—	—	—	—	—	—	H	—	—	—
Byte = 1101	1 1 0 1	0 0	—	—	—	—	—	—	—	—	—	—	—	—	—	H	—	—
Byte = 1110	1 1 1 0	0 0	—	—	—	—	—	—	—	—	—	—	—	—	—	—	H	—
Byte = 1111	1 1 1 1	0 0	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	H
B. E. Half = 0000	0 0 0 0	0 1	G	H	—	—	—	—	—	—	—	—	—	—	—	—	—	—
B. E. Half = 0001	0 0 0 1	1 0#	—	G	H	—	—	—	—	—	—	—	—	—	—	—	—	—

Table 11-12. Big- and Little-Endian Storage (continued)

Program Size and Byte Offset	A(3:0)	HSIZE (1:0)	Even Double Word—0								0dd Double Word—1							
			B0	B1	B2	B3	B4	B5	B6	B7	B0	B1	B2	B3	B4	B5	B6	B7
B. E. Half = 0010	0 0 1 0	0 1	—	—	G	H	—	—	—	—	—	—	—	—	—	—	—	—
B. E. Half = 0011	0 0 1 1	1 1[#]	—	—	—	G	H	—	—	—	—	—	—	—	—	—	—	—
B. E. Half = 0100	0 1 0 0	0 1	—	—	—	—	G	H	—	—	—	—	—	—	—	—	—	—
B. E. Half = 0101	0 1 0 1	1 0[#]	—	—	—	—	—	G	H	—	—	—	—	—	—	—	—	—
B. E. Half = 0110	0 1 1 0	0 1	—	—	—	—	—	—	G	H	—	—	—	—	—	—	—	—
B. E. Half = 0111	0 1 1 1	0 1	—	—	—	—	—	—	—	G	—	—	—	—	—	—	—	—
	1 0 0 0	0 0	—	—	—	—	—	—	—	—	H	—	—	—	—	—	—	—
B. E. Half = 1000	1 0 0 0	0 1	—	—	—	—	—	—	—	—	G	H	—	—	—	—	—	—
B. E. Half = 1001	1 0 0 1	1 0[#]	—	—	—	—	—	—	—	—	—	G	H	—	—	—	—	—
B. E. Half = 1010	1 0 1 0	0 1	—	—	—	—	—	—	—	—	—	—	G	H	—	—	—	—
B. E. Half = 1011	1 0 1 1	1 1[#]	—	—	—	—	—	—	—	—	—	—	—	G	H	—	—	—
B. E. Half = 1100	1 1 0 0	0 1	—	—	—	—	—	—	—	—	—	—	—	—	G	H	—	—
B. E. Half = 1101	1 1 0 1	1 0[#]	—	—	—	—	—	—	—	—	—	—	—	—	—	G	H	—
B. E. Half = 1110	1 1 1 0	0 1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	G	H
B. E. Half = 1111	1 1 1 1	0 1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	G
	0 0 0 0 (next dword)	0 0	H	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
L E. Half = 0000	0 0 0 0	0 1	H	G	—	—	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Half = 0001	0 0 0 1	1 0[#]	—	H	G	—	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Half = 0010	0 0 1 0	0 1	—	—	H	G	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Half = 0011	0 0 1 1	1 1[#]	—	—	—	H	G	—	—	—	—	—	—	—	—	—	—	—

Table 11-12. Big- and Little-Endian Storage (continued)

Program Size and Byte Offset	A(3:0)	HSIZE (1:0)	Even Double Word—0								0dd Double Word—1							
			B0	B1	B2	B3	B4	B5	B6	B7	B0	B1	B2	B3	B4	B5	B6	B7
L. E. Half = 0100	0 1 0 0	0 1	—	—	—	—	H	G	—	—	—	—	—	—	—	—	—	—
L. E. Half = 0101	0 1 0 1	1 0#	—	—	—	—	—	H	G	—	—	—	—	—	—	—	—	—
L. E. Half = 0110	0 1 1 0	0 1	—	—	—	—	—	—	H	G	—	—	—	—	—	—	—	—
L. E. Half = 0111	0 1 1 1	0 1	—	—	—	—	—	—	—	H	—	—	—	—	—	—	—	—
	1 0 0 0	0 0	—	—	—	—	—	—	—	—	G	—	—	—	—	—	—	—
L. E. Half = 1000	1 0 0 0	0 1	—	—	—	—	—	—	—	—	H	G	—	—	—	—	—	—
L. E. Half = 1001	1 0 0 1	1 0#	—	—	—	—	—	—	—	—	—	H	G	—	—	—	—	—
L. E. Half = 1010	1 0 1 0	0 1	—	—	—	—	—	—	—	—	—	—	H	G	—	—	—	—
L. E. Half = 1011	1 0 1 1	1 1#	—	—	—	—	—	—	—	—	—	—	—	H	G	—	—	—
L. E. Half = 1100	1 1 0 0	0 1	—	—	—	—	—	—	—	—	—	—	—	—	H	G	—	—
L. E. Half = 1101	1 1 0 1	1 0#	—	—	—	—	—	—	—	—	—	—	—	—	—	H	G	—
L. E. Half = 1110	1 1 1 0	0 1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	H	G
L. E. Half = 1111	1 1 1 1	0 1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	H
	+ 0 0 0 0 (next dword)	0 0	G	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
B. E. Word = 0000	0 0 0 0	1 0	E	F	G	H	—	—	—	—	—	—	—	—	—	—	—	—
B. E. Word = 0001	0 0 0 1	1 1#	—	E	F	G	H	—	—	—	—	—	—	—	—	—	—	—
B. E. Word = 0010	0 0 1 0	1 1#	—	—	E	F	G	H	—	—	—	—	—	—	—	—	—	—
B. E. Word = 0011	0 0 1 1	1 1#	—	—	—	E	F	G	H	—	—	—	—	—	—	—	—	—
B. E. Word = 0100	0 1 0 0	0 1 0	—	—	—	—	E	F	G	H	—	—	—	—	—	—	—	—
B. E. Word = 0101	0 1 0 1	1 0	—	—	—	—	—	E	F	G	—	—	—	—	—	—	—	—
	1 0 0 0	0 0	—	—	—	—	—	—	—	—	H	—	—	—	—	—	—	—

Table 11-12. Big- and Little-Endian Storage (continued)

Program Size and Byte Offset	A(3:0)	HSIZE (1:0)	Even Double Word—0								0dd Double Word—1							
			B0	B1	B2	B3	B4	B5	B6	B7	B0	B1	B2	B3	B4	B5	B6	B7
B. E. Word = 0110	0 1 1 0	1 0	—	—	—	—	—	—	E	F	—	—	—	—	—	—	—	—
	1 0 0 0	0 1	—	—	—	—	—	—	—	—	G	H	—	—	—	—	—	—
B. E. Word = 0111	0 1 1 1	1 0	—	—	—	—	—	—	—	E	—	—	—	—	—	—	—	—
	1 0 0 0	1 0	—	—	—	—	—	—	—	—	F	G	H	—	—	—	—	—
B. E. Word = 1000	1 0 0 0	1 0	—	—	—	—	—	—	—	—	E	F	G	H	—	—	—	—
B. E. Word = 1001	1 0 0 1	1 1$^{\#}$	—	—	—	—	—	—	—	—	—	E	F	G	H	—	—	—
B. E. Word = 1010	1 0 1 0	1 1$^{\#}$	—	—	—	—	—	—	—	—	—	—	E	F	G	H	—	—
B. E. Word = 1011	1 0 1 1	1 1$^{\#}$	—	—	—	—	—	—	—	—	—	—	—	E	F	G	H	—
B. E. Word = 1100	1 1 0 0	1 0	—	—	—	—	—	—	—	—	—	—	—	—	E	F	G	H
B. E. Word = 1101	1 1 0 1	1 0	—	—	—	—	—	—	—	—	—	—	—	—	—	E	F	G
	+ 0 0 0 0 (next dword)	0 0	H	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
B. E. Word = 1110	1 1 1 0	1 0	—	—	—	—	—	—	—	—	—	—	—	—	—	—	E	F
	+ 0 0 0 0 (next dword)	0 1	G	H	—	—	—	—	—	—	—	—	—	—	—	—	—	—
B. E. Word = 1111	1 1 1 1	1 0	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	E
	+ 0 0 0 0 (next dword)	1 0	F	G	H	—	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Word = 0000	0 0 0 0	1 0	H	G	F	E	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Word = 0001	0 0 0 1	1 1$^{\#}$	—	H	G	F	E	—	—	—	—	—	—	—	—	—	—	—
L. E. Word = 0010	0 0 1 0	1 1$^{\#}$	—	—	H	G	F	E	—	—	—	—	—	—	—	—	—	—
L. E. Word = 0011	0 0 1 1	1 1$^{\#}$	—	—	—	H	G	F	E	—	—	—	—	—	—	—	—	—
L. E. Word = 0100	0 1 0 0	1 0	—	—	—	—	H	G	F	E	—	—	—	—	—	—	—	—
L. E. Word = 0101	0 1 0 1	1 0	—	—	—	—	—	H	G	F	—	—	—	—	—	—	—	—
	1 0 0 0	0 0	—	—	—	—	—	—	—	—	E	—	—	—	—	—	—	—

Table 11-12. Big- and Little-Endian Storage (continued)

Program Size and Byte Offset	A(3:0)	HSIZE (1:0)	Even Double Word—0								0dd Double Word—1							
			B0	B1	B2	B3	B4	B5	B6	B7	B0	B1	B2	B3	B4	B5	B6	B7
L. E. Word = 0110	0 1 1 0	1 0	—	—	—	—	—	—	H	G	—	—	—	—	—	—	—	—
	1 0 0 0	0 1	—	—	—	—	—	—	—	—	F	E	—	—	—	—	—	—
L. E. Word = 0111	0 1 1 1	1 0	—	—	—	—	—	—	—	H	—	—	—	—	—	—	—	—
	1 0 0 0	1 0	—	—	—	—	—	—	—	—	G	F	E	—	—	—	—	—
L. E. Word = 1000	1 0 0 0	1 0	—	—	—	—	—	—	—	—	H	G	F	E	—	—	—	—
L. E. Word = 1001	1 0 0 1	1 1[#]	—	—	—	—	—	—	—	—	—	H	G	F	E	—	—	—
L. E. Word = 1010	1 0 1 0	1 1[#]	—	—	—	—	—	—	—	—	—	—	H	G	F	E	—	—
L. E. Word = 1011	1 0 1 1	1 1[#]	—	—	—	—	—	—	—	—	—	—	—	H	G	F	E	—
L. E. Word = 1100	1 1 0 0	1 0	—	—	—	—	—	—	—	—	—	—	—	—	H	G	F	E
L. E. Word = 1101	1 1 0 1	1 0	—	—	—	—	—	—	—	—	—	—	—	—	—	H	G	F
	+ 0 0 0 0 (next dword)	0 0	E	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Word = 1110	1 1 1 0	1 0	—	—	—	—	—	—	—	—	—	—	—	—	—	—	H	G
	+ 0 0 0 0 (next dword)	0 1	F	E	—	—	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Word = 1111	1 1 1 1	1 0	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	H
	+ 0 0 0 0 (next dword)	1 0	G	F	E	—	—	—	—	—	—	—	—	—	—	—	—	—
B.E. Double Word = 0000	0 0 0 0	1 1	A	B	C	D	E	F	G	H	—	—	—	—	—	—	—	—
B. E. Double Word = 0001	0 0 0 1	1 1	—	A	B	C	D	E	F	G	—	—	—	—	—	—	—	—
	1 0 0 0 (next dword)	0 0	—	—	—	—	—	—	—	—	H	—	—	—	—	—	—	—
B. E. Double Word = 0010	0 0 1 0	1 1	—	—	A	B	C	D	E	F	—	—	—	—	—	—	—	—
	1 0 0 0 (next dword)	0 1	—	—	—	—	—	—	—	—	G	H	—	—	—	—	—	—

Table 11-12. Big- and Little-Endian Storage (continued)

Program Size and Byte Offset	A(3:0)	HSIZE (1:0)	Even Double Word—0								0dd Double Word—1							
			B0	B1	B2	B3	B4	B5	B6	B7	B0	B1	B2	B3	B4	B5	B6	B7
B. E. Double Word = 0011	0 0 1 1	1 1	—	—	—	A	B	C	D	E	—	—	—	—	—	—	—	—
	1 0 0 0 (next dword)	1 0#	—	—	—	—	—	—	—	—	F	G	H	—	—	—	—	—
B. E. Double Word = 0100	0 1 0 0	1 1	—	—	—	—	A	B	C	D	—	—	—	—	—	—	—	—
	1 0 0 0 (next dword)	1 0	—	—	—	—	—	—	—	—	E	F	G	H	—	—	—	—
B. E. Double Word = 0101	0 1 0 1	1 1	—	—	—	—	—	A	B	C	—	—	—	—	—	—	—	—
	1 0 0 0 (next dword)	1 1#	—	—	—	—	—	—	—	—	D	E	F	G	H	—	—	—
B. E. Double Word = 0110	0 1 1 0	1 1	—	—	—	—	—	—	A	B	—	—	—	—	—	—	—	—
	1 0 0 0 (next dword)	1 1#	—	—	—	—	—	—	—	—	C	D	E	F	G	H	—	—
B. E. Double Word = 0111	0 1 1 1	1 1	—	—	—	—	—	—	—	A	—	—	—	—	—	—	—	—
	1 0 0 0 (next dword)	1 1#	—	—	—	—	—	—	—	—	B	C	D	E	F	G	H	—
B.E. Double Word = 1000	1 0 0 0	1 1	—	—	—	—	—	—	—	—	A	B	C	D	E	F	G	H
B. E. Double Word = 1001	1 0 0 1	1 1	—	—	—	—	—	—	—	—	—	A	B	C	D	E	F	G
	+0 0 0 0 (next dword)	0 0	H	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
B. E. Double Word = 1010	1 0 1 0	1 1	—	—	—	—	—	—	—	—	—	—	A	B	C	D	E	F
	+ 0 0 0 0 (next dword)	0 1	G	H	—	—	—	—	—	—	—	—	—	—	—	—	—	—
B. E. Double Word = 1011	1 0 1 1	1 1	—	—	—	—	—	—	—	—	—	—	—	A	B	C	D	E
	+ 0 0 0 0 (next dword)	1 0#	F	G	H	—	—	—	—	—	—	—	—	—	—	—	—	—
B. E. Double Word = 1100	1 1 0 0	1 1	—	—	—	—	—	—	—	—	—	—	—	—	A	B	C	D
	+ 0 0 0 0 (next dword)	1 0	E	F	G	H	—	—	—	—	—	—	—	—	—	—	—	—

Table 11-12. Big- and Little-Endian Storage (continued)

Program Size and Byte Offset	A(3:0)	HSIZE (1:0)	Even Double Word—0								0dd Double Word—1							
			B0	B1	B2	B3	B4	B5	B6	B7	B0	B1	B2	B3	B4	B5	B6	B7
B. E. Double Word = 1101	1 1 0 1	1 1	—	—	—	—	—	—	—	—	—	—	—	—	—	A	B	C
	+ 0 0 0 0 (next dword)	1 1#	D	E	F	G	H	—	—	—	—	—	—	—	—	—	—	—
B. E. Double Word = 1110	1 1 1 0	1 1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	A	B
	+ 0 0 0 0 (next dword)	1 1#	C	D	E	F	G	H	—	—	—	—	—	—	—	—	—	—
B. E. Double Word = 1111	1 1 1 1	1 1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	A
	+ 0 0 0 0 (next dword)	1 1#	B	C	D	E	F	G	H	—	—	—	—	—	—	—	—	—
L.E. Double Word = 0000	0 0 0 0	1 1	H	G	F	E	D	C	B	A	—	—	—	—	—	—	—	—
L. E. Double Word = 0001	0 0 0 1	1 1	—	H	G	F	E	D	C	B	—	—	—	—	—	—	—	—
	1 0 0 0 (next dword)	0 0	—	—	—	—	—	—	—	—	A	—	—	—	—	—	—	—
L. E. Double Word = 0010	0 0 1 0	1 1	—	—	H	G	F	E	D	C	—	—	—	—	—	—	—	—
	1 0 0 0 (next dword)	0 1	—	—	—	—	—	—	—	—	B	A	—	—	—	—	—	—
L. E. Double Word = 0011	0 0 1 1	1 1	—	—	—	H	G	F	E	D	—	—	—	—	—	—	—	—
	1 0 0 0 (next dword)	1 0#	—	—	—	—	—	—	—	—	C	B	A	—	—	—	—	—
L. E. Double Word = 0100	0 1 0 0	1 1	—	—	—	—	H	G	F	E	—	—	—	—	—	—	—	—
	1 0 0 0 (next dword)	1 0	—	—	—	—	—	—	—	—	D	C	B	A	—	—	—	—
L. E. Double Word = 0101	0 1 0 1	1 1	—	—	—	—	—	H	G	F	—	—	—	—	—	—	—	—
	1 0 0 0 (next dword)	1 1#	—	—	—	—	—	—	—	—	E	D	C	B	A	—	—	—
L. E. Double Word = 0110	0 1 1 0	1 1	—	—	—	—	—	—	H	G	—	—	—	—	—	—	—	—
	1 0 0 0 (next dword)	1 1#	—	—	—	—	—	—	—	—	F	E	D	C	B	A	—	—

Table 11-12. Big- and Little-Endian Storage (continued)

Program Size and Byte Offset	A(3:0)	HSIZE (1:0)	Even Double Word—0								0dd Double Word—1							
			B0	B1	B2	B3	B4	B5	B6	B7	B0	B1	B2	B3	B4	B5	B6	B7
L. E. Double Word = 0111	0 1 1 1	1 1	—	—	—	—	—	—	—	H	—	—	—	—	—	—	—	—
	1 0 0 0 (next dword)	1 1#	—	—	—	—	—	—	—	—	G	F	E	D	C	B	A	—
L.E. Double Word = 1000	0 0 0 0	1 1	—	—	—	—	—	—	—	—	H	G	F	E	D	C	B	A
L. E. Double Word = 1001	1 0 0 1	1 1	—	—	—	—	—	—	—	—	—	H	G	F	E	D	C	B
	+ 0 0 0 0 (next dword)	0 0	A	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Double Word = 1010	1 0 1 0	1 1	—	—	—	—	—	—	—	—	—	—	H	G	F	E	D	C
	+ 0 0 0 0 (next dword)	0 1	B	A	—	—	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Double Word = 1011	1 0 1 1	1 1	—	—	—	—	—	—	—	—	—	—	—	H	G	F	E	D
	+ 0 0 0 0 (next dword)	1 0#	C	B	A	—	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Double Word = 1100	1 1 0 0	1 1	—	—	—	—	—	—	—	—	—	—	—	—	H	G	F	E
	+ 0 0 0 0 (next dword)	1 0	D	C	B	A	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Double Word = 1101	1 1 0 1	1 1	—	—	—	—	—	—	—	—	—	—	—	—	—	H	G	F
	+ 0 0 0 0 (next dword)	1 1#	E	D	C	B	A	—	—	—	—	—	—	—	—	—	—	—
L. E. Double Word = 1110	1 1 1 0	1 1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	H	G
	+ 0 0 0 0 (next dword)	1 1#	F	E	D	C	B	A	—	—	—	—	—	—	—	—	—	—
L. E. Double Word = 1111	1 1 1 1	1 1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	H
	+ 0 0 0 0 (next dword)	1 1#	G	F	E	D	C	B	A	—	—	—	—	—	—	—	—	—

Note:

Assumes a 64-bit GPR contains ‘A B C D E F G H’

[#] These misaligned transfers drive size according to the size of the power of two aligned “container” in which the byte strobes are asserted.

11.2.6 Transfer Control Signals

The following paragraphs describe the transfer control signals.

11.2.6.1 Transfer Ready (p_d_hready, p_i_hready)

The **p_[d,i]_hready** input signal indicates completion of a requested transfer operation. An external device asserts **p_[d,i]_hready** to terminate the transfer. The **p_[d,i]_hresp[2:0]** signals indicate the status of the transfer.

11.2.6.2 Transfer Response (p_d_hresp[2:0], p_i_hresp[1:0])

The **p_d_hresp[2:0]** and **p_i_hresp[1:0]** signals indicate the status of a terminating transfer on the respective interfaces. Table 11-13 shows the definitions of the **p_d_hresp[2:0]** and **p_i_hresp[1:0]** encodings. Table 11-14 shows the definitions of the **p_i_hresp[1:0]** encoding.

Table 11-13. p_d_hresp[2:0] Transfer Response Encoding

p_d_hresp[2:0]	Response Type
000	OKAY—transfer terminated normally
001	ERROR—transfer terminated abnormally
010	Reserved (RETRY not supported in AHB-Lite protocol)
011	Reserved (SPLIT not supported in AHB-Lite protocol)
100	XFAIL—Exclusive store failed (**stwcx.** did not completed successfully)
101	Reserved
110	Reserved
111	Reserved

Table 11-14. p_i_hresp[1:0] Transfer Response Encoding

p_i_hresp[1:0]	Response Type
00	OKAY—transfer terminated normally
01	ERROR—transfer terminated abnormally
10	Reserved (RETRY not supported in AHB-Lite protocol)
11	Reserved (SPLIT not supported in AHB-Lite protocol)

The ERROR and XFAIL responses are required to be two cycle responses. In this case, the ERROR or XFAIL responses must be signaled one cycle prior to assertion of **p_[d,i]_hready** and must remain unchanged during the cycle **p_[d,i]_hready** is asserted.

The XFAIL response is signaled to the CPU via the **p_d_xfail_b** internal signal.

11.2.6.3 Bus Stall Global Write Request (p_stall_bus_gwrite)

The active-high **p_stall_bus_gwrite** signal is provided to request that new bus activity for global writes (writes with the M page attribute set) be stalled (postponed) for a period of time. When asserted, no new transfer requests are generated for global writes following initiation and completion of all currently requested and outstanding accesses. This signal is provided to allow control over global write access initiation to prevent overruns or overflows of external agents that observe or act upon bus transfers, but are not actually addressed slaves. One particular use of this throttling mechanism is to prevent overflow of the snoop (coherency) FIFO in another CPU or a trace FIFO present in the system.

When asserted, no new global write transaction request are generated, although a pending transaction (p_htrans ≠ IDLE) awaiting completion of an outstanding transaction will still be taken and performed—unless an error response is received for the current outstanding transaction and the pending transaction is canceled.

11.2.7 AHB Clock Enable Signals

The following paragraphs describe the AHB clock enable signals. These inputs are used to qualify the processor **m_clk** edges used for AHB output signal state updates and AHB input signal sampling for the memory interfaces. This allows for system AHB interfaces that run at sub-multiples of the **m_clk** frequency. These signals do not affect non-AHB interface signals.

11.2.7.1 Instruction AHB Clock Enable (p_i_ahb_clken)

The **p_i_ahb_clken** input signal is used to qualify the rising edges of m_clk on which the input signals **p_i_hready**, **p_i_hresp[1:0]** and **p_i_hrdata[63:0]** are sampled. (Note that by definition, **p_i_hrdata[63:0]** sampling is also qualified by the recognized assertion of **p_i_hready**, per the AHB protocol). When driven low, no sampling of these signals occurs, since **m_clk** is gated at the sampling logic. The **p_i_ahb_clken** input signal is also used to qualify the rising edges of **m_clk** on which the output signals **p_i_haddr[31:0]**, **p_i_hbstrb[7:0]**, **p_i_hburst[1:0]**, **p_i_hmaster[3:0]**, **p_i_hprot[5:0]**, **p_i_hsize[1:0]**, **p_i_htrans[1:0]**, and **p_i_hunalign** change state (by definition, in conjunction with the **p_i_hready** input per the AHB protocol).

The **p_i_ahb_clken** signal should normally be driven (change state) off the falling edge of **m_clk** to ensure the proper setup and hold times surrounding the **m_clk** high period. It must remain stable throughout the duration of **m_clk** high. This signal is not internally synchronized. It should be tied high when operating the data AHB at **m_clk** frequency. The integration guide defines the required setup time before **m_clk** rises and hold time after **m_clk** falls.

11.2.7.2 Data AHB Clock Enable (p_d_ahb_clken)

The **p_d_ahb_clken** input signal is used to qualify the rising edges of **m_clk** on which the input signals **p_d_hready**, **p_d_hresp[2:0]**, and **p_d_hrdata[63:0]** are sampled. (Note that by definition, **p_d_hrdata[63:0]** sampling is also qualified by the recognized assertion of **p_d_hready**, per the AHB protocol). When driven low, no sampling of these signals occurs, since **m_clk** is gated at the sampling logic. The **p_d_ahb_clken** input signal is also used to qualify the rising edges of **m_clk** on which the output signals **p_d_haddr[31:0]**, **p_d_hbstrb[7:0]**, **p_d_hburst[1:0]**, **p_d_hmaster[3:0]**,

p_d_hprot[5:0], **p_d_hsize[1:0]**, **p_d_htrans[1:0]**, **p_d_hunalign**, **p_d_hwdata[63:0]**, and **p_d_hwrite** change state (by definition, in conjunction with the **p_d_hready** input per the AHB protocol).

The **p_d_ahb_clken** signal should normally be driven (change state) off the falling edge of **m_clk** to ensure the proper setup and hold times surrounding the **m_clk** high period. It must remain stable throughout the duration of **m_clk** high. This signal is not internally synchronized. It should be tied high when operating the data AHB at **m_clk** frequency. The integration guide defines the required setup time before **m_clk** rises and hold time after **m_clk** falls.

11.2.8 Master ID Configuration Signals

The following paragraphs describe the master ID configuration signals. These inputs are used to drive the **p_[d,i]_hmaster[3:0]** outputs when a bus cycle is active.

11.2.8.1 CPU Master ID (p_masterid[3:0])

The **p_masterid[3:0]** input signals configure the master ID for the CPU. These values are driven on the **p_[d,i]_hmaster[3:0]** outputs for a CPU-initiated bus cycle.

11.2.8.2 Nexus Master ID (nex_masterid[3:0])

The **nex_masterid[3:0]** input signals configure the master ID for the Nexus 3 unit. These values are driven on the **p_d_hmaster[3:0]** outputs for a Nexus 3 initiated bus cycle.

11.2.9 Coherency Control Signals

The following paragraphs describe the signals that control the cache coherency hardware functions. Examples of operation are provided in Section 11.3.5, “Cache Coherency Interface Operation.”

11.2.9.1 Snoop Ready (p_snp_rdy)

This active-high output signal indicates that the CPU is ready to accept a new snoop request. When asserted, it indicates that a new snoop cycle may be requested via the **p_snp_req** input during the following two clock cycles. When this signal is negated, a new snoop request will not be accepted after the next clock cycle, even if **p_snp_req** is asserted. This signal is asserted when the internal snoop queue contains two or more available entries for a new snoop request if either a request is pending or if three or more entries are available and no request is pending. The protocol is designed to prevent unnecessary transitions of the **p_snp_rdy** signal, as well as to support using **p_snp_rdy** to affect the **p_stall_bus_gwrite** input of another CPU to prevent queue overruns.

11.2.9.2 Snoop Request (p_snp_req)

This active-high input signal indicates that the CPU should perform a new snoop request operation. When asserted, it indicates that a new snoop cycle is being requested based on additional information provided on the **p_snp_cmd[0:1]**, **p_snp_addr[0:26]**, and **p_snp_id_in[0:3]** input signals. A new snoop request is ignored if the **p_snp_rdy** signal was negated during the previous two clock cycles, even if **p_snp_req** is asserted.

11.2.9.3 Snoop Command Input (p_snp_cmd_in[0:1])

These input signals provide a command indicator for a snoop request. The command value is stored in the snoop queue along with the snoop address and snoop ID value. Table 11-15 shows the definitions of the **p_snp_cmd[0:1]** encodings.

Table 11-15. p_snp_cmd[0:1] Snoop Command Encoding

p_snp_cmd[0:1]	Response Type
00	NULL—no status bit operation performed, lookup is performed (queue entry allocated) .
01	INV—invalidate matching cache entry (queue entry allocated)
10	SYNC—synchronize snoop queue (queue entry allocated). p_snp_addr[0:26] is unused.
11	Reserved—do not use

The NULL command is used to test interface handshaking and for other status gathering purposes. The NULL command performs a snoop lookup operation, but performs no actual cache tag or status modifications (even in the presence of tag parity or EDC errors). The INV command causes a snoop lookup and subsequent invalidation of a matching cache line. The SYNC command causes the snoop queue to be emptied with highest priority relative to CPU requests.

11.2.9.4 Snoop Request ID Input (p_snp_id_in[0:3])

These input signals provide an identifier value for a snoop request. The identifier value is stored in the snoop queue along with the snoop address and snoop command, and is only used by the CPU to be reflected on the **p_snp_id_out[0:3]** outputs when a snoop cycle is subsequently acknowledged via the **p_snp_ack** output.

11.2.9.5 Snoop Address Input (p_snp_addr_in[0:26])

These input signals provide the address value for a snoop request. The address value is stored in the snoop queue along with the snoop ID value and snoop command, and is used by the CPU to perform a cache line lookup when a snoop cycle is subsequently performed to the cache from the queue. The snoop address signals are used to index the cache and perform a tag compare with the physical cache tags. These inputs are not translated, thus they reflect the physical addresses of cached memory.

11.2.9.6 Snoop Acknowledge (p_snp_ack)

This active high output signal is used to acknowledge that a previous snoop command request has been performed. When asserted, the signal indicates that the **p_snp_id_out[0:3]** and **p_snp_resp[0:4]** outputs are valid, and reflect the result of a completed snoop command.

11.2.9.7 Snoop Request ID Output (p_snp_id_out[0:3])

These output signals provide an ID value for a snoop request. The ID value is the value of **p_snp_id_in[0:3]** which was stored in the snoop queue along with the snoop address and snoop command during a previous snoop command request, and are only used by the CPU to be reflected on the **p_snp_id_out[0:3]** outputs when a snoop command is subsequently acknowledged via the **p_snp_ack** output.

11.2.9.8 Snoop Response (p_snp_resp[0:4])

These output signals provide a response indicator for a processed snoop command request. The command value is stored in the snoop queue along with the snoop address and snoop tag value. Table 11-15 shows the definitions of the **p_snp_resp[0:4]** encodings.

Table 11-16. p_snp_resp[0:4] Snoop Response Encoding

p_snp_resp[0:4][1]	Response Type
000cc	Null—no operation performed or no matching cache entry
001cc	Reserved
010cc	ERROR—Error in processing a snoop invalidation request due to TAG parity error. • For NULL commands, a tag parity error occurred and no hit to a tag without error occurred. No modification of cache entries; no machine check generated internally. • For INV commands, possible invalidation of locked line with tag parity error occurred or dirty line left valid with tag parity error. Machine check generated internally.
01100	SYNC—Sync completed; snoop queue synchronized
100cc	HIT Clean—matching unlocked cache entry found
101cc	HIT Dirty—matching unlocked dirty cache entry found
110cc	HIT Locked—matching clean locked cache entry found
111cc	HIT Dirty Locked - matching dirty locked cache entry found

[1] cc = # collapsed requests
00 = no collapsing
01 = two requests combined
10 = three requests combined
11 = four requests combined

11.2.9.9 Cache Stalled (p_cac_stalled)

The active-high **p_cac_stalled** output signal is used to indicate that a CPU access to the data cache is stalled due to a snoop access to the cache. This signal may be monitored by system logic to determine the impact of snooping on CPU performance and to adjust the rate of snoops accordingly to minimize or distribute stall cycles.

11.2.9.10 Data Cache Enabled (p_d_cache_en)

The active-high **p_d_cache_en** output signal is used to indicate that the data cache is enabled or disabled. When disabled, no snoop lookups are performed, and a default null response is given for snoop requests.

This signal may be monitored by system logic to cancel pending snoop requests or to manage a directory ownership or snoop filter by noting when the cache has been disabled and enabled. This signal reflects the state of the L1CSR0[DCE] control bit.

11.2.10 Memory Synchronization Control Signals

The following paragraphs describe the signals that form the memory synchronization control functions. Examples of operation are shown in Section 11.3.3, “Memory Synchronization Control Operation.”

11.2.10.1 Synchronization Request In (p_sync_req_in)

This active-high input signal indicates that a synchronization operation is being requested by system logic. Assertion of this signal causes the CPU to empty the snoop queue of all valid entries present at the time the **p_sync_req_in** input was asserted, including any valid snoop command request accepted on the same clock cycle. This is a heavyweight synchronization operation that may affect system performance. This signal should remain asserted until acknowledged via assertion of the **p_sync_ack_out** signal; otherwise, proper synchronization of all queues is not guaranteed. This signal is allowed to negate early, however, if an interrupt causes a synchronization operation request to be aborted. Early negation does not guarantee that synchronization will not be aborted as requested.

This signal is not sampled during the stopped low power state and will not be acknowledged unless the stopped state is exited and the signal is still seen asserted. SoC logic should ensure that no undesired system delay or deadlock can occur due to this behavior in the stopped state.

11.2.10.2 Synchronization Request Acknowledge Out (p_sync_ack_out)

This active-high output signal indicates that a synchronization operation being requested by system logic via the **p_sync_req_in** input has completed. Assertion of this signal occurs after the CPU has emptied the snoop queue of all valid entries present at the time the **p_sync_req_in** input was asserted, including any valid snoop command request accepted on the same clock cycle.

This signal is qualified with the sampled value of **p_sync_req_in** and will negate the cycle following negation of **p_sync_req_in**. If **p_sync_req_in** is negated prior to assertion of **p_sync_ack_out**, **p_sync_ack_out** is not asserted.

During the stopped state, **p_sync_ack_out** remains negated. SoC logic must be aware of this and handle any synchronization request handshaking required to prevent a deadlock condition when another CPU attempts to execute a synchronization instruction and handshake a synchronization operation.

11.2.10.3 Synchronization Request Out (p_sync_req_out)

This active-high output signal indicates that a synchronization operation is being requested by the CPU. Assertion of this signal occurs during execution of an **msync** and **mbar** with MO = 0 or 1 instruction by the CPU after it has suspended instruction and data fetches and emptied the store buffer. Assertion of this signal does not occur for **mbar** with MO = 2.

This signal remains asserted until acknowledged via assertion of the **p_sync_ack_in** signal unless a pending interrupt occurs. In this case, the synchronization operation is aborted and restarted at a later time, and the **p_sync_req_output** is negated.

11.2.10.4 Synchronization Request Acknowledge In (p_sync_ack_in)

This active-high input signal indicates that a synchronization operation being requested by the assertion of **p_sync_req_out** by the CPU has completed.

This signal is sampled beginning with the clock cycle following assertion of **p_sync_req_out**, and will cause negation of **p_sync_req_out** the cycle after it is recognized as asserted. This signal should be negated the cycle after **p_sync_req_out** negates. This signal is ignored during the clock cycle that **p_sync_req_out** is initially asserted.

11.2.11 Interrupt Signals

The following paragraphs describe the signals that control the interrupt functions. Interrupt request inputs **p_extint_b** and **p_critint_b** to the core are level sensitive, not edge-triggered. Therefore, the interrupt controller module must keep the interrupt request as well as the **p_voffset** or **p_avec_b** inputs (as appropriate) asserted until the interrupt is serviced to guarantee that the CPU core recognizes the request. Once a request is generated, there is no guarantee the CPU will not recognize the interrupt request even if the request is later removed. Interrupt requests must be held stable to avoid spurious responses. The interrupt inputs **p_nmi_b** and **p_mcp_b** are transition sensitive as described in Section 11.2.11.8, “Machine Check (p_mcp_b) and Section 11.2.11.3, “Nonmaskable Input Interrupt Request (p_nmi_b).

11.2.11.1 External Input Interrupt Request (p_extint_b)

This active-low signal provides the External Input interrupt request to the e200 core. **p_extint_b** is masked by MSR[EE]. This signal is not internally synchronized by the e200 core. It must meet setup and hold time constraints relative to **m_clk** when the e200 core clock is running. This signal is level sensitive and must remain asserted to be guaranteed to be recognized.

11.2.11.2 Critical Input Interrupt Request (p_critint_b)

This active-low signal provides the Critical Input interrupt request to the e200 core. **p_critint_b** is masked by the MSR[CE] bit. This signal is not internally synchronized by the e200 core. It must meet setup and hold time constraints relative to **m_clk** when the e200 core clock is running. This signal is level sensitive and must remain asserted to be guaranteed to be recognized.

11.2.11.3 Nonmaskable Input Interrupt Request (p_nmi_b)

This active-low, transition sensitive signal provides a nonmaskable interrupt request to the e200 core. This signal is not internally synchronized by the e200 core. It must meet setup and hold time constraints to **m_clk** when the e200 core clock is running. The **p_nmi_b** input is sampled on two consecutive **m_clk** periods to detect a transition from the negated to the asserted state. Initiation of exception processing for the NMI is internally qualified with this transition.

Note that when the core is halted or stopped without clocks, transitions on this signal are not immediately detected. The **p_ipend** and **p_wakeup** signals are asserted to indicate to system logic that an interrupt is pending. The clocks should be started, and the **p_halt** and **p_stop** inputs should be negated in order for the interrupt to be processed.

11.2.11.4 Interrupt Pending (p_ipend)

This active-high signal indicates that an asserted **p_extint_b, p_critint_b**, or **p_nmi_b** interrupt request input, or an enabled timer facility interrupt (watchdog, fixed-interval, or decrementer) has been recognized internally by the core and is enabled by the appropriate bit in the MSR, (**p_nmi_b** is never masked), and is asserted combinationally from the qualified interrupt request inputs as well as when the MCSR[NMI] syndrome bit is set. The **p_ipend** signal can be used to signal other bus masters or a bus arbiter that an interrupt condition is pending. External power management logic can use this output to control operation of the core and other logic or may use the **p_wakeup** signal similarly. Actual handling of the interrupt request may be delayed due to higher priority exceptions; assertion of **p_ipend** does not mean that exception processing for the interrupt has begun. The **p_nmi_b** input will affect the **p_ipend** signal slightly differently; the **p_ipend** output will assert any time the **p_nmi_b** input is asserted or whenever the MCSR[NMI] syndrome bit is set.

11.2.11.5 Auto-vector (p_avec_b)

This active-low signal is asserted with either the **p_extint_b** or **p_critint_b** interrupt request to request use of the internal IVOR4 or IVOR0 registers for obtaining an exception vector offset. If this signal is negated when a **p_extint_b** or **p_critint_b** interrupt is requested, an external vector offset is taken from the **p_voffset[0:15]** input signals. This signal is level sensitive and must remain asserted to be guaranteed to be recognized. This signal must be driven to a valid state during each clock cycle that either **p_extint_b** or **p_critint_b** is asserted.

11.2.11.6 Interrupt Vector Offset (p_voffset[0:15])

These input signals provide a vector offset to be used when exception processing begins for an incoming interrupt request. These signals are sampled along with the **p_extint_b** and **p_critint_b** interrupt request inputs, and must be driven to a valid value when either of these signals is asserted unless the **p_avec_b** signal is also asserted. If **p_avec_b** is asserted, these inputs are not used. The **p_voffset[0:15]** signals correspond to bits 16–31 of the IVOR registers. **p_voffset[0:11]** are used in forming the exception handler address, and **p_voffset[12:15]** are reserved and should be driven low. The **p_voffset[0:15]** signals are level sensitive and must remain asserted to be guaranteed to be recognized correctly. In addition, these signals must be asserted concurrently with the **p_extint_b** and **p_critint_b** inputs when used.

11.2.11.7 Interrupt Vector Acknowledge (p_iack)

The **p_iack** output signal provide an interrupt vector acknowledge indicator to allow external interrupt controllers to be informed when a critical input or external input interrupt is being processed. The **p_iack** signal will be asserted after the cycle in which the **p_avec_b** and **p_voffset[0:15]** signals are sampled in preparation for exception processing. See Figure 11-66 and Figure 11-67 for timing diagrams of operation.

11.2.11.8 Machine Check (p_mcp_b)

This active-low, transition sensitive signal provides a Machine Check interrupt request to the e200 core. **p_mcp_b** is masked by HID0[EMCP]. This signal is not internally synchronized by the e200 core and thus must meet setup and hold time constraints to **m_clk** when the e200 core clock is running. The **p_mcp_b** input is sampled on two consecutive **m_clk** periods to detect a transition from the negated to the asserted state.

Note that when the core is halted or stopped without clocks, transitions on this signal will not be immediately detected, so it must be held asserted until it can be recognized with the **m_clk** running.

The **p_mcp_b** signal is sampled while the e200 core is in debug mode or is in the waiting, halted, or stopped power management states if the **m_clk** is running. See Section 11.2.19.1, “Wait, Halt, Stop Signals.”

11.2.12 Lockstep Enable Signal (p_lkstep_en)

The **p_lkstep_en** signal enables lockstep cross-signaling operation for the caches and the Nexus1 (OnCE) unit. When asserted, the cache and debug lockstep cross-signaling inputs are enabled. When negated, these input signals are ignored, but the cross-signaling output signals are still driven. Refer to Section 11.2.13, “Cache Error Cross-signaling Signals,” and Section 11.2.23, “Debug Lockstep Cross-signaling Signals.” Transitions on this signal must be properly coordinated by the SoC to ensure that the enabling and disabling of lockstep operation is performed at appropriate operational boundaries or undefined behavior may result.

11.2.13 Cache Error Cross-signaling Signals

The following sections describe the cache error cross-signaling interface signals. Section 11.3.4, “Cache Error Cross-signaling Operation,” provides examples of operation.

11.2.13.1 Cache Tag Error Out (p_[d,i]_cache_tagerr_out)

The active-high **p_d_cache_tagerr_out**/**p_i_cache_tagerr_out** output signal indicates that a valid data or instruction cache tag parity error has occurred during this cycle. It is only signaled if a cache operation or exception would be signaled by the detected error condition.

When L1CSR0[DCEA]/L1CSR1[ICEA] indicates machine check generation on error, assertion of this signal indicates a machine check will be signaled for the access, or for dcbi/icbi operations, indicates that a remote invalidation of one or more cache lines should occur. When L1CSR0[DCEA]/L1CSR1[ICEA] indicates auto-invalidation on error, assertion of this signal indicates that the cache will insert an additional cycle to perform auto-invalidation on cache ways with uncorrectable tag errors, and to correct tags in ways with correctable errors.This signal is reset to 0.

11.2.13.2 Cache Data Error Out (p_[d,i]_cache_dataerr_out)

The active-high **p_d_cache_dataerr_out/p_i_cache_dataerr_out**) output signal indicates a valid data or instruction cache data array parity error has occurred during this cycle. It is only signaled if a cache operation or exception would be signaled by the detected error condition. This signal is only associated on a valid cache hit with no tag error generation or signaling, with a data parity error on the hitting way. This signal is reset to 0.

11.2.13.3 Cache Push Data Error Out (p_d_pusherr_out)

The active-high **p_d_pusherr_out** output signal indicates a data cache data array parity error has occurred or is pending for a cache push (copyback) operation during this cycle. It is only signaled if a cache exception would be signaled by the detected error condition. This signal is associated with a data parity error on the copyback data read from the data array. This signal is reset to 0.

11.2.13.4 Cache Error Address Out (p_[d,i]_cerraddr_out[0:31])

The active-high **p_d_cerraddr_out[0:31]/p_i_cerraddr_out[0:31]** output signals are used to provide the physical address corresponding to a data or instruction cache error signaled by the **p_d_cache_tagerr_out** (**p_i_cache_tagerr_out**) and **p_d_cache_dataerr_out** (**p_i_cache_dataerr_out**) output signals. These signals should be qualified with the assertion of **p_d_cache_tagerr_out** (**p_i_cache_tagerr_out**) or **p_d_cache_dataerr_out** (**p_i_cache_dataerr_out**). These signals are undefined following reset. These signals are provided for external monitoring logic only.

11.2.13.5 Cache Tag Error Way(s) Out (p_[d,i]_tagerrway_out[0:3])

The active-high **p_d_tagerrway_out[0:3]** (**p_i_tagerrway_out[0:3]**) output signals are used to indicate which way(s) of the data or instruction cache encountered an uncorrectable cache tag array error. These signals should be qualified with the assertion of **p_d_cache_tagerr_out** (**p_i_cache_tagerr_out**). Either a machine check or an invalidation occurs depending on the L1CSR0/L1CSR1 settings. In auto-invalidation/correction mode, the ways associated with asserted signals are invalidated due to uncorrectable errors. Correctable ways are not signaled with these outputs. These signals are reset to 0.

11.2.13.6 Cache Dirty Error Way(s) Out (p_d_drterrway_out[0:3])

The active-high **p_d_drterrway_out[0:3]** output signals are used to indicate which way(s) of the data cache encountered a dirty error. The assertion indicates that the cache will perform a correction cycle in which the dirty bits are set for those ways with errors. These signals should be qualified with the assertion of **p_d_cache_tagerr_out**. It is possible for multiple ways to be indicated, and it is also possible for these signals to assert in conjunction with the **p_d_tagerrway_out[0:3]** outputs. In auto-invalidation/correction mode, the dirty bits for the way are set if the corresponding way is not to be invalidated. These signals are reset to 0.

11.2.13.7 Cache Lock Error Way(s) Out (p_[d,i]_lkerrway_out[0:3])

The active-high **p_[d,i]_lkerrway_out[0:3]** output signals indicate which way(s) of the data cache encountered an uncorrectable lock parity error. The assertion indicates that the cache has encountered an uncorrectable lock error for those ways with errors signaled. When this occurs, the cache is operating in correction/auto-invalidation mode (L1CSR0[DCEA]/L1CSR1[ICEA] = 01), and **p_lkstep_en** is asserted, the cache rewrites the corresponding lock bits of the error ways to 0110 to indicate a double-bit error. This allows the corresponding emulation of the error in the external cache.

In machine check mode (L1CSR0[DCEA]/L1CSR1[ICEA] = 00), no rewrite of the lock bits is performed. These signals should be qualified with the assertion of **p_[d,i]_cache_tagerr_out**. It is possible for multiple ways to be indicated, and it is also possible for these signals to assert in conjunction with the **p_d_tagerrway_out[0:3]** and/or **p_d_drterrway_out[0:3]** outputs. These signals are reset to 0.

11.2.13.8 Cache Data Error In (p_[d,i]_cache_dataerr_in)

When assertion of the **p_lkstep_en** input signal enables the cross-signaling operation, the active-high **p_d_cache_dataerr_in** or **p_i_cache_dataerr_in** input signal indicates that a data or instruction cache data array parity error is being cross-signaled from another cache during this cycle. Assertion of this signal should be used to cause a data or instruction cache data parity error to be emulated for the indicated way(s) of the cache. Depending on the settings of L1CSR0/L1CSR1, either a machine check or an auto-reload of the hitting cache line should occur.

11.2.13.9 Cache Push Data Error In (p_d_pusherr_in)

When assertion of the **p_lkstep_en** input signal enables the cross-signaling operation, the active-high **p_d_pusherr_in** input signal indicates that a data cache data array parity error for a cache push (copyback) operation is being cross-signaled from another cache during this cycle. A machine check should occur for the push parity error. This signal may assert for multiple cycles for a pending push parity error case.

11.2.13.10 Cache Tag Error In (p_[d,i]_cache_tagerr_in)

When assertion of the **p_lkstep_en** input signal enables the cross-signaling operation, the active-high **p_d_cache_tagerr_in** or **p_i_cache_tagerr_in** input signal indicates that a data or instruction cache tag parity error is being cross-signaled from another cache during this cycle. Depending on the settings of L1CSR0[DCEA]/L1CSR1[ICEA], either a machine check or an invalidation should occur.

Assertion of this signal indicates that the values of **p_d_tagerrway_in[0:3]** (**p_i_tagerrway_in[0:3]**) should be used to cause a data or instruction cache parity error to be emulated for the indicated way(s) of the data or instruction cache. It also indicates that the **p_drterrway_in[0:3]** inputs should be used to emulate a dirty error correction when operating in auto-invalidation/correction mode.

11.2.13.11 Cache Tag Error Way(s) In (p_[d,i]_tagerrway_in[0:3])

When assertion of the **p_lkstep_en** input signal enables the cross-signaling operation, the active-high **p_d_tagerrway_in[0:3]** or **p_i_tagerrway_in[0:3**) input signals indicate whether the corresponding ways of the data or instruction cache should emulate a cache error. These signals should be qualified with the assertion of **p_d_cache_tagerr_in** or **p_i_cache_tagerr_in**. Depending on the settings of L1CSR0/L1CSR1, either a machine check or an invalidation occurs.

In auto-invalidation/correction mode, the ways associated with asserted signals are invalidated due to uncorrectable errors in the signaling cache.

11.2.13.12 Cache Dirty Error Way(s) In (p_d_drterrway_in[0:3])

When assertion of the **p_lkstep_en** input signal enables the cross-signaling operation, the active-high **p_d_drterrway_in[0:3]** input signals indicate whether the corresponding ways of the data cache should emulate a cache dirty error. These signals should be qualified with the assertion of **p_d_cache_tagerr_in**.

In auto-invalidation/correction mode, if the corresponding ways are not to be invalidated, the ways associated with asserted signals perform a correction cycle in which the dirty bits are set to 1 for those ways. These signals should be qualified with the assertion of **p_d_cache_tagerr_in**. It is possible for multiple ways to be indicated, and it is also possible for these signals to assert in conjunction with the **p_d_tagerrway_in[0:3]** inputs, which have priority for a given way.

11.2.13.13 Cache Lock Error Way(s) in (p_[d,i]_lkerrway_in[0:3])

When assertion of the **p_lkstep_en** input signal enables the cross-signaling operation, the active-high **p_[d,i]_lkerrway_in[0:3]** input signals indicate whether the corresponding ways of the data cache in another CPU have encountered an uncorrectable lock parity error. These signals should be qualified with the assertion of **p_d_cache_tagerr_in**.

It is possible for multiple ways to be indicated, and it is also possible for these signals to assert in conjunction with the **p_[d,i]_tagerrway_in[0:3]** and/or **p_d_drterrway_in[0:3]** inputs. When received, the cache rewrites the corresponding lock bits to 0110 to indicate a double-bit error in order to emulate the error in the external cache.

11.2.14 External Translation Alteration Signals

The following paragraphs describe the external translation alteration interface signals. A description of operation is provided in Section 10.11, “External Translation Alterations for Realtime Systems.

11.2.14.1 External PID Enable (p_extpid_en)

The active-high **p_extpid_en** input signal is used to enable the external translation alteration interface. Enabling of the dynamic mapping capability is controlled by asserting the **p_extpid_en** control input. This input is sampled with the rising edge of the clock, and when asserted, allows for the dynamic remapping capability to be used.

11.2.14.2 External PID In (p_extpid[6:7])

The active-high **p_extpid[6:7]** input signals are used to provide the PID[6:7] comparison values for certain TLB entries. These signals are qualified with the assertion of **p_extpid_en**.

11.2.15 Timer Facility Signals

The following subsections describe the processor signals associated with the timer facilities (time base, watchdog, fixed-interval, and decrementer).

11.2.15.1 Timer Disable (p_tbdisable)

The active-high **p_tbdisable** input signal is used to disable the internal time base and decrementer counters. When this signal is asserted, time base and decrementer updates are frozen. When this signal is negated, time base and decrementer updates are unaffected. This signal may be used to freeze the state of the time base and decrementer during low power or debug operation. This signal is not internally synchronized by the e200 core and thus must meet setup and hold time constraints relative to **m_clk** when the e200 core clock is running, as well as to **p_tbclk** when selected as an alternate clock source for the time base.

11.2.15.2 Timer External Clock (p_tbclk)

The active-high **p_tbclk** input signal is used as an alternate clock source for the time base and decrementer counters. Selection of this clock is made using the HID0[SEL_TBCLK] control bit (see Section 2.4.11, “Hardware Implementation Dependent Register 0 (HID0)”). This clock source must be synchronous to the **m_clk** input, and cannot exceed 50% of the **m_clk** frequency. This signal must be driven such that it changes state on the falling edge of **m_clk**.

11.2.15.3 Timer Interrupt Status (p_tbint)

The active-high **p_tbint** output signal is used to indicate that an internal timer facility unit is generating an interrupt request (TSR[WIS] = 1 and TCR[WIE] = 1 and MSR[CE] = 1, or TSR[DIS] = 1 and TCR[DIE] = 1 and MSR[EE] = 1, or TSR[FIS] = 1 and TCR[FIE] = 1 and MSR[CE] = 1). This signal may be used to exit low power operation, or for other system purposes.

11.2.16 Processor Reservation Signals

The following subsections describe processor reservation signals associated with the **lbarx**, **lharx**, **lwarx**, **stbcx.**, **sthcx.**, and **stwcx.** instructions.

11.2.16.1 CPU Reservation Status (p_rsrv)

The active-high **p_rsrv** output signal is used to indicate that a reservation has been established by the execution of a load and reserve (**lbarx**, **lharx**, **lwarx**) instruction. This signal is set following the successful completion of a load and reserve instruction. This signal remains set until the reservation has been cleared (refer to Section 3.5, “Memory Synchronization and Reservation Instructions”). This signal is provided as a status indicator for specialized system applications only.

11.2.16.2 CPU Reservation Clear (p_rsrv_clr)

The active-high **p_rsrv_clr** input signal is used to clear a reservation that has been previously established. External reservation management logic may use this signal to implement reservation management policies which are outside of the scope of the CPU. (Refer to Section 3.5, “Memory Synchronization and Reservation Instructions”). This signal may be asserted independently of any bus transfer.

The **p_rsrv_clr** input signal is not intended for normal use in managing reservations. It is provided for specialized system applications. The normal bus protocol is used to manage reservations using external reservation logic in systems with multiple coherent bus masters, using the transfer type and transfer response signals. In single coherent master systems, no external logic is required, and the internal reservation flag is sufficient to support multi-tasking applications.

The **p_d_xfail_b** signal is provided to indicate success/failure of a **stbcx.**, **sthcx.**, or **stwcx.** instruction as part of bus transfer termination using the XFAIL **p_d_hresp[2:0]** encoding.

11.2.17 Miscellaneous Processor Signals

The following paragraph describes several miscellaneous processor signals.

11.2.17.1 CPU ID (p_cpuid[0:7])

The active-high **p_cpuid[0:7]** input signals are used to provide an identity for a particular processor. These inputs are reflected in the processor ID register (Section 2.4.2, “Processor ID Register (PIR)”) following reset. These inputs are intended to remain in a static condition and are not internally synchronized.

11.2.17.2 PID0 outputs (p_pid0[0:7])

The active-high **p_pid0[0:7]** output signals are used to provide the current process ID in the process ID register 0 (PID0). These outputs correspond to the low order eight bits of PID0.

11.2.17.3 PID0 Update (p_pid0_updt)

The active-high **p_pid0_updt** signal is used to indicate that the process ID register 0 (PID0) is being updated by a **mtspr** instruction. This output asserts during the clock cycle the **p_pid0[0:7]** outputs are changing.

11.2.17.4 System Version (p_sysvers[0:31])

The active-high **p_sysvers[0:31]** input signals are used to provide a version number for the particular system incorporating a e200 CPU. These inputs are reflected in the system version register (Section 2.4.4, “System Version Register (SVR)”). These inputs are intended to remain in a static condition and are not internally synchronized.

11.2.17.5 Processor Version (p_pvrin[16:31])

The active-high **p_pvrin[16:31]** input signals are used to provide a portion of the version number for a particular e200 CPU. These inputs are reflected in the processor version register (Section 2.4.3, “Processor Version Register (PVR)”). These inputs are intended to remain in a static condition and are not internally synchronized.

11.2.17.6 HID1 System Control (p_hid1_sysctl[0:7])

The active-high **p_hid1_sysctl[0:7]** output signals are used to provide a set of control output signals external to the CPU via values written to the HID1 special purpose register. These outputs change state following the rising edge of **m_clk**, and may need synchronization depending on actual use. See Section 2.4.12, "Hardware Implementation Dependent Register 1 (HID1)."

11.2.17.7 Debug Event Outputs (p_devnt_out[0:7])

The active-high **p_devnt_out[0:7]** output signals are used to provide a single-clock pulse based on the values written to the DEVNT field of the DEVENT debug register. These outputs correspond to the low order eight bits of DEVENT. Note that **p_devnt_out[0]** corresponds to the low-order bit, not the MSB of the DEVNT field.

11.2.18 Processor State Signals

The following subsections describe processor internal state signals.

11.2.18.1 Processor Mode (p_mode[0:3])

These signals indicate the global processor execution status. The timing is synchronous with **m_clk**. Table 11-18 shows **p_mode[0:3]** encoding.

Table 11-17. Processor Mode Encoding

p_mode[0:3]				Internal Processor Mode
0	0	0	0	Execution Stalled
0	0	0	1	Execute Exception
0	0	1	0	Instruction Squashed
0	0	1	1	Normal Processing
0	1	0	0	Processor in Halted state
0	1	0	1	Processor in Stopped state
0	1	1	0	Processor in Debug mode[1]
0	1	1	1	Reserved
1	0	0	0	Processor in Waiting state

[1] As reflected on the **cpu_dbgack** internal state signal

11.2.18.2 Processor Execution Pipeline Status (p_pstat_pipe0[0:5], p_pstat_pipe1[0:5])

These signals indicate the internal execution pipeline status. The timing is synchronous with the **m_clk**, so the indicated status may not apply to a current bus transfer. Pipe0 corresponds to the oldest instruction in the pipeline, pipe1 to the next to oldest instruction. Table 11-18 shows **p_pstat_pipe{0,1}[0:5]** encodings.

Table 11-18. Processor Execution PIpeline Status Encoding[1]

p_pstat_pipe{0,1}[0:5]						Processor PIpeline Status
0	0	0	0	s	m	Complete Instruction[2,3]
0	0	0	1	0	0	Complete **lmw**, **stmw, e_lmw, e_stmw, e_lmvgprw, e_stmvgprw, e_lmvsprw, e_stmvsprw, e_lmv[c,d,mc,]srrw, e_stmv[c,d,mc,]srrw**
0	0	0	1	0	1	Complete **e_lmw**, or **e_stmw**
0	0	1	0	0	0	Complete **isync**
0	0	1	0	1	1	Complete **se_isync**
0	0	1	1	0	m	Complete **lbarx**, **lharx**, **lwarx**, **stbcx.**, **sthcx.**, or **stwcx.**[4]
0	1	0	0	0	m	Complete **evsel** with condition false for both elements
0	1	0	1	0	m	Complete **evsel** with condition false for high element and true for low element
0	1	1	0	0	m	Complete **evsel** with condition true for high element and false for low element
0	1	1	1	0	m	Complete **evsel** with condition true for both elements
1	0	0	0	0	0	Complete Branch Instruction **bc**, **bcl**, **bca**, **bcla, b, bl, ba, bla** resolved as not taken
1	0	0	0	0	1	Complete Branch Instruction **e_bc**, **e_bcl**, **e_b, e_bl** resolved as not taken
1	0	0	0	1	1	Complete Branch Instruction **se_bc, se_b, se_bl** resolved as not taken
1	0	0	1	0	0	Complete Branch Instruction **bc**, **bcl**, **bca**, **bcla, b, bl, ba, bla** resolved as taken
1	0	0	1	0	1	Complete Branch Instruction **e_bc**, **e_bcl**, **e_b, e_bl** resolved as taken
1	0	0	1	1	1	Complete Branch Instruction **se_bc**, **se_b, se_bl** resolved as taken
1	0	1	0	0	0	Complete **bclr**, **bclrl, bcctr**, **bcctrl** resolved as not taken
1	0	1	1	0	0	Complete **bclr**, **bclrl, bcctr**, **bcctrl** resolved as taken
1	0	1	1	1	1	Complete **se_blr**, **se_blrl, se_bctr**, **se_bctrl** (always taken)
1	1	0	0	0	m	Complete **isel** with condition false
1	1	0	1	0	m	Complete **isel** with condition true
1	1	1	0	x	x	No instruction completed
1	1	1	1	0	0	Complete **rfi**, **rfci**, **rfdi**, or **rfmci**
1	1	1	1	1	1	Complete **se_rfi**, **se_rfci**, **se_rfdi**, or **se_rfmci**

---

[1] All encodings which do not appear in the table are reserved

[2] Except **rfi, rfci, rfdi, rfmci, lmw, stmw, lbarx**, **lharx**, **lwarx**, **stbcx.**, **sthcx.**, **stwcx., isync, isel, se_rfi, se_rfci, se_rfdi, se_rfmci, e_lmw, e_stmw, se_isel,** and Change of Flow Instructions

[3] s - instruction size, 0=32-bit, 1=16-bit.
m - 0 for Power ISA page, 1 for VLE page

[4] m - 0 for Power ISA page, 1 for VLE page

11.2.18.3 Branch Prediction Status (p_brstat[0:1])

These signals indicate the status of a branch prediction prefetch. Branch prediction prefetches are performed for Branch Target Buffer hits with predict taken status to accelerate branches. The timing is synchronous with the **m_clk**, so the indicated status may not apply to a current bus transfer.

Table 11-19 shows **p_brstat[0:1]** encoding.

Table 11-19. Branch Prediction Status Encoding

p_brstat[0:1]		Branch Prediction Status
0	x	Default (no branch predicted taken prefetch)
1	0	Branch predicted taken prefetch resolved as not taken
1	1	Branch predicted taken prefetch resolved as taken

11.2.18.4 Processor Exception Enable MSR Values (p_msr_EE, p_msr_CE, p_msr_DE, p_msr_ME)

These active-high output signals reflect the state of the corresponding MSR[EE,CE,DE,ME] bits. They may be used by external system logic to determine the set of enabled exceptions. These signals change state on execution of a **mtmsr**, **rfi**, **rfci**, **rfdi**, **rfmci**, **se_rfi**, **se_rfci**, **se_rfdi**, **se_rfmci**, **wrtee**, or **wrteei** instruction, or during exception processing where one or more bits may be cleared during the exception processing sequence.

11.2.18.5 Processor Return from Interrupt (p_rfi, p_rfci, p_rfdi, p_rfmci)

These active-high output signals reflect the state of the processor when executing a return from interrupt class instruction. The signals are asserted for one clock during the execution of the corresponding **rfi**, **rfci**, **rfdi**, **rfmci**, **se_rfi**, **se_rfci**, **se_rfdi**, or **se_rfmci** instruction. They may be used by external system logic to determine the execution state of one or more nested or unnested interrupt exception handlers. They may also be used to provide hardware assist to external interrupt controllers or priority elevation mechanisms. In conjunction with the interrupt acknowledge and exception enable outputs, an external state machine may track the entry and exit status of handlers for various classes and priorities of interrupts.

11.2.18.6 Processor Machine Check (p_mcp_out)

The active-high **p_mcp_out** output signal is asserted by the processor when a machine check condition has caused an “Async Mchk” or “Error Report” type syndrome bit to be set in the machine check syndrome register. Refer to Section 2.4.7, “Machine Check Syndrome Register (MCSR).”

11.2.19 Power Management Control Signals

This section discusses the signals that the external control logic provides for power management or other control functions.

11.2.19.1 Wait, Halt, Stop Signals

The following signals request or indicate that the processor enters either wait, halt, or stop state.

- Processor Waiting (p_waiting)—active-high output signal indicates that the processor entered the waiting state (Section 12.1.2, “Waiting State”).
- Processor Halt Request (p_halt)—active-high input signal requests that the processor enters the halted state (Section 12.1.3, “Halted State”).
- Processor Halted (p_halted)—active-high **p_halted** output signal indicates that the processor has entered the halted state (Section 12.1.3, “Halted State”).
- Processor Stop Request (p_stop)—active-high input signal requests that the processor enters the stopped state (Section 12.1.4, “Stopped State”).
- Processor Stopped (p_stopped)—active-high output signal indicates that the processor entered the stopped state (Section 12.1.4, “Stopped State”).

11.2.19.2 Low-Power Mode Signals (p_doze, p_nap, p_sleep)

The active-high **p_doze**, **p_nap**, and **p_sleep** output signals are asserted by the processor to reflect the settings of the HID0[DOZE], HID0[NAP], and HID0[SLEEP] control bits when MSR[WE] is set.

These outputs may assert for one or more clock cycles. External logic can detect the asserted edge or level of these signals to determine which low-power mode has been requested and then place the e200 core and peripherals in a low-power consumption state. The **p_wakeup** signal can be monitored to determine when to end the low-power condition.

The e200 core can be placed in a low-power state by forcing the **m_clk** input to a quiescent state and brought out of low-power state by re-enabling **m_clk**. The time base facilities may be separately enabled or disabled using combinations of the timer facility control signals described in Section 11.2.15, “Timer Facility Signals.”

11.2.19.3 Wakeup (p_wakeup)

The active-high **p_wakeup** output signal should be used by external logic to remove the e200 core and system logic from a low-power state. It also is used to indicate to the system clock controller that the **m_clk** input should be re-enabled for debug purposes. This signal is asynchronous to the system clock and should be synchronized to the system clock domain to avoid hazards.

p_wakeup asserts whenever one of the following occurs:

- A valid pending interrupt is detected by the core
- A request to enter debug mode is made by setting the DR bit in the OnCE control register (OCR) or via the assertion of the **jd_de_b** or **p_ude** input signals.
- The processor is in a debug session and the **jd_debug_b** output is asserted

- A request to enable the **m_clk** input has been made by setting the WKUP bit in the OnCE control register
- The **p_nmi_b** input is asserted or the MCSR[NMI] syndrome bit is set.

p_wakeup (or other system state) should be monitored to determine when to release the processor (and system if applicable) from a low-power state.

11.2.20 Performance Monitor Signals

The performance monitor unit uses the following interface signals:

- Performance Monitor Event (p_pm_event)
  - — Active-high input signal
  - — Used to signal a performance monitor counted event (described in Section 8.7, "Event Selection")

NOTE

> The e200 core does not internally synchronize this signal. Therefore, it must meet setup and hold time constraints relative to **m_clk** when the e200 core clock is running. This signal is both level and transition sensitive.

- Performance Monitor Counter 0 Overflow state (p_pmc0_ov)
  - — Active-high, output signal
  - — Used to reflect the state of the performance monitor counter 0 OV bit (PMC0[OV]) (described in Section 8.3.9, "Performance Monitor Counter Registers (PMC0–PMC3)")
- Performance Monitor Counter 1 Overflow state (p_pmc1_ov)
  - — Active-high output signal
  - — Used to reflect the state of the performance monitor counter 1 OV bit (PMC1[OV]) (described in Section 8.3.9, "Performance Monitor Counter Registers (PMC0–PMC3)")
- Performance Monitor Counter 2 Overflow state (p_pmc2_ov)
  - — The active-high output signal
  - — Used to reflect the state of the performance monitor counter 2 OV bit (PMC2[OV]) (described in Section 8.3.9, "Performance Monitor Counter Registers (PMC0–PMC3)")
- Performance Monitor Counter 3 Overflow state (p_pmc3_ov)
  - — The active-high output signal
  - — Used to reflect the state of the performance monitor counter 3 OV bit (PMC3[OV]) (described in Section 8.3.9, "Performance Monitor Counter Registers (PMC0–PMC3)")
- Performance Monitor Counter 3 Qualifier inputs **(p_pmc[0,1,2,3]_qual)**
  - — Active-high input signals
  - — Used to provided additional triggering control means for the respective performance monitor counters (described in Section 8.3.7, "Local Control B Registers (PMLCb0–PMLCb3)")

11.2.21 Debug Event Input Signals

The following interface signals are provided to signal debug events to the e200 core:

- Unconditional Debug Event (p_ude)
- External Debug Event 1 (p_devt1)
- External Debug Event 2 (p_devt2)

The following subsections discuss these signals in greater detail.

11.2.21.1 Unconditional Debug Event (p_ude)

The active-high **p_ude** input signal requests an unconditional debug event. This event is described in detail in Section 13.2.13, "Unconditional Debug Event." Note that this signal is not internally synchronized by the e200 core and therefore must meet setup and hold time constraints relative to **m_clk** when the e200 core clock is running.

This signal is level sensitive. To guarantee that it is recognized, it must be held asserted until acknowledged by software or by assertion of the **jd_debug_b** output when external debug mode is enabled. In addition, only a transition from the negated state to the asserted state of the **p_ude** signal causes an event to occur. However, the level on this signal is used to cause assertion of the **p_wakeup** output.

11.2.21.2 External Debug Event 1 (p_devt1)

The active-high **p_devt1** input signal requests an external debug event. This event is described in detail in Section 13.2.12, "External Debug Event." Note that this signal is not internally synchronized by the e200 core and therefore must meet setup and hold time constraints relative to **m_clk** when the e200 core clock is running.

If the e200 core clock is disabled, this signal is not recognized. In addition, only a transition from the negated state to the asserted state of the **p_devt1** signal causes an event to occur. It is intended to signal e200-related events that are generated while the CPU is active.

11.2.21.3 External Debug Event 2 (p_devt2)

The active-high **p_devt2** input signal requests an external debug event. This event is described in detail in Section 13.2.12, "External Debug Event." Note that this signal is not internally synchronized by the e200 core and therefore must meet setup and hold time constraints relative to **m_clk** when the e200 core clock is running.

If the e200 core clock is disabled, this signal is not recognized. In addition, only a transition from the negated state to the asserted state of the **p_devt2** signal will cause an event to occur. It is intended to signal e200 related events which are generated while the CPU is active.

11.2.21.4 14.2.22 Debug Event Output Signals (p_devnt_out[0:7])

The active-high **p_devnt_out[0:7]** output signals provide a single-clock pulse based on the values written to the DEVNT field of the DEVENT debug register. These outputs correspond to the low order eight bits of DEVENT.

Note that **p_devnt_out[0]** corresponds to the low order bit, not the MSB of the DEVNT field.

11.2.22 Debug/Emulation (Nexus 1/OnCE) Support Signals

Table 11-20 shows the interface signals that to assist with implementing an on-chip emulation capability with a controller external to the e200 core.

Table 11-20. e200 Debug/Emulation Support Signals

Signal	Type	Description
jd_en_once	I	Enable full OnCE operation
jd_debug_b	O	Debug session indicator
jd_de_b	I	Debug request
jd_de_en	O	**DE_b** active high output enable
jd_mclk_on	I	CPU clock is active indicator

11.2.22.1 OnCE Enable (jd_en_once)

The OnCE enable signal **jd_en_once** is used to enable the OnCE controller to allow certain instructions and operations to be executed. Assertion of this signal enables the full OnCE command set and the operation of control signals and OnCE control register functions.

When this signal is disabled, only the bypass, ID, and Enable_OnCE commands are executed by the OnCE unit. All other commands default to a bypass command. When OnCE operation is disabled, the OnCE status register (OSR) is not visible; the OnCE control register (OCR) functions are disabled; and the operation of the **jd_de_b** input is disabled. Secure systems may choose to leave **jd_en_once** negated until a security check has been performed. Other systems should tie this signal asserted to enable full OnCE operation.

The **j_en_once_regsel** signal assists external logic with performing security checks. Refer to Section 11.2.25.8, “Enable Once Register Select (j_en_once_regsel),” for a description of the **j_en_once_regsel** output signal.

The **jd_en_once** input must only change state during the Test-Logic-Reset, Run-Test/Idle, or Update_DR TAP states. A new value takes effect after one additional **j_tclk** cycle of synchronization.

11.2.22.2 Debug Session (jd_debug_b)

The **jd_debug_b** active-low output signal is asserted when the processor first enters into debug mode. It remains asserted for the duration of a debug session.

NOTE

A debug session includes single-step operations (Go + NoExit OnCE commands). That is, **jd_debug_b** remains asserted during OnCE single-step executions.

This signal lets system resources know that access is occurring for debug purposes. This permits freezing or otherwise controlling certain resource side-effects, such as FIFO state change control or control of the side-effects of register or memory accesses.

Refer to Section 13.4.5.3, “e200 OnCE Debug Output (jd_debug_b),” for additional information on this signal.

11.2.22.3 Debug Request (jd_de_b)

This signal is the debug mode request input. This signal is not internally synchronized by the e200 core and thus must meet setup and hold time constraints relative to **j_tclk.** To be recognized, it must be held asserted for a minimum of two **j_tclk** periods, and the **jd_en_once** input must be in the asserted state. **jd_de_b** is synchronized to **m_clk** in the debug module before being sent to the processor (two clocks).

This signal is normally the input from the top-level **DE_b** open-drain bidirectional I/O cell. Refer to Section 13.4.5.2, “OnCE Debug Request/Event (jd_de_b, jd_de_en),” for additional information.

11.2.22.4 DE_b Active High Output Enable (jd_de_en)

This output signal is an active-high enable for the top-level **DE_b** open-drain bidirectional I/O cell. This signal is asserted for three **j_tclk** periods upon processor entry into debug mode. Refer to Section 13.4.5.2, “OnCE Debug Request/Event (jd_de_b, jd_de_en),” for additional information on this signal.

11.2.22.5 Processor Clock On (jd_mclk_on)

This active-high input signal is driven by system level clock control logic to indicate that the processor’s **m_clk** input is active. This signal is synchronized to **j_tclk** and provided as a status bit in the OnCE status register.

11.2.22.6 Watchpoint Events (jd_watchpt[0:26])

The **jd_watchpt[0:29]** active-high output signals are used to indicate that a watchpoint has occurred. Each debug address compare function (IAC1–8, DAC1–2) and debug counter event (DCNT1–2) is capable of triggering a watchpoint output. Refer to Section 13.5, “Watchpoint Support,” for the signal assignments of each watchpoint source.

11.2.23 Debug Lockstep Cross-signaling Signals

The following subsections describe the debug lockstep cross-signaling interface signals, which are used to enable lockstep debug operations. Section 11.3.6, “Debug Lockstep Cross-signaling Operation,” provides examples of operation.

11.2.23.1 Debug Request EDM In (p_dbgrq_edm_in)

The active-high **p_dbgrq_edm_in** input signal indicates that a request to enter a debug halted state has been recognized in another CPU and that debug mode may be entered when the receiving CPU has an internally generated debug request present. The request may have occurred via a **tclk** domain mechanism, such as setting of the OCR[DR] control bit or assertion of the **jd_de_b** input, or by a set bit in the EDBSR0 register when EDBCR0[EDM] is set. This signal is assumed to be synchronized to the **m_clk** clock domain and must meet setup and hold times to **m_clk**. The **p_dbgrq_edm_in** input signal is qualified with the **p_lkstep_en** input and is only used to condition debug mode entry when **p_lkstep_en** is asserted. If **p_lkstep_en** is negated, this input signal is ignored.

11.2.23.2 Debug Go Request In (p_dbg_go_in)

The active-high **p_dbg_go_in** input signal indicates that a request to exit a debug halted state has been recognized in another CPU and that debug mode may be exited when the receiving CPU has an internally generated "GO" request present. A request occurs via the **tclk** clock domain when the Update_DR state is entered and the OCMD "GO" bit is set. This signal is assumed to be synchronized to the **m_clk** clock domain and must meet setup and hold times to **m_clk**. The **p_dbg_go_in** input signal is qualified with the **p_lkstep_en** input and is only used to condition debug mode exit when **p_lkstep_en** is asserted. If **p_lkstep_en** is negated, this input signal is ignored.

11.2.23.3 Debug Request EDM Out (p_dbgrq_edm_out)

The active-high **p_dbgrq_edm_out** output signal indicates that a request to enter a debug halted state has occurred. The request may have occurred via a **tclk** domain mechanism, such as setting of the OCR[DR] control bit or assertion of the **jd_de_b** input, or by a set bit in the EDBSR0 register when EDBCR0[EDM] is set. This signal is synchronized to the **m_clk** clock domain. This signal is reset to 0.

11.2.23.4 Debug Go Request Out (p_dbg_go_out)

The active-high **p_dbg_go_out** output signal indicates that a request to exit a debug halted state has occurred. A request occurs via the **tclk** clock domain when the Update_DR state is entered and the OCMD "GO" bit is set. This signal is synchronized to the **m_clk** clock domain. This signal is reset to 0.

11.2.24 Development Support (Nexus 3) Signals

Table 11-21 lists the signals that assist with implementing a real-time development tool capability with a controller external to the e200 core.

Table 11-21. e200 Development Support (Nexus) Signals

Signal	Type	Description
nex_mcko	O	Nexus Clock Output
nex_rdy_b	O	Nexus Ready Output
nex_evto_b	O	Nexus Event-Out Output
nex_wevto[2:0]	O	Nexus Event-Out Output

Table 11-21. e200 Development Support (Nexus) Signals (continued)

Signal	Type	Description
nex_evti_b	I	Nexus Event-In Input
nex_mdo[n:0]	O	Nexus Message Data Output
nex_mseo_b[1:0]	O	Nexus Message Start/End Output
nex_ext_src_id[0:3]	I	Nexus SRC ID Input

11.2.25 JTAG Support Signals

Table 11-22 lists the primary JTAG interface signals. These signals are usually connected directly to device pins (except for **j_tdo**, which needs tri-state and edge support logic). However, this may not be the case when JTAG TAP controllers are concatenated together.

Table 11-22. JTAG Primary Interface Signals

Signal Name	Type	Description
j_trst_b	I	JTAG test reset
j_tclk	I	JTAG test clock
j_tms	I	JTAG test mode select
j_tdi	I	JTAG test data input
j_tdo	O	Test data out to master controller or pad
j_tdo_en[1]	O	Enables TDO output buffer

[1] j_tdo_en is asserted when the TAP controller is in the shift_dr or shift_ir state.

11.2.25.1 JTAG/OnCE Serial Input (j_tdi)

Data and commands are provided to the OnCE controller through the **j_tdi** pin. Data is latched on the rising edge of the **j_tclk** serial clock. Data is shifted into the OnCE serial port least significant bit (LSB) first.

11.2.25.2 JTAG/OnCE Serial Clock (j_tclk)

The **j_tclk** pin supplies the serial clock to the OnCE control block. The serial clock provides pulses required to shift data and commands into and out of the OnCE serial port. Data is clocked into the OnCE on the rising edge and is clocked out of the OnCE serial port on the rising edge. The debug serial clock frequency must be no greater than 50% of the processor clock frequency.

11.2.25.3 JTAG/OnCE Serial Output (j_tdo)

Serial data is read from the OnCE block through the **j_tdo** pin. Data is always shifted out the OnCE serial port least significant bit (LSB) first. When data is clocked out of the OnCE serial port, **j_tdo** changes on the rising edge of **j_tclk**. The **j_tdo** output signal is always driven.

An external system-level TDO pin may be tri-stateable and should be actively driven in the shift-IR and shift-DR controller states. The **j_tdo_en** signal is supplied to indicate when an external TDO pin should be enabled and is asserted during the shift-IR and shift-DR controller states. In addition, for conformity to the IEEE Std. 1149 standard, the system level pin should change state on the falling edge of TCLK.

11.2.25.4 JTAG/OnCE Test Mode Select (j_tms)

The **j_tms** input is used to cycle through states in the OnCE debug controller. Toggling the **j_tms** pin while clocking with **j_tclk** controls transitions through the TAP state controller.

11.2.25.5 JTAG/OnCE Test Reset (j_trst_b)

The **j_trst_b** input is used to externally reset the OnCE controller by placing it in the Test-Logic-Reset state.

Table 11-23 lists additional signals that may be used to support external JTAG data registers using the e200 TAP controller.

Table 11-23. JTAG Signals Used to Support External Registers

Signal Name	Type	Description
j_tst_log_rst	O	Indicates the TAP controller is in the Test-Logic-Reset state
j_rti	O	JTAG controller run-test/idle state
j_capture_ir	O	Indicates the TAP controller is in the capture IR state
j_shift_ir	O	Indicates the TAP controller is in shift IR state
j_update_ir	O	Indicates the TAP controller is in update IR state
j_capture_dr	O	Indicates the TAP controller is in the capture DR state
j_shift_dr	O	Indicates the TAP controller is in shift DR state
j_update_gp_reg	O	Updates JTAG controller general-purpose data register
j_gp_regsel[0:9]	O	General-purpose external JTAG register select
j_en_once_regsel	O	External Enable OnCE register select
j_key_in	I	Serial data from external key logic
j_nexus_regsel	O	External Nexus register select
j_lsrl_regsel	O	External LSRL register select
j_serial_data	I	Serial data from external JTAG register(s)

11.2.25.6 TAP Controller State Indicator Signals

The following signals indicate that the TAP controller is in a state as described:

- **j_tst_log_rst**—the Test-Logic-Reset state
- **j_rti**—the Run-Test/Idle state
- **j_capture_ir**—the Capture_IR state

- **j_shift_ir**—the Shift_IR state
- **j_update_ir**—the Update_IR state
- **j_capture_drr**—the Capture_DR state
- **j_shift_dr**—the Shift_DR state
- **j_update_gp_reg**—the Update_DR state

  This signal also indicates that the R/W bit in the OnCE command register is low (write command). The **j_gp_regsel[0:9]** signals should be monitored to see which register, if any, needs to be updated.

11.2.25.7 Register Select (j_gp_regsel)

The outputs shown in Table 11-24 are a decode of the REGSEL[0–6] field in the OnCE command register (OCMD). They are used to specify which external general-purpose JTAG register to access via the e200 TAP controller.

Table 11-24. JTAG General Purpose Register Select Decoding

Signal Name	Type	Description
j_gp_regsel[0]	O	REGSEL[0–6] = 0x70
j_gp_regsel[1]	O	REGSEL[0–6] = 0x71
j_gp_regsel[2]	O	REGSEL[0–6] = 0x72
j_gp_regsel[3]	O	REGSEL[0–6] = 0x73
j_gp_regsel[4]	O	REGSEL[0–6] = 0x74
j_gp_regsel[5]	O	REGSEL[0–6] = 0x75
j_gp_regsel[6]	O	REGSEL[0–6] = 0x76
j_gp_regsel[7]	O	REGSEL[0–6] = 0x77
j_gp_regsel[8]	O	REGSEL[0–6] = 0x78
j_gp_regsel[9]	O	REGSEL[0–6] = 0x79

11.2.25.8 Enable Once Register Select (j_en_once_regsel)

The **j_en_once_regsel** output is asserted when a decode of the REGSEL[0–6] field in the OnCE command register (OCMD) indicates an external Enable_OnCE register is selected (0b1111110 encoding) for access via the e200 TAP controller. This control signal may be used by external security logic to assist in controlling the **jd_enable_once** input signal. The external Enable_OnCE register should be muxed onto the **j_serial_data** input (Refer to Section 11.2.25.11, “Serial Data (j_serial_data)”). During the Shift_DR state, **j_serial_data** is supplied to the **j_tdo** output.

11.2.25.9 External Nexus Register Select (j_nexus_regsel)

The **j_nexus_regsel** output is asserted when a decode of the REGSEL[0–6] field in the OnCE command register (OCMD) indicates an external Nexus register is selected (0b1111100 encoding) for access via the e200 TAP controller.

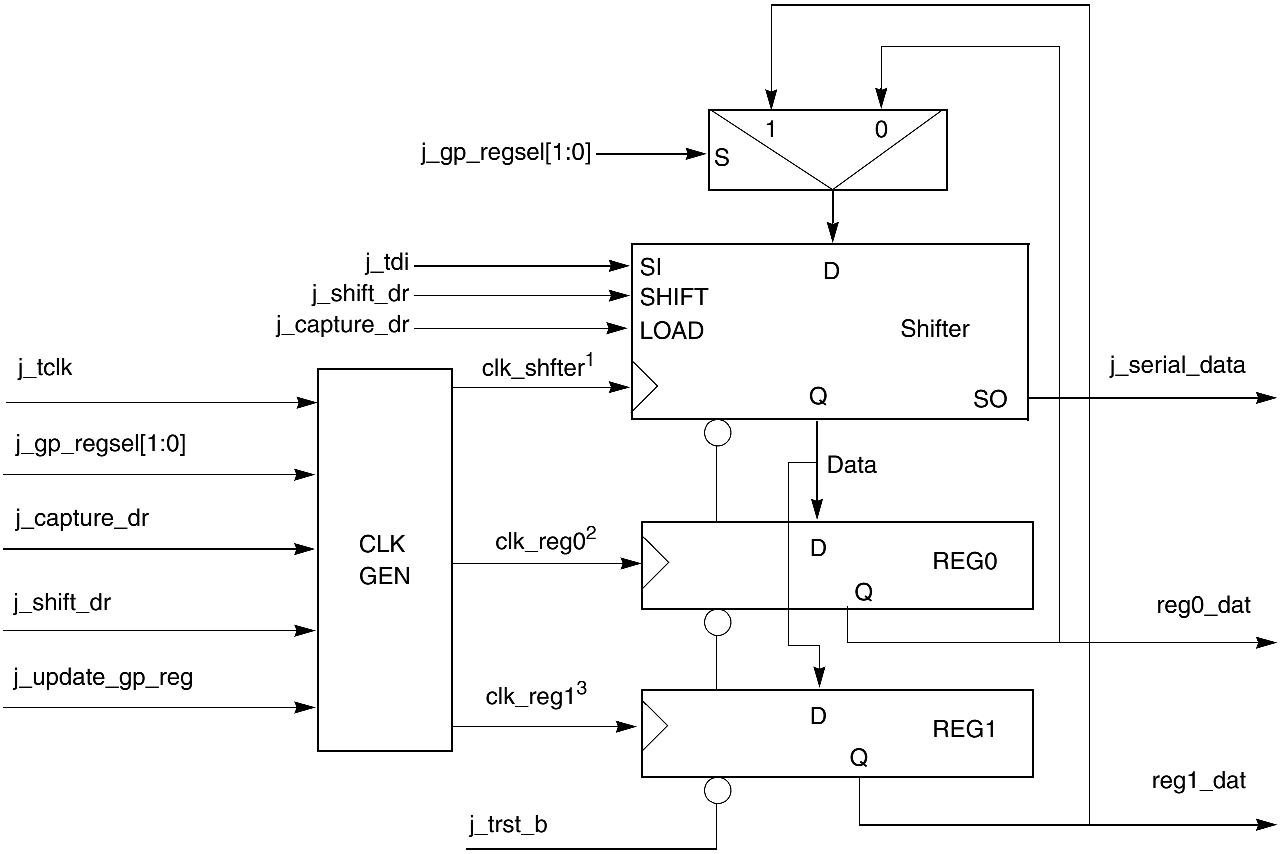
11.2.25.10 External LSRL Register Select (j_lsrl_regsel)

The **j_lsrl_regsel** output is asserted when a decode of the REGSEL[0–6] field in the OnCE command register (OCMD) indicates an external LSRL register is selected (0b1111101 encoding) for access via the e200 TAP controller.

11.2.25.11 Serial Data (j_serial_data)

This input signal receives serial data from external JTAG registers. All external registers share one serial output back to the core; therefore, it must be muxed using the **j_gp_regsel[0:9]**, **j_lsrl_regsel**, and **j_en_once_regsel** signals. The data is internally routed to **j_tdo**.

Figure 11-3 shows one example of how an external JTAG register set (2) can be designed using the inputs and outputs provided and by the JTAG primary inputs themselves. The main components are a clock generation unit, a JTAG shifter (load, shift, hold, clr), the registers (load, hold, clr), and an input mux to the shifter for the serial output back to the e200 core. The shifter and the registers may be as wide as the application warrants [0:x]. The length determines the number of states the TAP controller is held in Shift_DR (x+1).



NOTES:

1. clk_shfter = j_tclk & (j_shift_dr | j_capture_dr)
2. clk_reg0 = j_tclk & j_update_gp_reg & j_gp_regsel[0]
3. clk_reg1 = j_tclk & j_update_gp_reg & j_gp_regsel[1]

Figure 11-3. Example External JTAG Register Design

11.2.25.12 Key Data In (j_key_in)

This input signal receives serial data from logic to indicate a key or other value to be scanned out in the Shift_IR state when the current value in the IR is the Enable_OnCE instruction. This input is provided to assist in implementing security logic outside of the e200z760n3 which conditionally asserts **jd_en_once**. During the Shift_IR state, when **jd_en_once** is negated, this input is sampled on the rising edge of **j_tclk**, and after a two clock delay the data is internally routed to **j_tdo**. This allows provision of a key value via the **j_tdo** output following a transition from Capture_IR to Shift_IR. The key value is provided via the **j_key_in** input.

11.2.26 JTAG ID Signals

Table 11-25 shows the JTAG ID register unique to Freescale as specified by IEEE 1149.1 JTAG. Note that bit 31 is the MSB of this register.

Table 11-25. JTAG Register ID Fields

Bit Field	Type	Description	Value
[31–28]	Variable	Version Number	Variable
[27–22]	Fixed	Design Center Number (e200)	0b011111
[21–12]	Variable	Sequence Number	Variable
[11–1]	Fixed	Freescale Manufacturer ID	0b00000001110
0	Fixed	JTAG ID Register Identification Bit	0b1

The e200 core shifts out a 1 as the first bit on **j_tdo** if the Shift_DR state is entered directly from the test-logic-reset state. This is per the JTAG specification and informs any JTAG controller that an ID register exists on the part. The e200 JTAG ID register is accessed by writing the OCMR (OnCE command register) with the value 7’h02 in the REGSEL[0–6] field.

The JTAG ID bit, manufacturer ID field, and design center number are fixed by the JTAG consortium and/or Freescale. The version numbers and the two most significant bits (MSBs) of the sequence number are variable and brought out to external ports. The lower eight bits of the sequence number are variable and strapped internally to track variations in processor deliverables.

Table 11-26 shows the inputs to the JTAG ID register that are input ports on the e200 core. These bits are provided for a customer to track revisions of a device using the e200 core.

Table 11-26. JTAG ID Register Inputs

Signal Name	Type	Description
j_id_sequence[0:1]	I	JTAG ID register (2 MSBs of sequence field)
j_id_version[0:3]	I	JTAG ID register version field

11.2.26.1 JTAG ID Sequence (j_id_sequence[0:1])

The **j_id_sequence[0:1]** inputs correspond to the two MSBs of the 10-bit sequence number in the JTAG ID register. These inputs are normally static. They are provided for the customer for further component variation identification.

11.2.26.2 JTAG ID Sequence (j_id_sequence[2:9])

The **j_id_sequence[2:9]** field is internally strapped to track variations in processor and module deliverables. Each e200 deliverable has a unique sequence number. Additionally, each revision of these modules can be identified by unique sequence numbers.

11.2.26.3 JTAG ID Version (j_id_version[0:3])

The **j_id_version[0:3]** inputs correspond to the 4-bit version number in the JTAG ID register. These inputs are normally static. They are provided to the customer for strapping in order to facilitate easy identification of component variants.

11.3 Timing Diagrams

This section discusses the timing diagrams. It consists of the following subsections:

- Section 11.3.1, "AHB Clock Enable and the Internal HCLK"
- Section 11.3.2, "Processor Instruction/Data Transfers"
- Section 11.3.3, "Memory Synchronization Control Operation"
- Section 11.3.4, "Cache Error Cross-signaling Operation"
- Section 11.3.5, "Cache Coherency Interface Operation"
- Section 11.3.6, "Debug Lockstep Cross-signaling Operation"
- Section 11.3.7, "Power Management"
- Section 11.3.8, "Interrupt Interface"
- Section 11.3.9, "Time Base Interface"
- Section 11.3.10, "JTAG Test Interface"

11.3.1 AHB Clock Enable and the Internal HCLK

The CPU generates an internal HCLK to control AHB signal input sampling and output transitions based on the internal **m_clk** and the **p_[i,d]_ahb_clken** signals. The following diagrams show the relationships of these signals and the resulting HCLK. Note that since no AHB signals are sampled or change state on the falling edge of HCLK, the duty cycle is not an issue.

Figure 11-4 shows an example of a free-running half-speed HCLK relative to **m_clk**.

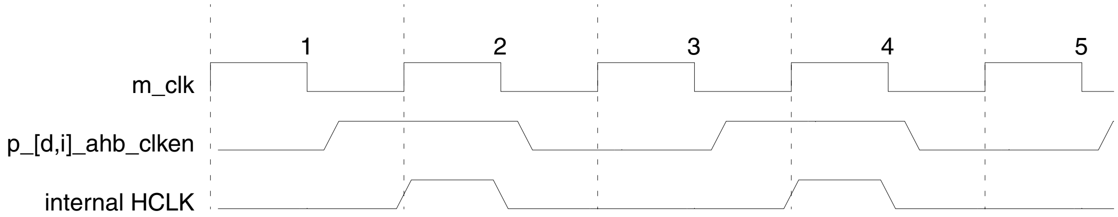


Figure 11-4. AHB Clock Enable Operation—1

Figure 11-5 shows an example of a free-running $\frac{1}{3}$ speed HCLK relative to **m_clk**.

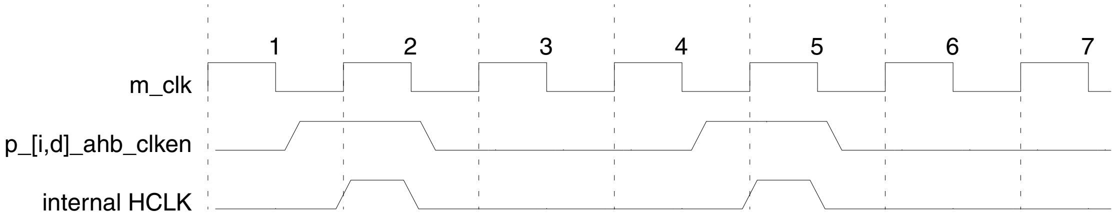


Figure 11-5. AHB Clock Enable Operation—2

Figure 11-6 shows an example of a non-periodic HCLK, used for power reduction, relative to **m_clk**.

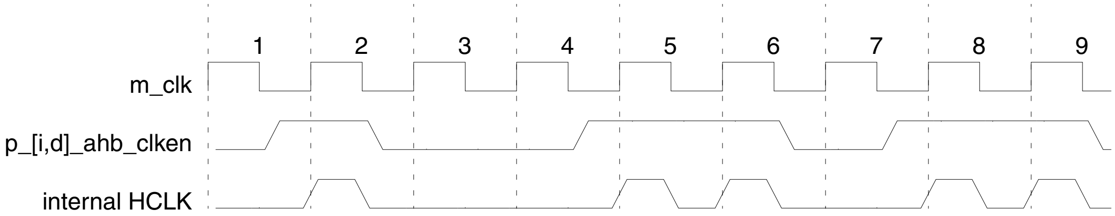


Figure 11-6. AHB Clock Enable Operation—3

11.3.2 Processor Instruction/Data Transfers

Transfer of data between the core and peripherals involves the address bus, data buses, and control and attribute signals. The address and data buses are parallel, non-multiplexed buses, supporting byte, half-word, three byte, word, and double-word transfers. All bus input and output signals are sampled and driven with respect to the rising edge of the **m_clk** signal. The core moves data on the bus by issuing control signals and using a handshake protocol to ensure correct data movement.

The memory interface operates in a pipelined fashion to allow additional access time for memory and peripherals. AHB transfers consist of an address phase which lasts only a single cycle, followed by the data phase which may last for one or more cycles depending on the state of the **p_hready** signal.

Read transfers consist of a request cycle, where address and attributes are driven along with a transfer request, and one or more memory access cycles to perform accesses and return data to the CPU for alignment, sign or zero extension, and forwarding.

Write transfers consist of a request cycle, where address and attributes are driven along with a transfer request, and one or more data drive cycles where write data is driven and external devices accept write data for the access.

Access requests are generated in an overlapped fashion in order to support sustained single cycle transfers. Up to two access requests may be in progress at any one cycle, one access outstanding and a second in the pending request phase.

Access requests are assumed to be accepted as long as there are no accesses in progress, or if an access in progress is terminated during the same cycle a new request is active (**p_hready** asserted). Once an access has been accepted, the BIU is free to change the current request at any time, even if part of a burst transfer.

The local memory control logic is responsible for proper pipelining and latching of all interface signals to initiate memory accesses.

The system hardware can use the **p_hresp[2:0]** signals to signal that the current bus cycle has an error when a fault is detected, using the ERROR response encoding. ERROR assertion requires a two cycle response. In the first cycle of the response, the **p_hresp[2:0]** signals are driven to indicate ERROR and **p_hready** must be negated. During the following cycle, the ERROR response must continue to be driven, and **p_hready** must be asserted. When the core recognizes a bus error condition for an access at the end of the first cycle of the two cycle error response, a subsequent pending access request may be removed by the BIU driving the **p_htrans[2:0]** signals to the IDLE state in the second cycle of the two cycle error response. Not all pending requests will be removed however.

When a bus cycle is terminated with a bus error, the core can enter storage error exception processing immediately following the bus cycle, or it can defer processing the exception.

The instruction prefetch mechanism requests instruction words from the instruction memory unit before it is ready to execute them. If a bus error occurs on an instruction fetch, the core does not take the exception until it attempts to use the instruction. Should an intervening instruction cause a branch, or should a task switch occur, the storage error exception for the unused access does not occur. A bus error termination for any write access or read access that reference data specifically requested by the execution unit causes the core to begin exception processing.

NOTE

In the following diagrams showing AHB operations, note that the HCLK signal is that of the AHB bus, i.e. **m_clk** qualified by **p_[i,d]_ahb_clken**

11.3.2.1 Basic Read Transfer Cycles

During a read transfer, the core receives data from a memory or peripheral device. Figure 11-7 illustrates functional timing for basic read transfers and clock-by-clock descriptions of activity.

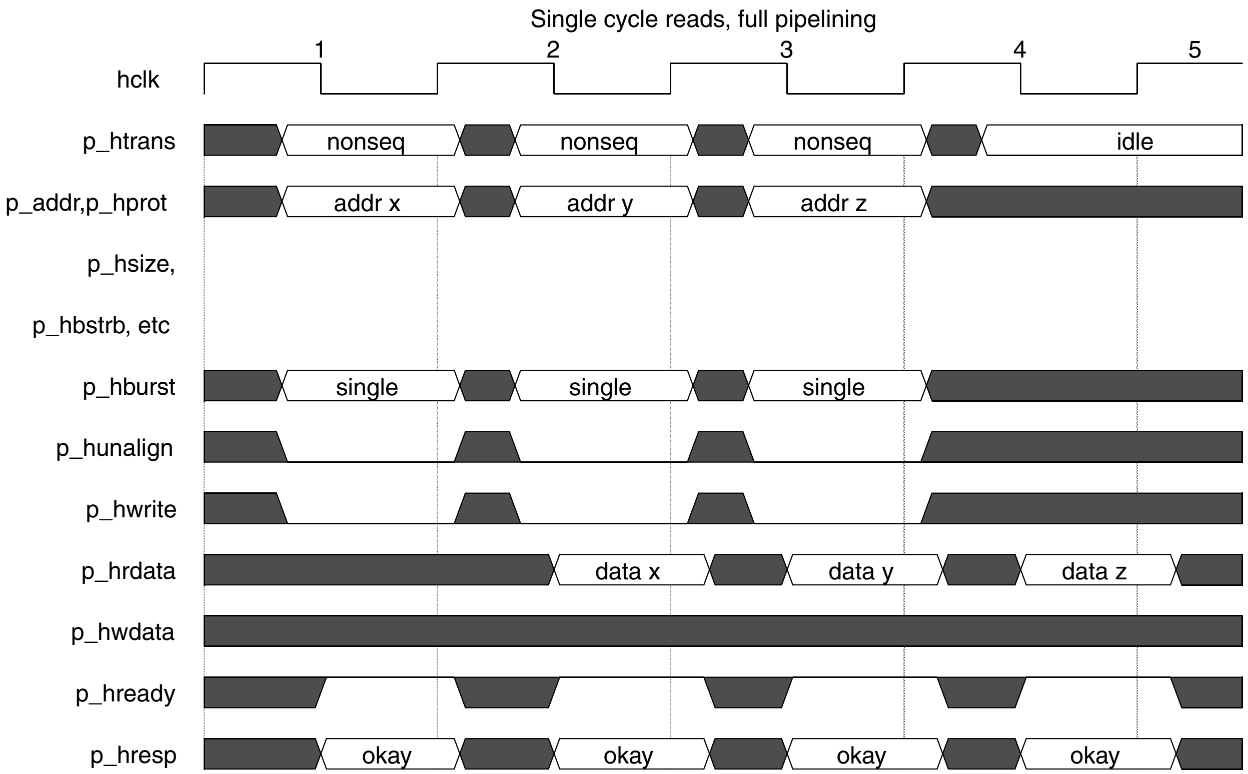


Figure 11-7. Basic Read Transfers

Clock 1 (C1)

The first read transfer starts in clock cycle 1. During C1, the core places valid values on the address bus and transfer attributes. The burst type (**p_hburst[2:0]**), protection control (**p_hprot[5:0]**), and transfer type (**p_htrans[1:0]**) attributes identify the specific access type. The transfer size attributes (**p_hsize[1:0]**) indicates the size of the transfer. The byte strobes (**p_hbstrb[7:0]**) are driven to indicate active byte lanes. The write (**p_hwrite**) signal is driven low for a read cycle.

The core asserts transfer request (**p_htrans** = NONSEQ) during C1 to indicate that a transfer is being requested. Since the bus is currently idle, (0 transfers outstanding), the first read request to addr_x is considered *taken* at the end of C1. The default slave drives a ready/OKAY response for the current idle cycle.

Clock 2 (C2)

During C2, the addr_x memory access takes place using the address and attribute values that were driven during C1 to enable reading of one or more bytes of memory. Read data from the slave device is provided on the **p_hrdata** inputs. The slave device responds by asserting **p_hready** to indicate the cycle is completing and drives an OKAY response.

Another read transfer request is made during C2 to addr_y (**p_htrans** = NONSEQ), and since the access to addr_x is completing, it is considered *taken* at the end of C2.

Clock 3 (C3)

During C3, the addr_y memory access takes place using the address and attribute values that were driven during C2 to enable reading one or more bytes of memory. Read data from the slave device for addr_y is provided on the **p_hrdata** inputs. The slave device responds by asserting **p_hready** to indicate the cycle is completing and drives an OKAY response.

Another read transfer request is made during C3 to addr_z (**p_htrans** = NONSEQ). Because the access to addr_y is completing, it is considered *taken* at the end of C3.

Clock 4 (C4)

During C4, the addr_z memory access takes place using the address and attribute values that were driven during C3 to enable reading one or more bytes of memory. Read data from the slave device for addr_z is provided on the **p_hrdata** inputs. The slave device responds by asserting **p_hready** to indicate that the cycle is completing and drives an OKAY response.

The CPU has no more outstanding requests, so **p_htrans** indicates IDLE. The address and attribute signals are thus undefined.

11.3.2.2 Read Transfer with Wait State

Figure 11-8 shows an example of wait state operation. Signal **p_hready** for the first request (addr$_x$) is not asserted during C2, so a wait state is inserted until **p_hready** is recognized (during C3).

Meanwhile, a subsequent request has been generated by the CPU for addr$_y$ which is not *taken* in C2, because the previous transaction is still outstanding. The address and transfer attributes remain driven in cycle C3 and are taken at the end of C3 because the previous access is completing. Data for addr$_x$ and a ready/OKAY response is driven back by the slave device. In cycle C4, a request for addr$_z$ is made. The request for access to addr$_z$ is taken at the end of C4, and during C5, the data and a ready/OKAY response is provided by the slave device. In cycle C5, no further accesses are requested.

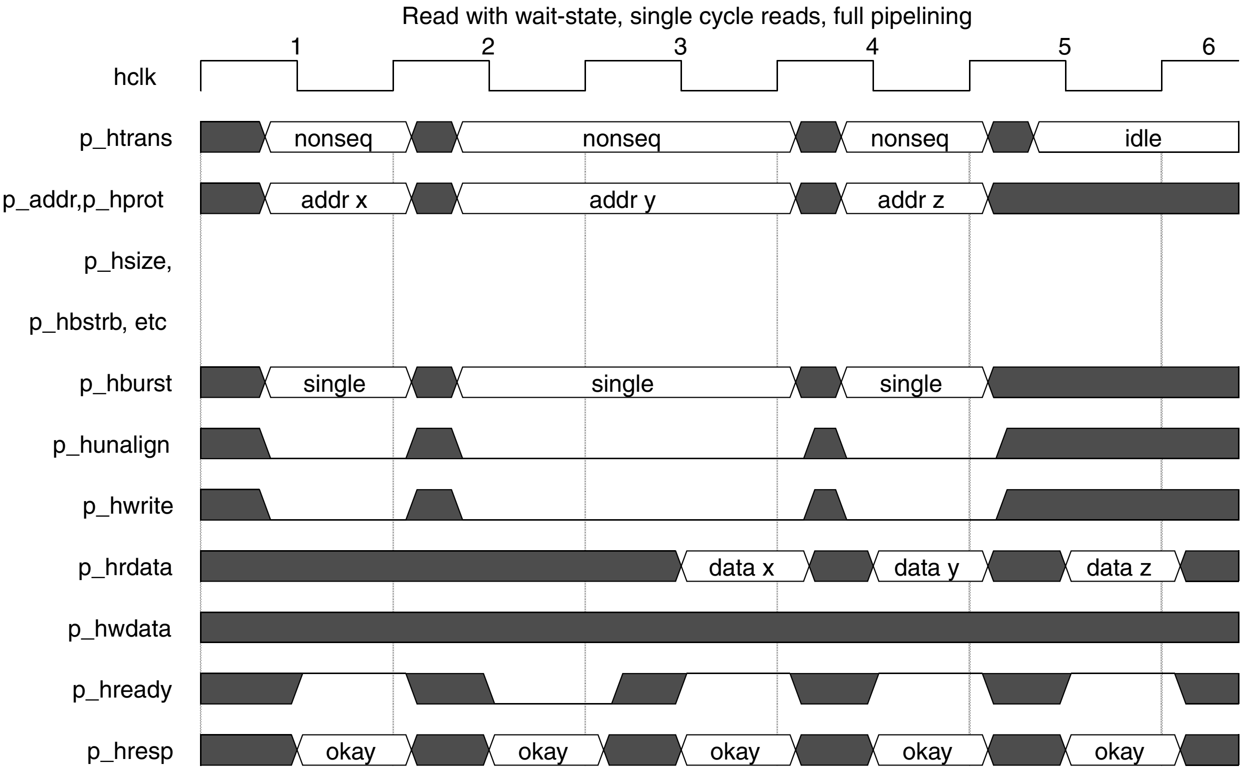


Figure 11-8. Read Transfer with Wait-state

11.3.2.3 Basic Write Transfer Cycles

During a write transfer, the core provides write data to a memory or peripheral device. Figure 11-9 illustrates functional timing for basic write transfers and clock-by-clock descriptions of activity.

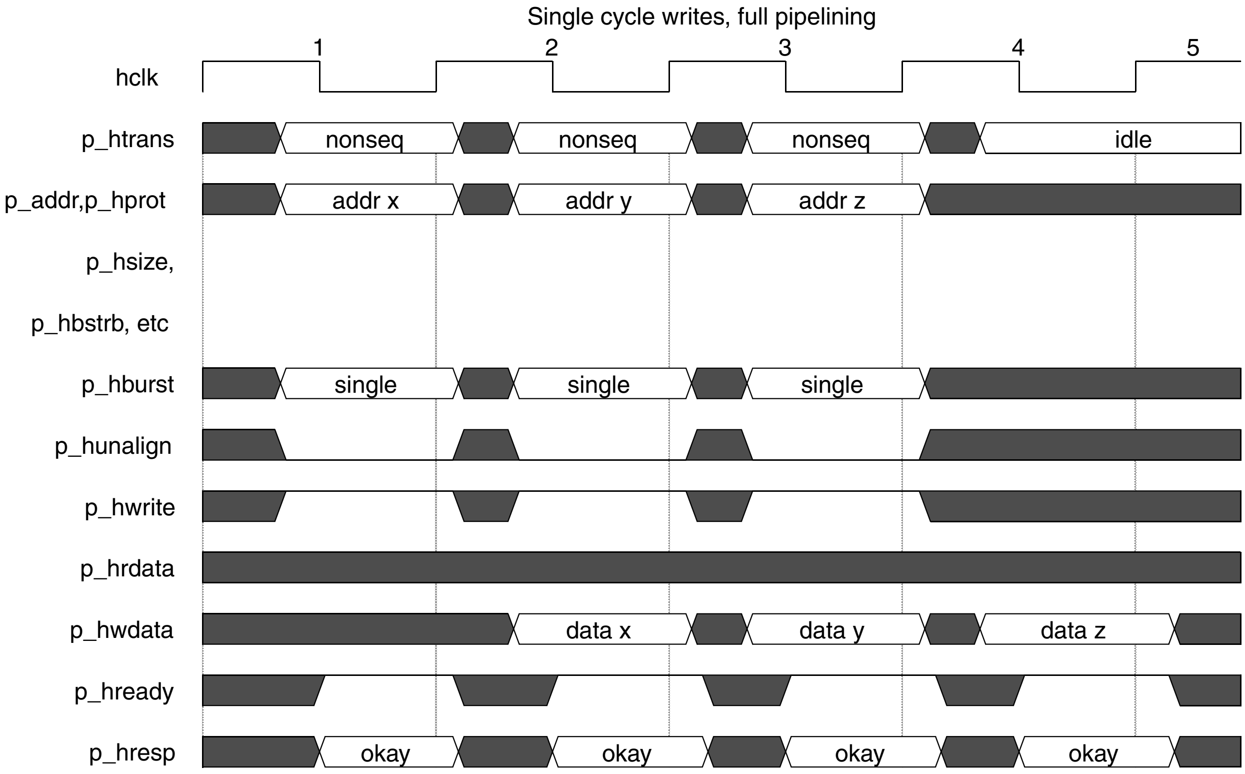


Figure 11-9. Basic Write Transfers

Clock 1 (C1)

The first write transfer starts in clock cycle 1. During C1, the core places valid values on the address bus and transfer attributes. The burst type (**p_hburst[2:0]**), protection control (**p_hprot[5:0]**), and transfer type (**p_htrans[1:0]**) attributes identify the specific access type. The transfer size attributes (**p_hsize[1:0]**) indicate the size of the transfer. The byte strobes (**p_hbstrb[7:0]**) are driven to indicate active byte lanes. The write (**p_hwrite**) signal is driven high for a write cycle.

The core asserts transfer request (**p_htrans** = NONSEQ) during C1 to indicate that a transfer is being requested. Because the bus is currently idle (0 transfers outstanding), the first write request to addr_x is considered *taken* at the end of C1. The default slave drives an ready/OKAY response for the current idle cycle.

Clock 2 (C2)

During C2, the write data for the access is driven, and the addr_x memory access takes place using the address and attribute values which were driven during C1 to enable writing of one or more bytes of memory. The slave device responds by asserting **p_hready** to indicate the cycle is completing and drives an OKAY response.

Another write transfer request is made during C2 to addr$_y$ (**p_htrans** = NONSEQ), and since the access to addr$_x$ is completing, it is considered *taken* at the end of C2.

Clock 3 (C3)

During C3, write data for addr$_y$ is driven, and the addr$_y$ memory access takes place using the address and attribute values which were driven during C2 to enable writing of one or more bytes of memory. The slave device responds by asserting **p_hready** to indicate the cycle is completing and drives an OKAY response.

Another write transfer request is made during C3 to addr$_z$ (**p_htrans** = NONSEQ), and since the access to addr$_y$ is completing, it is considered *taken* at the end of C3.

Clock 4 (C4)

During C4, write data for addr$_z$ is driven, and the addr$_z$ memory access takes place using the address and attribute values that were driven during C3 to enable writing of one or more bytes of memory. The slave device responds by asserting **p_hready** to indicate the cycle is completing and drives an OKAY response.

The CPU has no more outstanding requests, so **p_htrans** indicates IDLE. The address and attribute signals are thus undefined.

11.3.2.4 Write Transfer with Wait States

Figure 11-10 shows an example of a write wait state operation. Signal **p_hready** for the first request (addr_x) is not asserted during C2, so a wait state is inserted until **p_hready** is recognized (during C3).

Meanwhile, a subsequent request has been generated by the CPU for addr_y which is not *taken* in C2, since the previous transaction is still outstanding. The address, transfer attributes, and write data remain driven in cycle C3 and are taken at the end of C3 since a ready/OKAY response is driven back by the slave device for the previous access. In cycle C4, a request for addr_z is made. The request for access to addr_z is taken at the end of C4, and during C5, a ready/OKAY response is provided by the slave device. In cycle C5, no further accesses are requested.

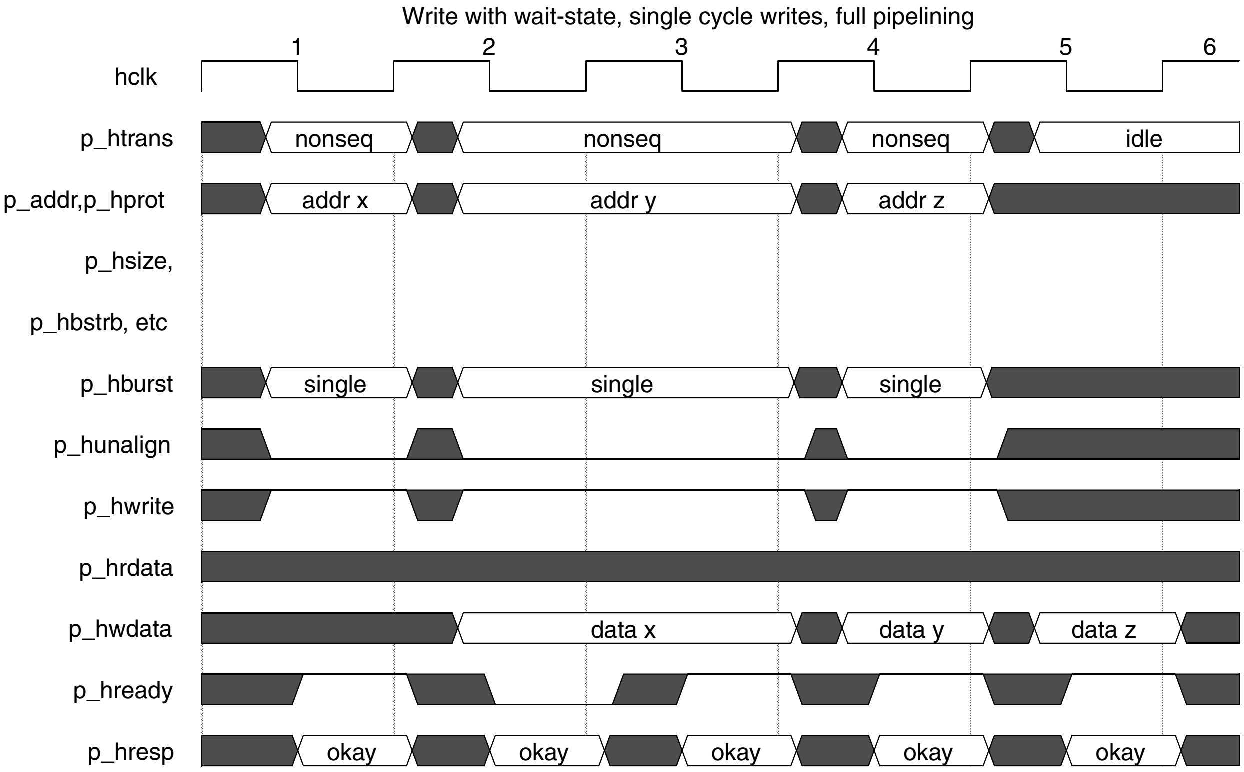


Figure 11-10. Write Transfer with Wait-State

11.3.2.5 Read and Write Transfers

Figure 11-11 shows a sequence of read and write cycles.

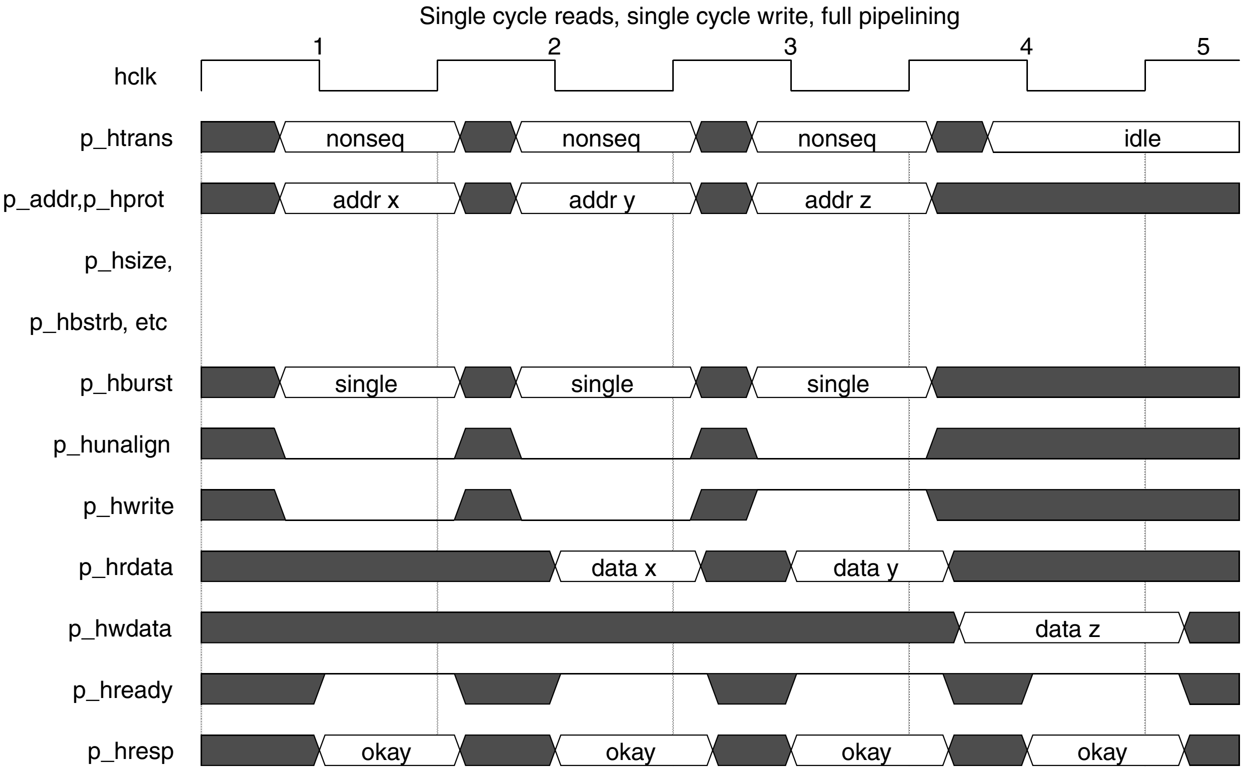


Figure 11-11. Single Cycle Read and Write Transfers—1

The first read request ($addr_x$) is *taken* at the end of cycle C1 since the bus is idle.

The second read request ($addr_y$) is *taken* at the end of C2 since a ready/OKAY response is asserted during C2 for the first read access ($addr_x$). During C3, a request is generated for a write to $addr_y$, which is taken at the end of C3 since the second access is terminating.

Data for the $addr_z$ write cycle is driven in C4, the cycle after the access is *taken*, and a ready/OKAY response is signaled to complete the write cycle to $addr_z$.

Figure 11-12 shows another sequence of read and write cycles. This example shows an interleaved write access between two reads.

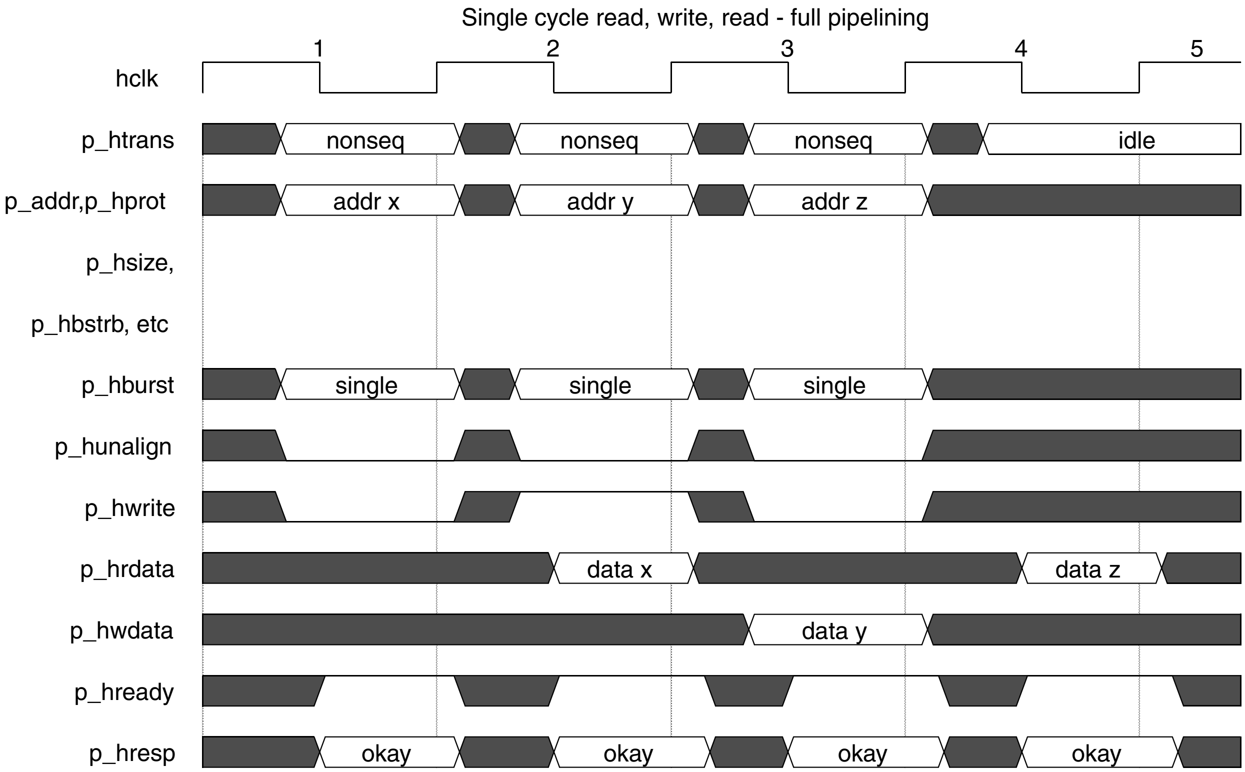


Figure 11-12. Single Cycle Read and Write Transfers—2

The sequence of events is as follows:

1. The first read request ($addr_x$) is *taken* at the end of cycle C1 since the bus is idle.
2. The first write request ($addr_y$) is *taken* at the end of C2 since the first access is terminating ($addr_x$).
3. Data for the $addr_y$ write cycle is driven in C3, the cycle after the access is *taken*. Also during C3, a request is generated for a read to $addr_z$, which is taken at the end of C3 since the write access is terminating.
4. During C4, the $addr_y$ write access is terminated, and no further access is requested

Figure 11-13 shows another sequence of read and write cycles. In this example, reads incur a single wait state.

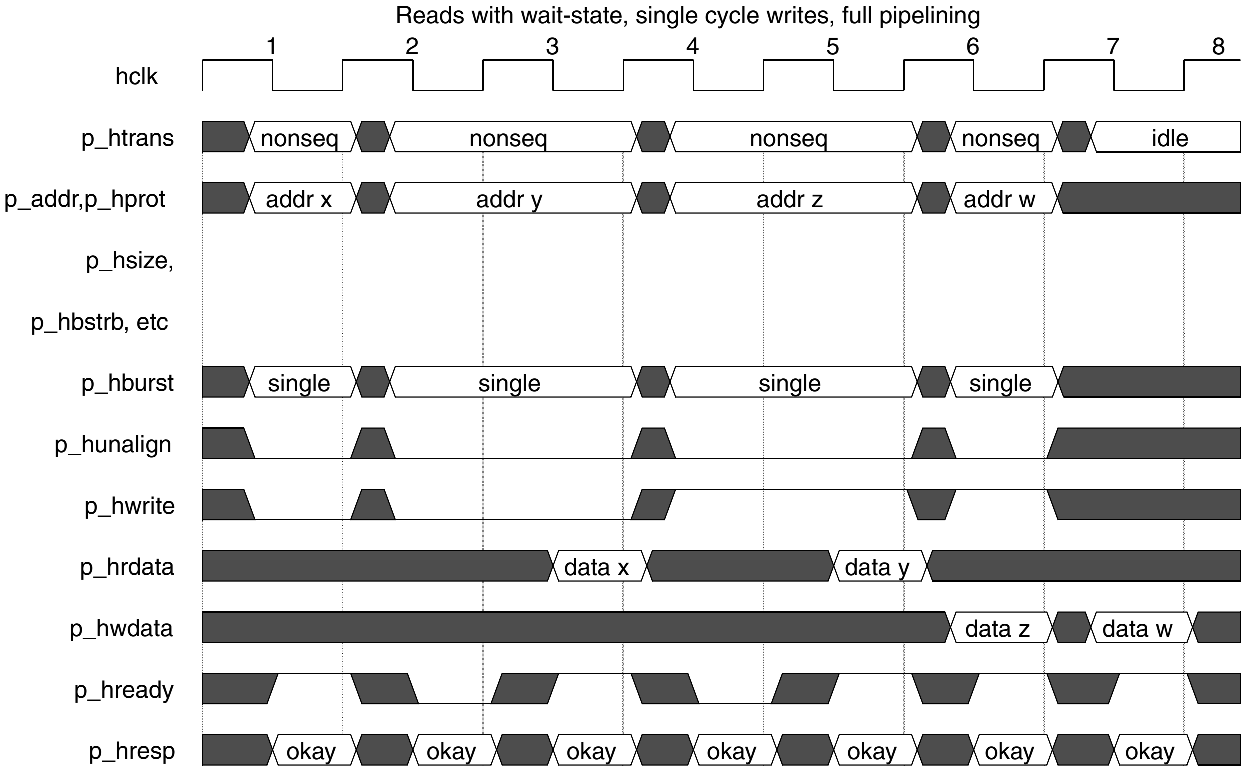


Figure 11-13. Multi-Cycle Read and Write Transfers—1

The sequence of events is as follows:

1. The first read request (addr$_x$) is *taken* at the end of cycle C1 since the bus is idle.
2. The second read request (addr$_y$) is not *taken* at the end of cycle C2 since no ready response is signaled and only one access can be outstanding (addr$_x$). It is taken at the end of C3 once the first read request has signaled a ready/OKAY response.
3. The first write request (addr$_z$) is not taken during C4 since a ready response is not asserted during C4 for the second read access (addr$_y$). During C5, the request for a write to addr$_z$ is taken since the second access is terminating.
4. Data for the addr$_z$ write cycle is driven in C6, the cycle after the access is *taken*.
5. During C6, the addr$_z$ write access is terminated and the addr$_w$ write request is *taken*.
6. During C7, data for the addr$_w$ write access is driven, and a ready/OKAY response is asserted to complete the write cycle to addr$_w$.

Figure 11-14 shows another sequence of read and write cycles. In this example, reads incur a single wait state.

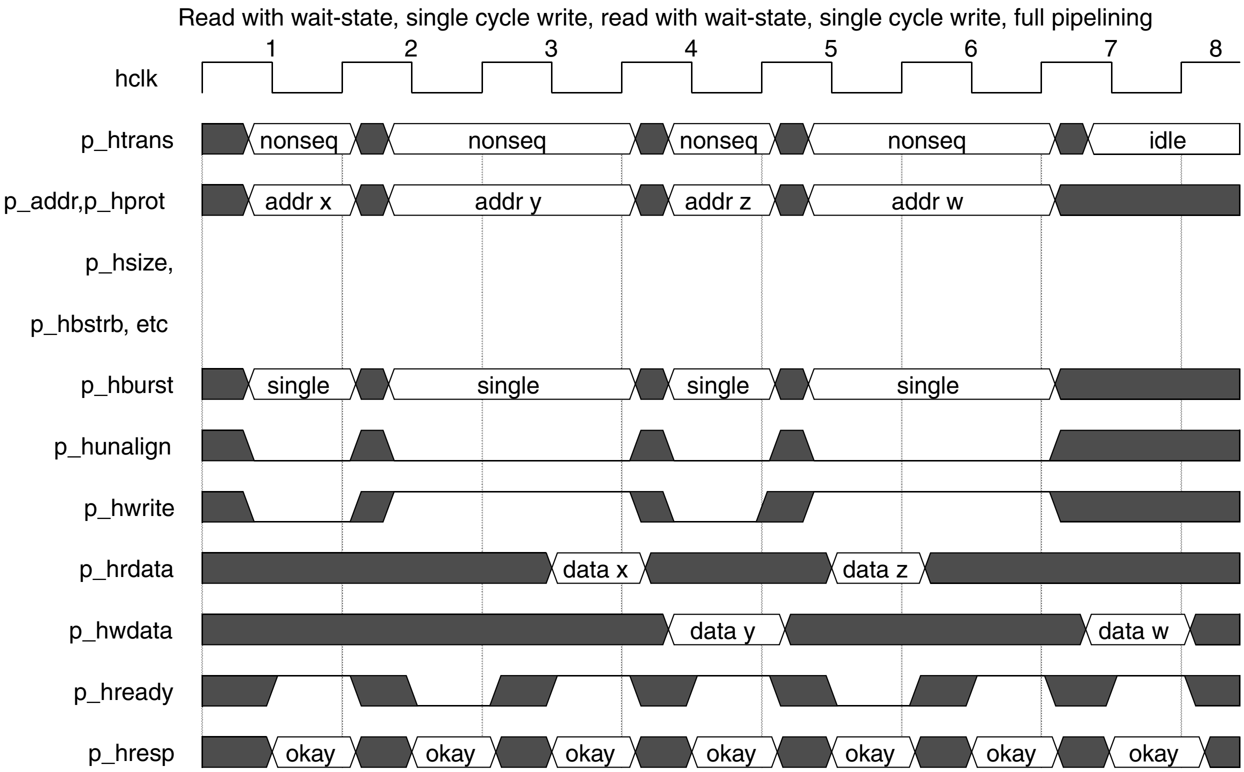


Figure 11-14. Multi-Cycle Read and Write Transfers—2

The sequence of events is as follows:

1. The first read request ($addr_x$) is *taken* at the end of cycle C1 since the bus is idle.
2. The first write request ($addr_y$) is not *taken* at the end of cycle C2 because no ready response is signaled, and only one access can be outstanding ($addr_x$). It is taken at the end of C3 once the first read request has signaled a ready/OKAY response.
3. Data for the $addr_y$ write cycle is driven in C4, the cycle after the access is *taken*.
4. The second read request ($addr_z$) is taken during C4 since the $addr_y$ write is terminating.
5. A second write request ($addr_w$) is not taken at the end of C5 since the second read access is not terminating, thus it continues to drive the address and attributes into cycle C6.
6. During C6, the $addr_z$ read access is terminated and the $addr_w$ write access is taken.
7. In cycle C7, data for the $addr_w$ write access is driven. During C7, a ready/OKAY response is asserted to complete the write cycle to $addr_w$. No further accesses are requested, so **p_htrans** signals IDLE.

11.3.2.6 Misaligned Accesses

Figure 11-15 illustrates functional timing for a misaligned read transfer. The read to addr_x is misaligned across a 64-bit boundary.

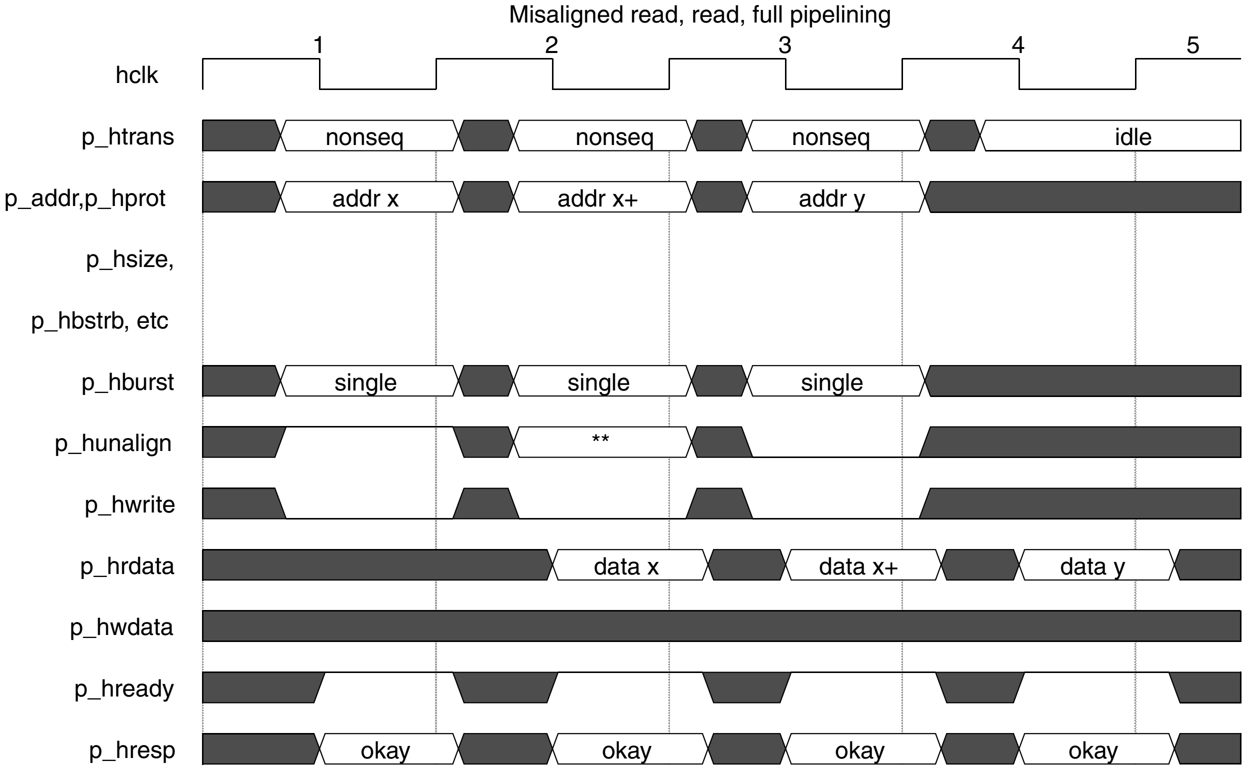


Figure 11-15. Misaligned Read Transfer

The first portion of the misaligned read transfer starts in C1. During C1, the core places valid values on the address bus and transfer attributes. The **p_hwrite** signal is driven low for a read cycle. The transfer size attributes (**p_hsize**) indicate the size of the transfer. Even though the transfer is misaligned, the size value driven corresponds to the size of the entire misaligned data item. **p_hunalign** is driven high to indicate that the access is misaligned. The **p_hbstrb** outputs are asserted to indicate the active byte lanes for the read, which may not correspond to size and low-order address outputs. **p_htrans** is driven to NONSEQ.

During C2, the addr_x memory access takes place using the address and attribute values which were driven during C1 to enable reading of one or more bytes of memory.

The second portion of the misaligned read transfer request is made during C2 to addr_{x+} (which will be aligned to the next higher 64-bit boundary), and since the first portion of the misaligned access is completing, it is *taken* at the end of C2. The p_htrans signals indicate NONSEQ. The size value driven is the size of the remaining bytes of data in the misaligned read, rounded up (for the 3-byte case) to the next higher power-of-2. The **p_hbstrb** signals indicate the active byte lanes. For the second portion of a misaligned transfer, the **p_hunalign** signal is driven high for the 3-byte case (low for all others). The next read access is requested in C3 and **p_htrans** indicates NONSEQ. **p_hunalign** is negated, since this access is aligned.

Figure 11-16 illustrates functional timing for a misaligned write transfer. The write to $addr_x$ is misaligned across a 64-bit boundary.

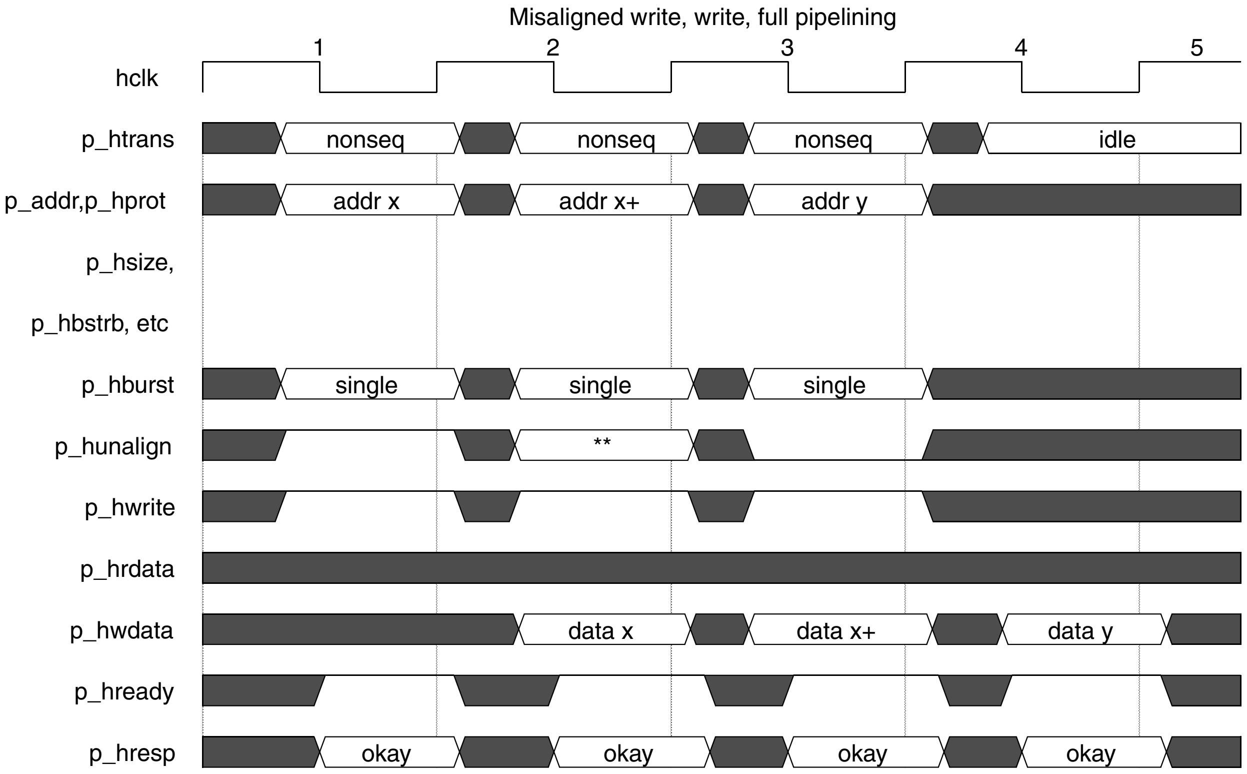


Figure 11-16. Misaligned Write Transfer

The first portion of the misaligned write transfer starts in C1. During C1, the core places valid values on the address bus and transfer attributes. The **p_hwrite** signal is driven high for a write cycle. The transfer size attribute (**p_hsize**) indicates the size of the transfer. Even though the transfer is misaligned, the size value driven corresponds to the size of the entire misaligned data item. **p_hunalign** is driven high to indicate that the access is misaligned. The **p_hbstrb** outputs are asserted to indicate the active byte lanes for the write, which may not correspond to size and low-order address outputs. **p_htrans** is driven to NONSEQ.

During C2, data for $addr_x$ is driven, and the $addr_x$ memory access takes place using the address and attribute values which were driven during C1 to enable writing of one or more bytes of memory.

The second portion of the misaligned write transfer request is made during C2 to $addr_{x+}$ (which will be aligned to the next higher 64-bit boundary), and since the first portion of the misaligned access is completing, it is *taken* at the end of C2. The p_htrans signals indicate NONSEQ. The size value driven is the size of the remaining bytes of data in the misaligned write, rounded up (for the 3-byte case) to the next higher power-of-2. The **p_hbstrb** signals indicate the active byte lanes. For the second portion of a misaligned transfer, the **p_hunalign** signal is driven high for the 3-byte case (low for all others).

The next write access is requested in C3 and **p_htrans** indicates NONSEQ. **p_hunalign** is negated, since this access is aligned.

An example of a misaligned write cycle followed by an aligned read cycle is shown in Figure 11-17. This is similar to the example shown in Figure 11-16.

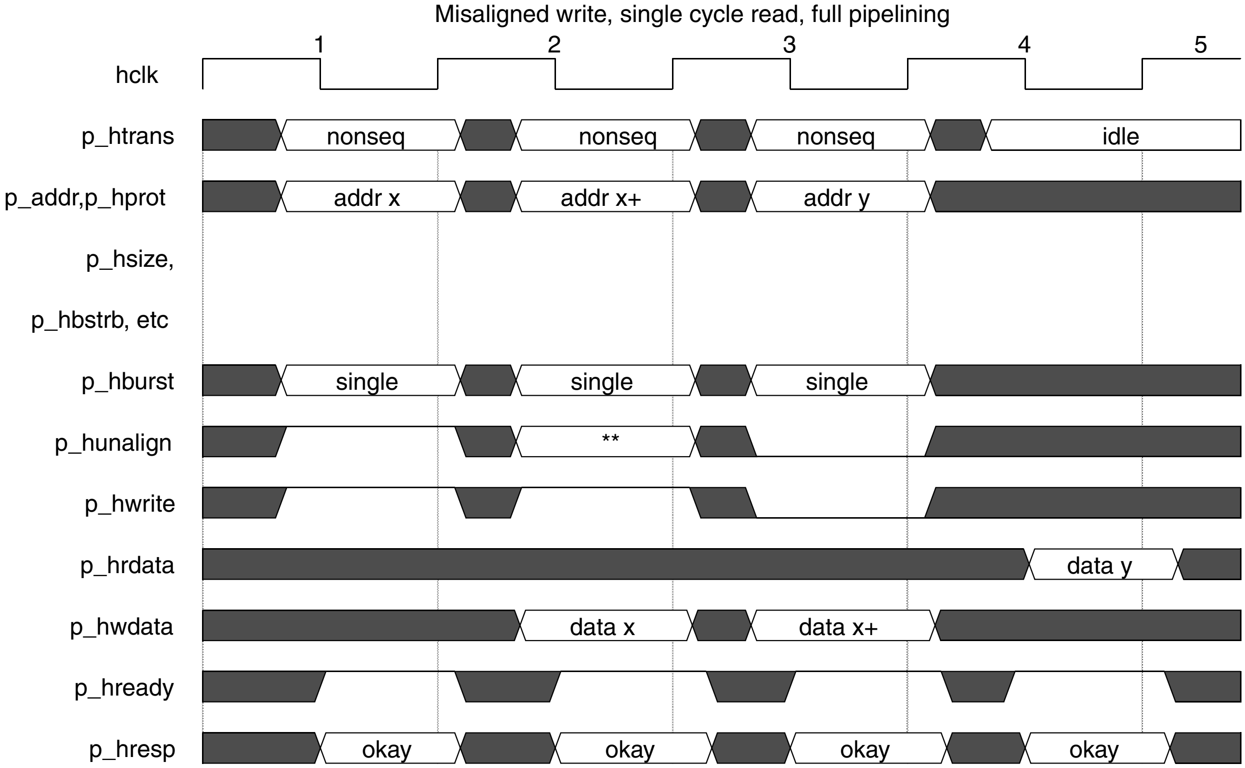


Figure 11-17. Misaligned Write, Single Cycle Read Transfer

11.3.2.7 Burst Accesses

Figure 11-18 illustrates functional timing for a burst read transfer.

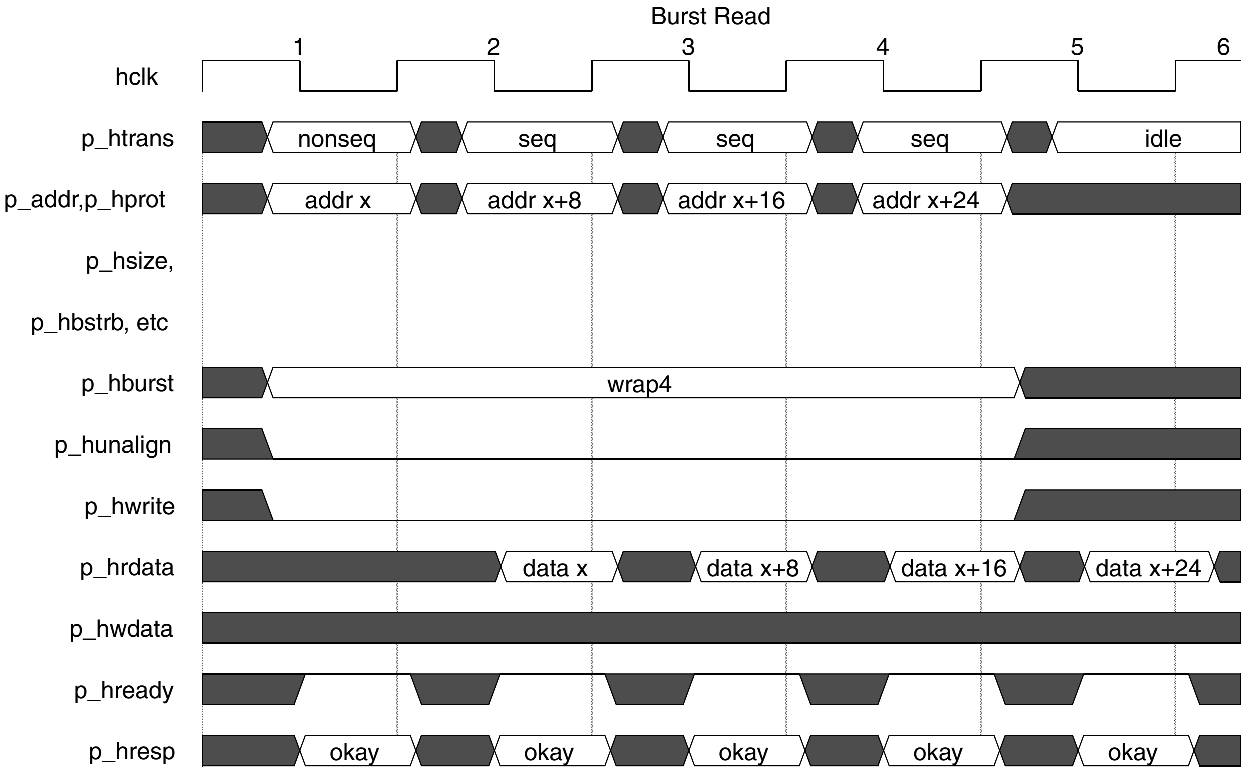


Figure 11-18. Burst Read Transfer

The p_hburst signals indicate WRAP4 for all burst transfers. The **p_hunalign** signal is negated. **p_hsize** indicates 64-bits, and all eight **p_hbstrb** signals are asserted. The burst address is aligned to a 64-bit boundary and wraps around modulo four double words. Note that in this example the **p_htrans** signal indicates IDLE after the last portion of the burst has been taken, but this is not always the case.

NOTE

Bursts may be followed immediately by any type of transfer. No idle cycle is required.

Figure 11-19 illustrates functional timing for a burst read with wait-state transfer.

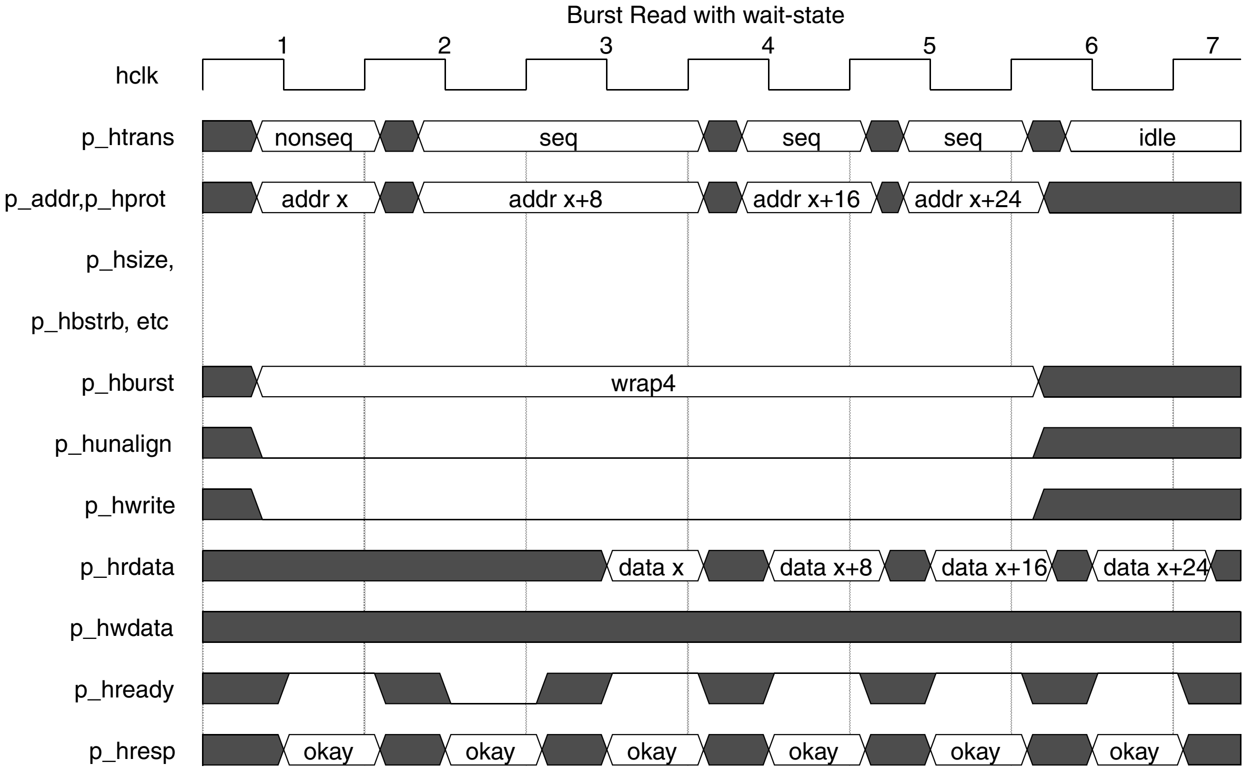


Figure 11-19. Burst Read with Wait-state Transfer

The first cycle of the burst incurs a single wait-state.

Figure 11-20 illustrates functional timing for a burst write transfer.

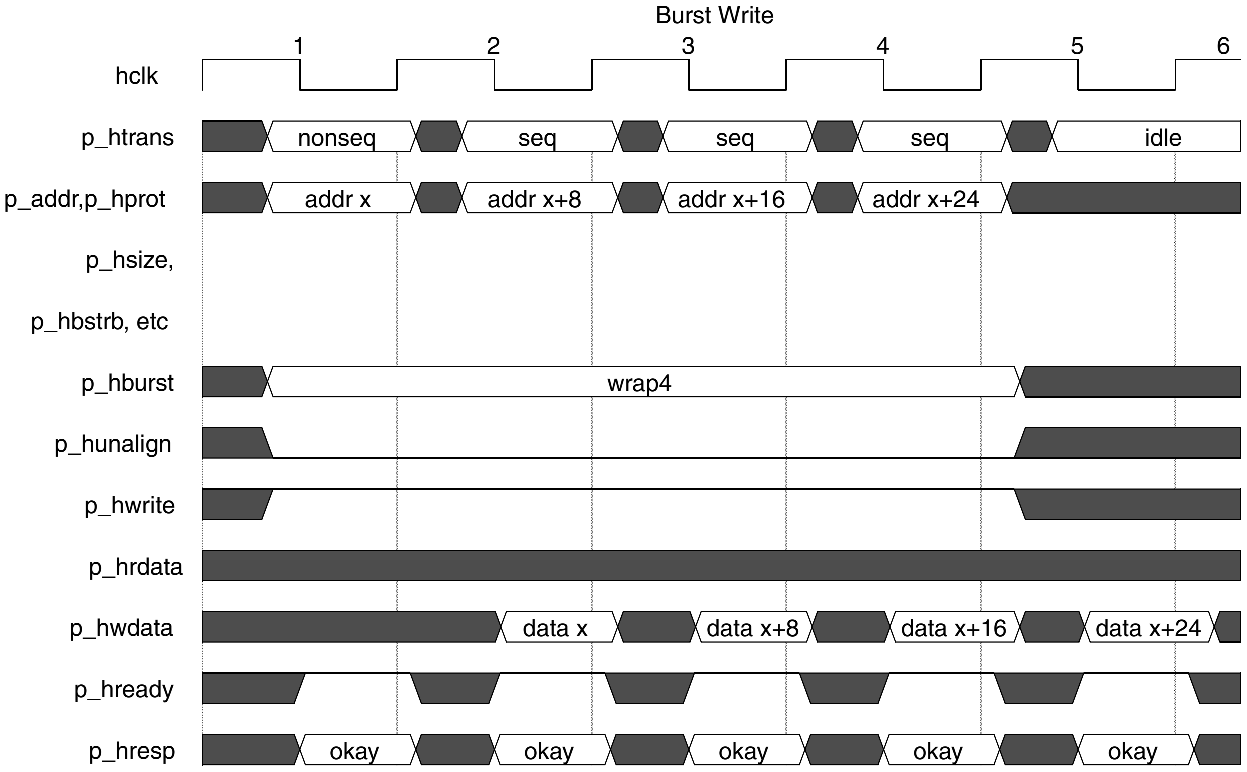


Figure 11-20. Burst Write Transfer

Figure 11-19 illustrates functional timing for a burst write with wait-state transfer.

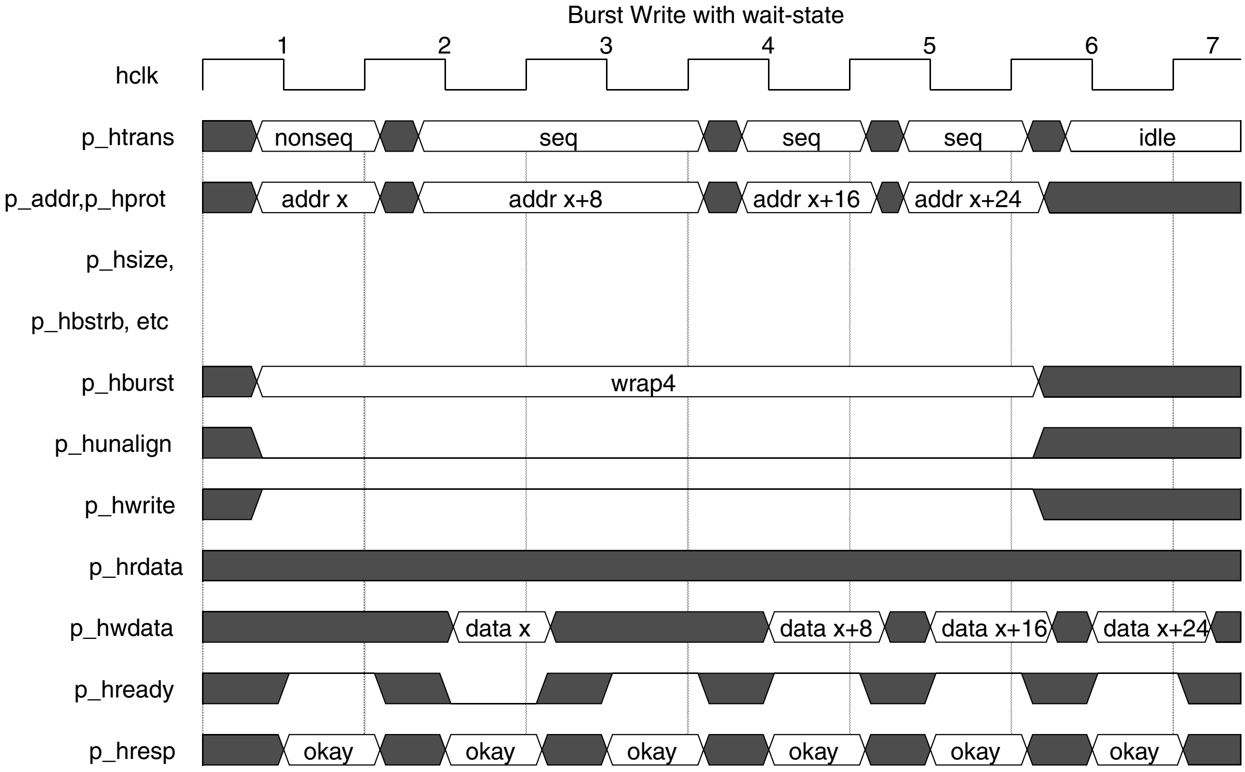


Figure 11-21. Burst Write with Wait-state Transfer

The first cycle of the burst incurs a single wait-state. Data for the second beat of the burst is valid the cycle after the second beat is *taken*.

11.3.2.8 Error Termination Operation

The **p_hresp[2:0]** inputs are used to signal an error termination for an access in progress. The ERROR encoding is used in conjunction with the assertion of **p_hready** to terminate a cycle with error. Error termination is a two-cycle termination. The first cycle consists of signaling the ERROR response on **p_hresp[2:0]** while holding **p_hready** negated. The second cycle consists of asserting **p_hready** while continuing to drive the ERROR response on **p_hresp[2:0]**. This two cycle termination allows the BIU to retract a pending access if it desires to do so. **p_htrans** may be driven to IDLE during the second cycle of the two-cycle error response, or it may change to any other value, and a new access unrelated to the pending access may be requested. The cycle that was previously pending while waiting for a response that terminates with error may be changed. It does not need to remain unchanged when an error response is received.

Figure 11-22 shows an example of error termination.

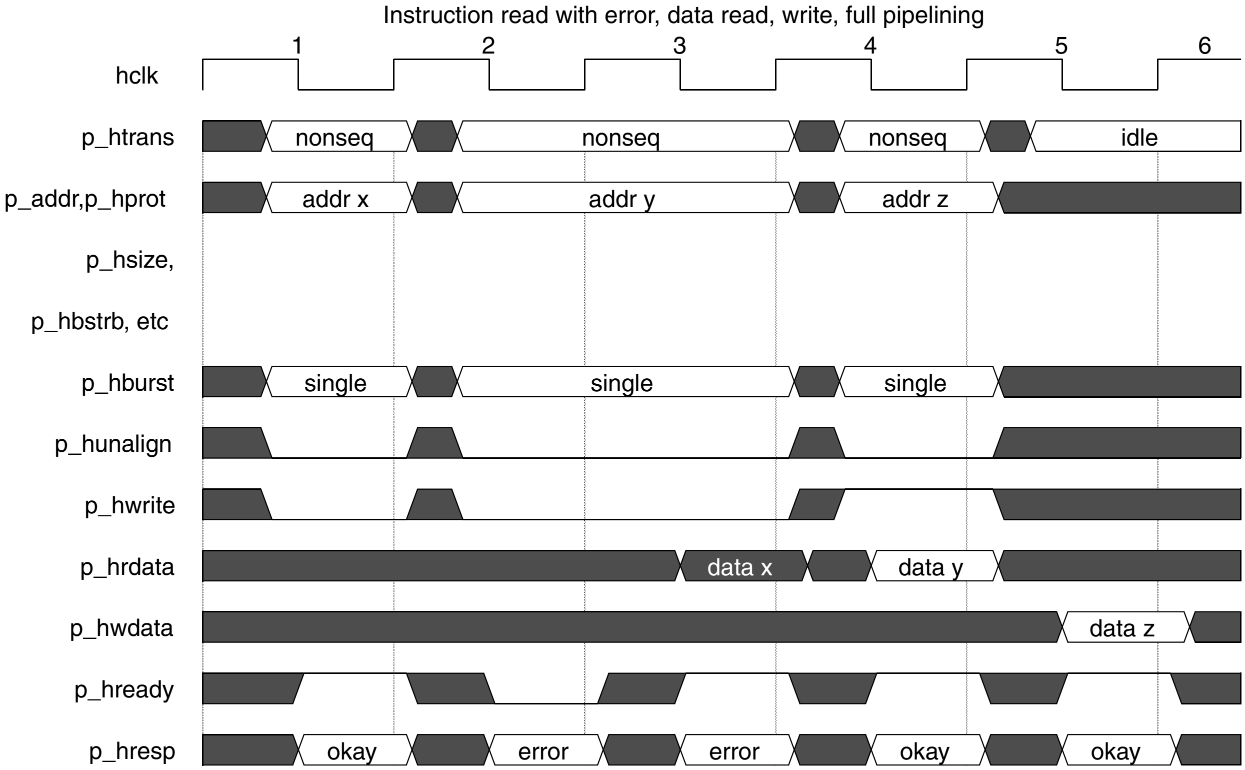


Figure 11-22. Read and Write Transfers, Instr. Read Error Termination

The sequence of events is as follows:

1. The first read request ($addr_x$) is *taken* at the end of cycle C1 since the bus is idle. It is an instruction prefetch.
2. The second read request ($addr_y$) is not *taken* at the end of C2 since the first access is still outstanding (no **p_hready** assertion). An error response is signaled by the addressed slave for $addr_x$ by driving ERROR onto the **p_hresp[2:0]** inputs. This is the first cycle of the two cycle error response protocol.
3. **p_hready** is asserted during C3 for the first read access ($addr_x$) while the ERROR encoding remains driven on **p_hresp[2:0]**, terminating the access. The read data bus is undefined.
4. In this example of error termination, the CPU continues to request an access to $addr_y$. It is taken at the end of C3. During C4, read data is supplied for the $addr_y$ read, and the access is terminated normally during C4.
5. Also during C4, a request is generated for a write to $addr_z$, which is taken at the end of C4 since the second access is terminating.
6. Data for the $addr_z$ write cycle is driven in C5, the cycle after the access is *taken*.
7. During C5, a ready/OKAY response is signaled to complete the write cycle to $addr_z$.

In this example of error termination, a subsequent access remained requested. This does not always occur when certain types of transfers are terminated with error. The following figures outline cases where an error termination for a given cycle causes a pending request to be aborted prior to initiation.

Figure 11-23 shows another example of error termination.

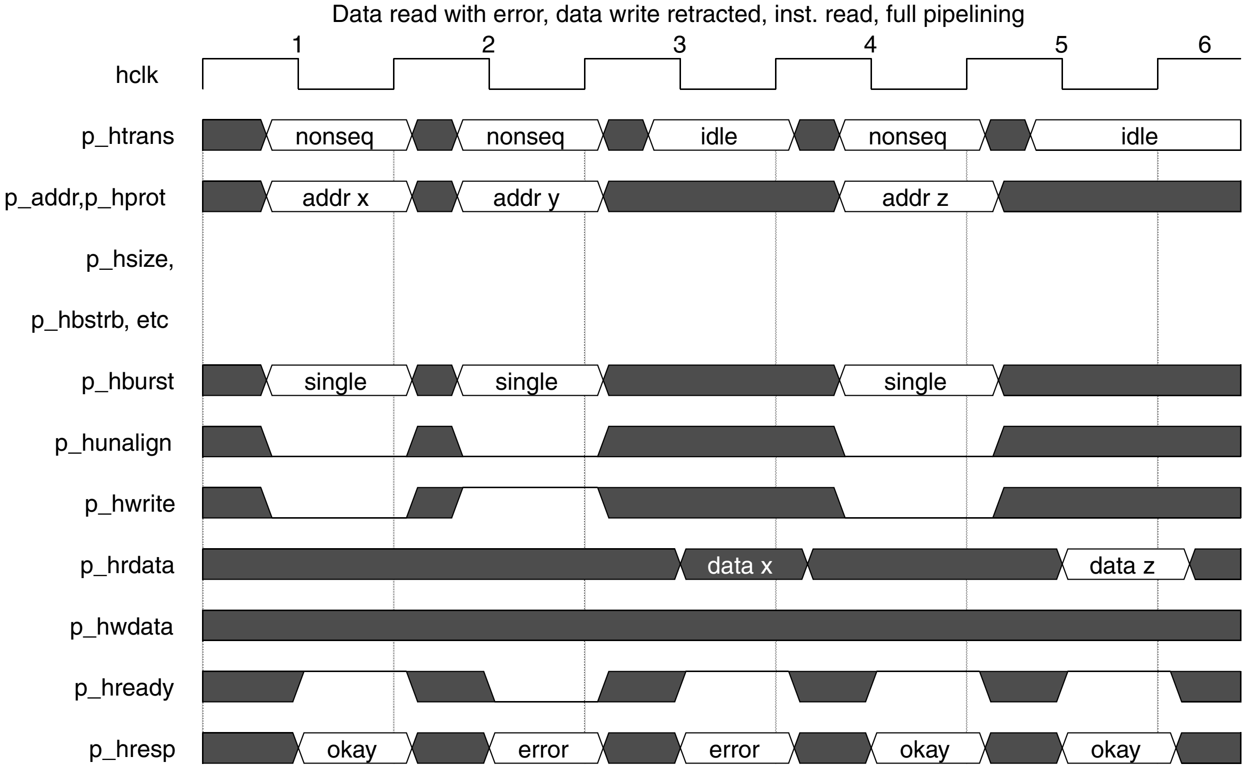


Figure 11-23. Data Read Error Termination

The sequence of events is as follows:

1. The first read request (addr$_x$) is *taken* at the end of cycle C1 since the bus is idle. It is a data read.
2. The second request (write to addr$_y$) is not *taken* at the end of C2 since the first access is still outstanding (no **p_hready** assertion). An error response is signaled by the addressed slave for addr$_x$ by driving ERROR onto the **p_hresp[2:0]** inputs. This is the first cycle of the two cycle error response protocol.
3. **p_hready** is asserted during C3 for the first read access (addr$_x$) while the ERROR encoding remains driven on **p_hresp[2:0]**, terminating the access. The read data bus is undefined.
4. In this example of error termination, the CPU retracts the requested access to addr$_y$ by driving the **p_htrans** signals to the IDLE state during the second cycle of the two-cycle error response.
5. A different access to addr$_z$ is requested during C4 and is taken at the end of C4. During C5, read data is supplied for the addr$_z$ read, and the access is terminated normally.

In this example of error termination, a subsequent access was aborted.

Figure 11-24 shows another example of error termination, this time on the initial portion of a misaligned write.

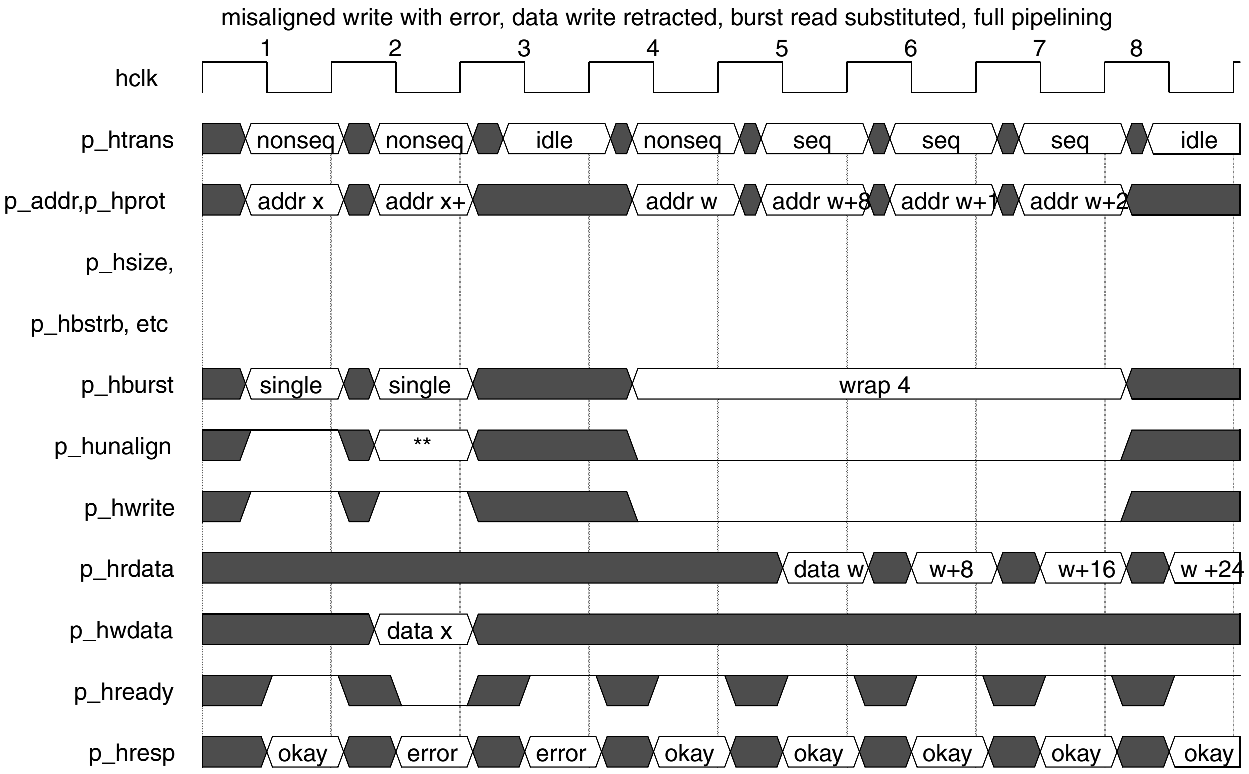


Figure 11-24. Misaligned Write Error Termination, Burst Substituted

The first portion of the misaligned write request is terminated with error. The second portion is aborted by the CPU during the second cycle of the two cycle error response, and a subsequent burst read access to $addr_w$ becomes pending instead.

Figure 11-25 shows another example of error termination—this time on the initial portion of a burst read. The aborted burst is followed by a burst write.

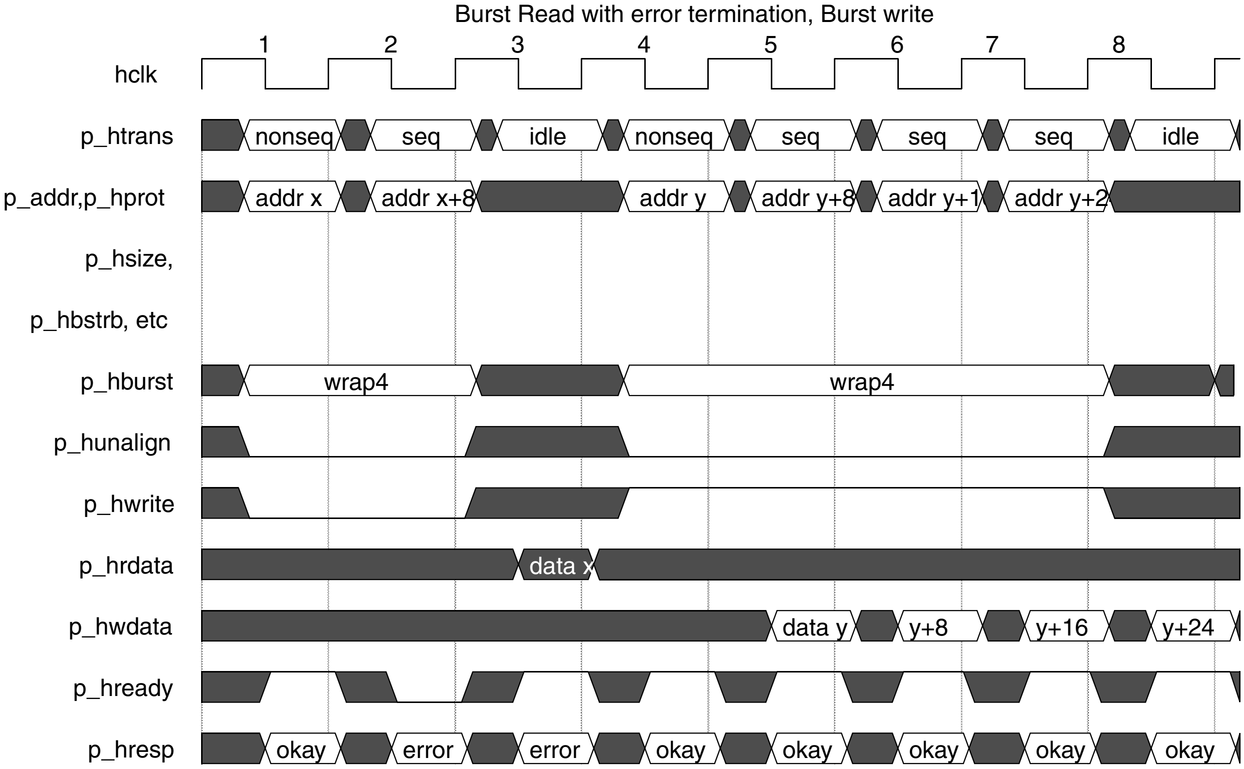


Figure 11-25. Burst Read Error Termination, Burst Write Substituted

The first portion of the burst read request is terminated with error. The second portion is aborted by the CPU during the second cycle of the two cycle error response, and a subsequent burst write access to $addr_y$ becomes pending instead.

11.3.3 Memory Synchronization Control Operation

The memory synchronization signaling interface allows the following functionalities:

- External signaling to the CPU of synchronization operations initiated by execution of an **msync** or **mbar** (MO = 0, 1)
- Handshaking of completion of the operations by other logic within the SoC
- Signaling that a synchronization operation should be performed
- Controlling of the abort of an operation if a pending interrupt is detected by the CPU performing the synchronization instruction
  - Allows minimization of interrupt latency while waiting for completion of the necessary operations required for performing the synchronization.
  - Aborted operation will be reattempted once the interrupt handler has completed and the synchronization instruction is re-executed.

Synchronization operations generally involve flushing any pending stores from the CPU executing the **msync** or **mbar** to their final destinations (performing of pending store operations). The flushing of pending stores requires at minimum the following actions: flushing the store buffers of the initiating CPU and flushing any pending snoop invalidation operations that the operations performed by the initiating CPU prior to execution of the **msync** or **mbar** instruction required. This may involve flushing various store buffers and snoop queues interposed between elements of the coherency domain in the SoC, including coherency manager structures and other queues.

Section 11.2.10, “Memory Synchronization Control Signals,” describes the signals that compose the memory synchronization control interface. The following diagrams show examples of basic operation of the interface: Figure 11-26, Figure 11-27, Figure 11-28, and Figure 11-29.

Figure 11-26 illustrates functional timing for an example memory synchronization basic operation.

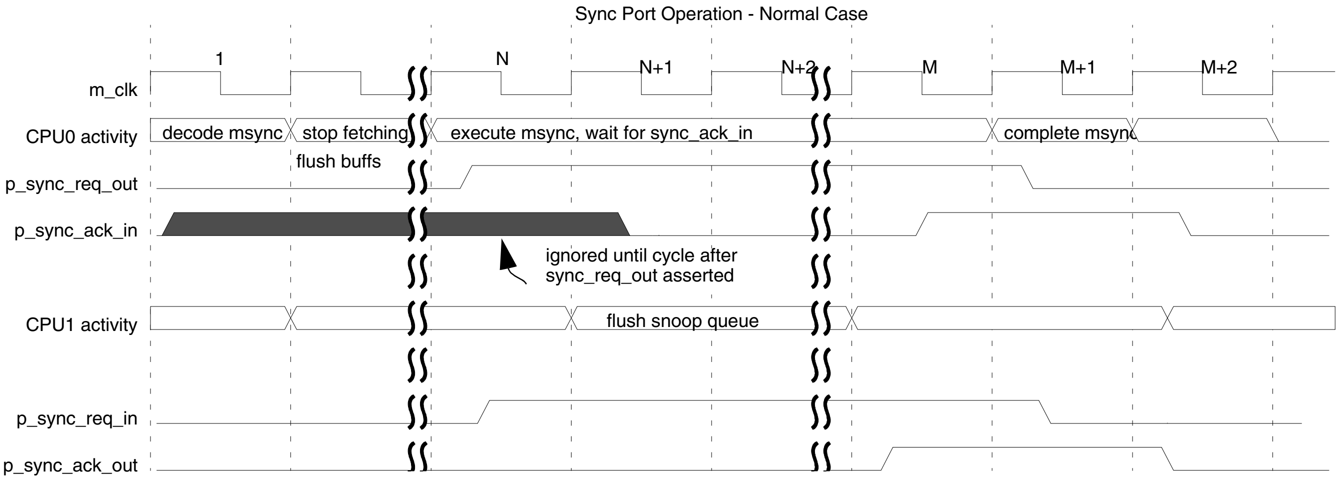


Figure 11-26. Memory Sync Operation (basic operation)

Figure 11-26 shows two CPUs in the system: CPU0 and CPU1. In cycles 1 and 2, CPU0 performs the following actions:

1. Decodes an **msync** instruction
2. Suspends any further operand transfers
3. Flushes the internal push and store buffers to ensure that the results of all previous store instructions are visible

After these actions are complete, CPU0 asserts the **p_sync_req_out** output to signal to the SoC that a memory synchronization operation is to be performed.

Because there are no intermediate buffers or queues in the SoC needing to be flushed, the memory synchronization request input **p_sync_req_in** of CPU1 is asserted in cycle N. If such queues and buffers exist and need to be flushed so that CPU1 can see the previously initiated memory operations performed, there will be additional delay. In cycle N+1 and N+2, CPU1 flushes its internal snoop queue to process all pending snoop operations present at the time of the receipt of the **p_sync_req_in**.

In cycle M, once all of the snoop commands pending up to the point of the memory sync request are processed, CPU1 responds by asserting its **p_sync_ack_out** output signal. This signal is driven back to CPU0's **p_sync_ack_in** input, which results in completion of the memory synchronization operation in cycle M+1. CPU0 negates the **p_sync_req_out** signal in cycle M+1, thus negating CPU1's **p_sync_req_in** signal. In response, in cycle M+2, CPU1 negates **p_sync_ack_out**.

Note that in this simple example, the corresponding inputs and outputs of CPU0 and CPU1 are tied together. However in many systems, this handshaking sequence is controlled by intermediary logic such as a cache coherency manager responsible for the correct directing of synchronization operations to the proper participants.

Figure 11-27 illustrates functional timing for an example memory synchronization operation that is interrupted by a pending interrupt request.

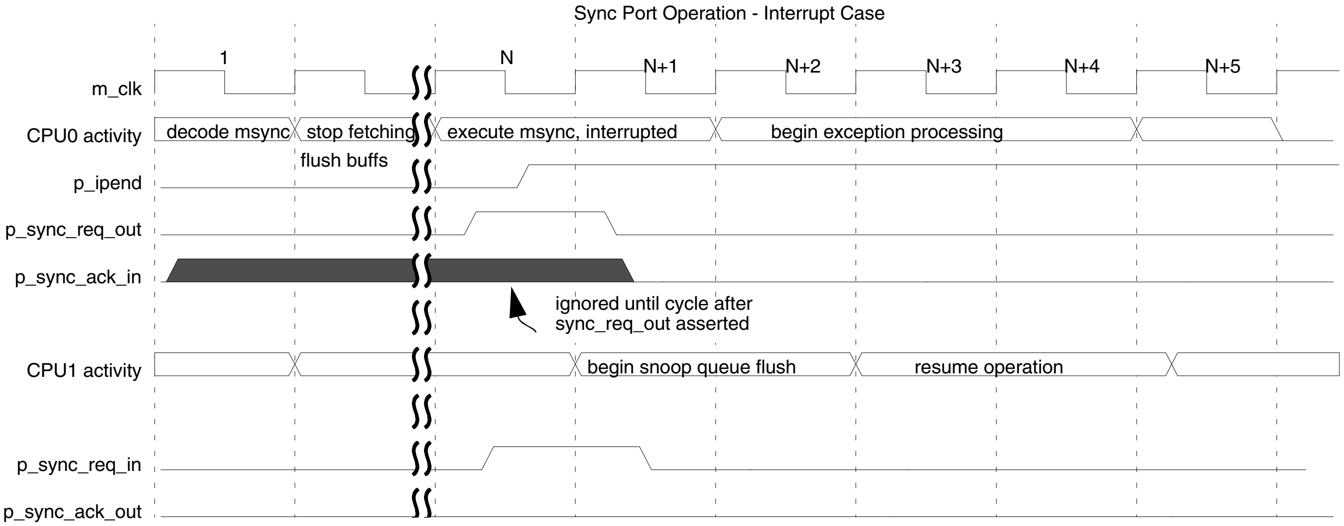


Figure 11-27. Memory Sync Operation (interruption operation)

In cycles 1 and 2, CPU0 performs the following actions:

1. Decodes an **msync** instruction
2. Suspends any further operand transfers
3. Flushes the internal push and store buffers to ensure the results of all previous store instructions have been made visible.

After these actions are complete, CPU0 asserts the **p_sync_req_out** output in cycle N to signal to the SoC that a memory synchronization operation is to be performed. However, an interrupt becomes pending in CPU0.

In cycle N+1, CPU1 begins flushing its internal snoop queue to process all pending snoop operations present at the time of the receipt of the **p_sync_req_in**. However, CPU0 negates the **p_sync_request_out** output prior to receiving a **p_sync_ack_in** completion handshake and aborts the **msync** instruction. CPU1's **p_sync_req_in** signal is negated in response.

In subsequent cycles, CPU0 initiates interrupt exception processing, and CPU1 is free to either complete the flushing of the snoop queue or to abort it and resume normal operation. After CPU0 completes the interrupt handler, it re-initiates the **msync** operation (not shown).

Figure 11-28 illustrates functional timing for another example memory synchronization operation.

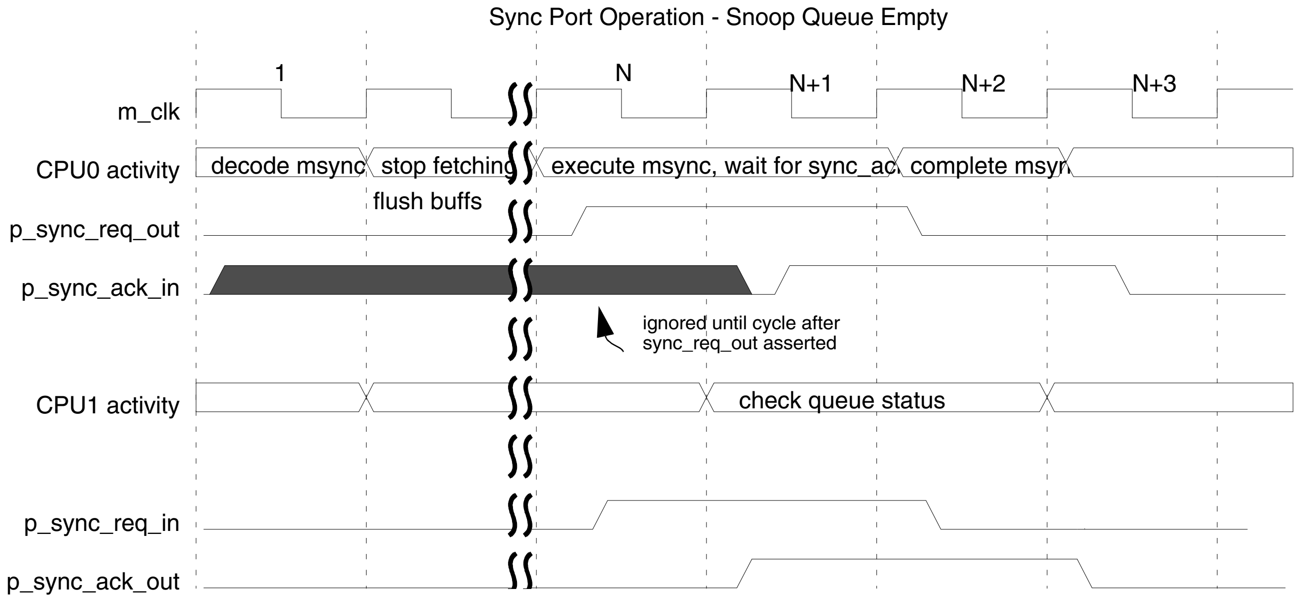


Figure 11-28. Memory Sync Operation (snoop queue empty)

In this example, the snoop queue of CPU1 is empty, and the handshake completes earlier than in other examples. This example shows that there is no minimum time requirement between the assertion of **p_sync_req_in** and the corresponding assertion of **p_sync_ack_out**, although not every implementation responds this quickly.

Figure 11-29 illustrates functional timing for another example memory synchronization operation.

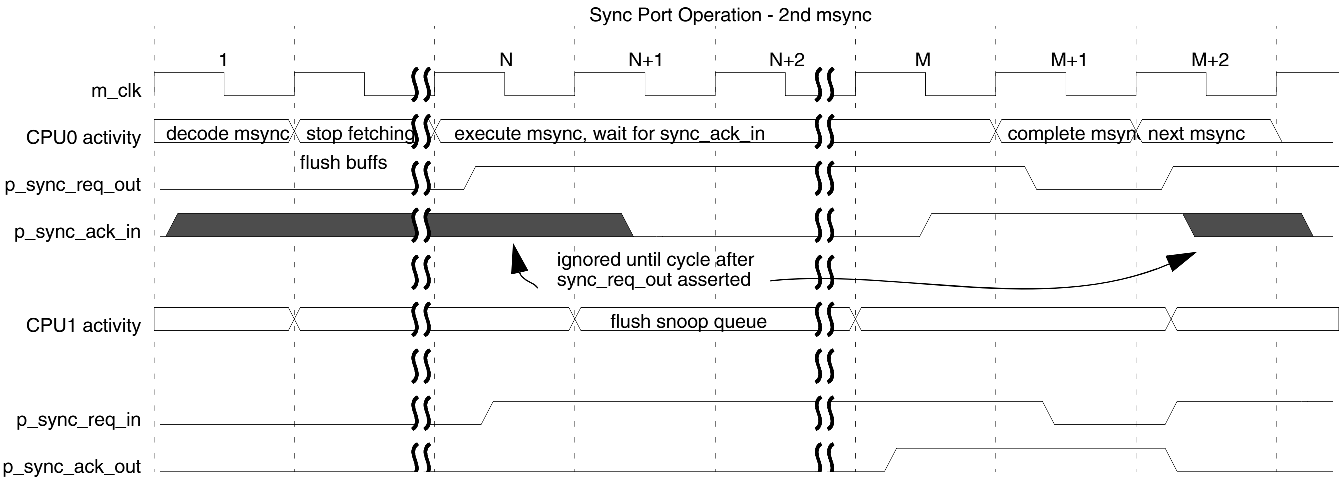


Figure 11-29. Memory Sync Operation (2nd msync back-to-back)

In this example, CPU0 executes back-to-back sync instructions. This example shows that the **p_sync_req_out** output transitions for each individual synchronization request operation, with a minimum of one clock of negation interval between operation requests. Because of the protocol on **p_sync_ack_out** assertion, **p_sync_req_in** must also negate and then reassert in order to request a second synchronization operation.

11.3.4 Cache Error Cross-signaling Operation

The cache error cross-signaling interface allows lockstep operation of two or more CPUs in the presence of cache parity/EDC errors. The interface also allows signaling that one or more errors has occurred and that other cache(s) in the lockstep operation should emulate an error condition. During valid cache lookups, if a parity/EDC error is detected in a CPU, the **p_cache_tagerr_out** and **p_cache_dataerr_out** outputs indicate the type of error, and the **p_tagerrway_out[0:3]** and **p_drterrway_out[0:3]** outputs indicate the cache way(s) incurring the error.

In a dual-CPU lockstep system, these outputs are normally tied to the corresponding **p_cache_err_in**, **p_cache_tagerr_in**, **p_cache_dataerr_in**, **p_tagerrway_in[0:3]**, and **p_drterrway_in[0:3]** inputs of the other CPU. The address corresponding to the error is signaled on the **p_cerraddr_out[0:31]** outputs. The **p_cache_tagerr_out** and **p_cache_dataerr_out** outputs are mutually exclusive and cannot both be asserted at the same time. The **p_tagerrway_out[0:3]** and **p_drterrway_out[0:3]** outputs may be asserted at the same time—but not for the same way—to indicate cache ways with uncorrectable errors and cache ways undergoing dirty error correction. For a given way, these outputs are mutually exclusive because dirty or potentially dirty lines are not auto-invalidated.

It is not expected that the two CPUs would incur an error during the same lookup cycle. However, if detection of this issue is needed, system logic may be utilized to detect the simultaneous assertion of more than one CPU’s error output signal(s) and perform appropriate error recovery, such as a reset operation if the errors cannot both be handled by the CPU. Most errors can be simultaneously handled, however, and the probability of occurrence is practically zero.

Enabling of cache error cross-signaling is performed by assertion of the p_lkstep_en input signal. When p_lkstep_en is negated, the **p_cache_tagerr_in**, **p_cache_dataerr_in**, **p_tagerrway_in[0:3]**, and **p_drterrway_in[0:3]** inputs are ignored. In the following subsections, it is assumed that p_lkstep_en has been properly asserted.

11.3.4.1 Cross-signaling with Machine Check Operation Selected

Figure 11-30 illustrates functional timing for a cross-signaling operation by a CPU that encounters an internal cache error with the error action indicating that a machine check should be generated. A cache error is detected in cycle 3 and results in a machine check exception being signaled.

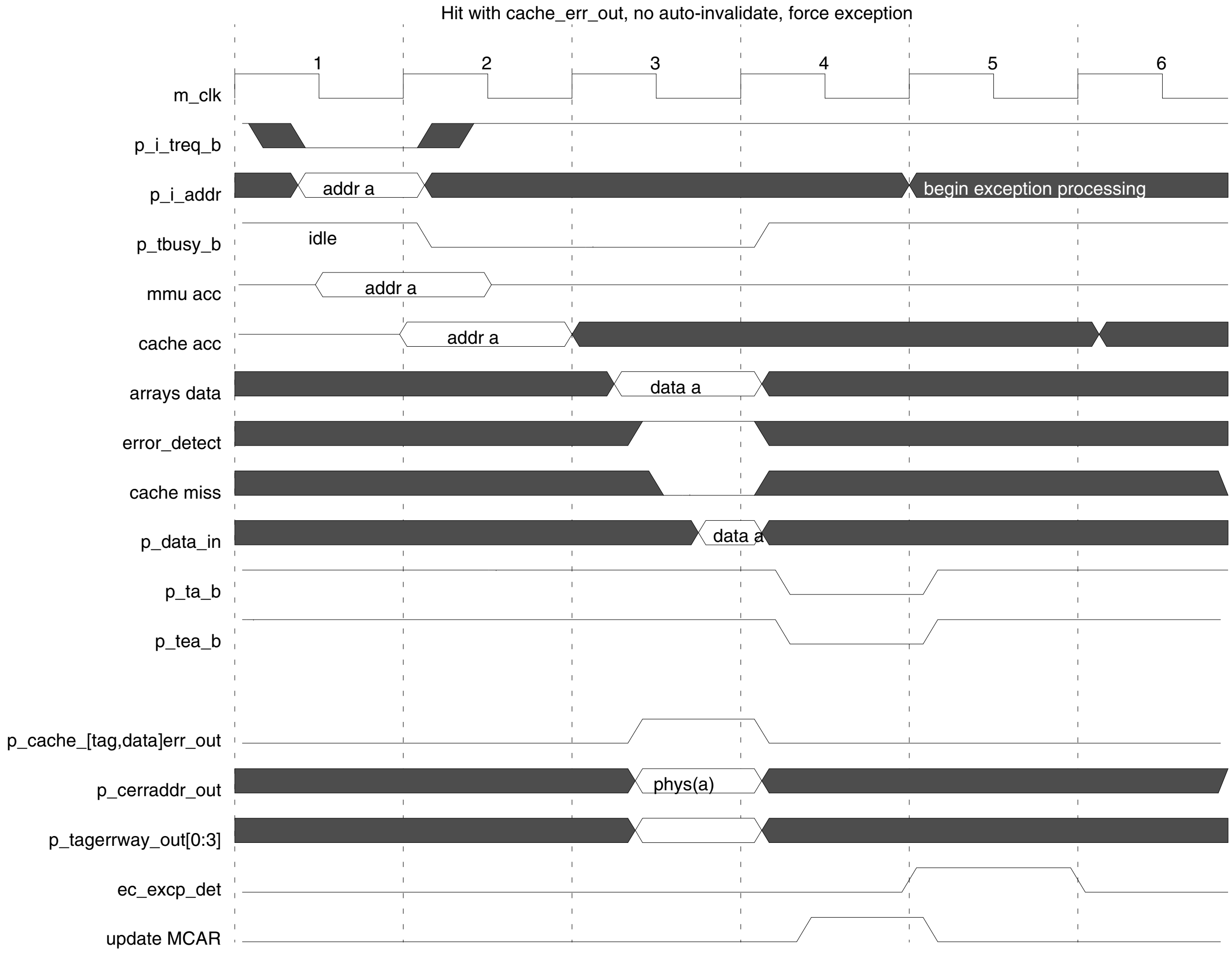


Figure 11-30. Cross-signaling Exception Output Operation

For cross-signaling operations during **dcbi**/**icbi** invalidate operations when machine check error action is selected (L1CSR0[DCEA]/L1CSR1[ICEA] = 00), the signaling of a **p_cache_tagerr_out** event indicates that a false hit to one or more unlocked lines occurred. In this case, the line(s) should be invalidated in the other CPU(s) regardless of hit or miss conditions rather than cause a machine check condition. The way(s) that incurred a false hit are signaled on the **p_tagerrway_out[0:3]** outputs. This is currently the only situation in which a machine check is not generated due to signaling of a **p_cache_tagerr_out** event when operating with machine check error action enabled (L1CSR0[DCEA]/L1CSR1[ICEA] = 00).

Figure 11-31 illustrates functional timing for a cross-signaling operation by a CPU that receives a cache error cross-signaling operation with the error action indicating that a machine check should be generated.

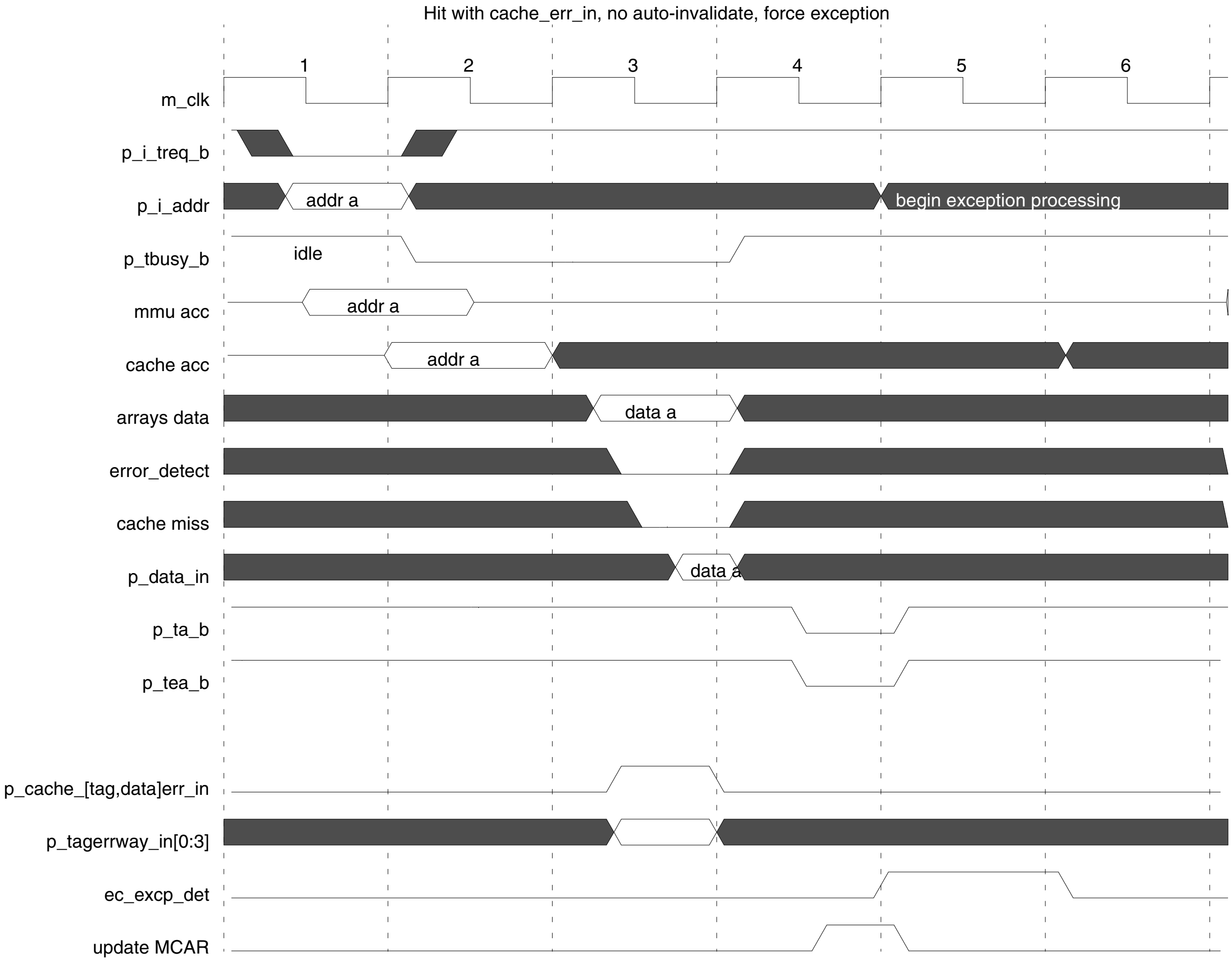


Figure 11-31. Cross-signaling Exception Input Operation

A cache error is detected in cycle 3 by an external cache and results in a machine check exception being generated.

11.3.4.2 Cross-signaling with Auto-invalidation Operation Selected

Figure 11-32 illustrates functional timing for a cross-signaling operation by a CPU that encounters an internal cache error in the cache data array. The error action indicates that an auto-invalidation/correction should be generated for way 2 by refilling the cache line. A cache data array error is detected in cycle 3, which results in a cache miss being forced. The error entry is refilled beginning in cycle 5.

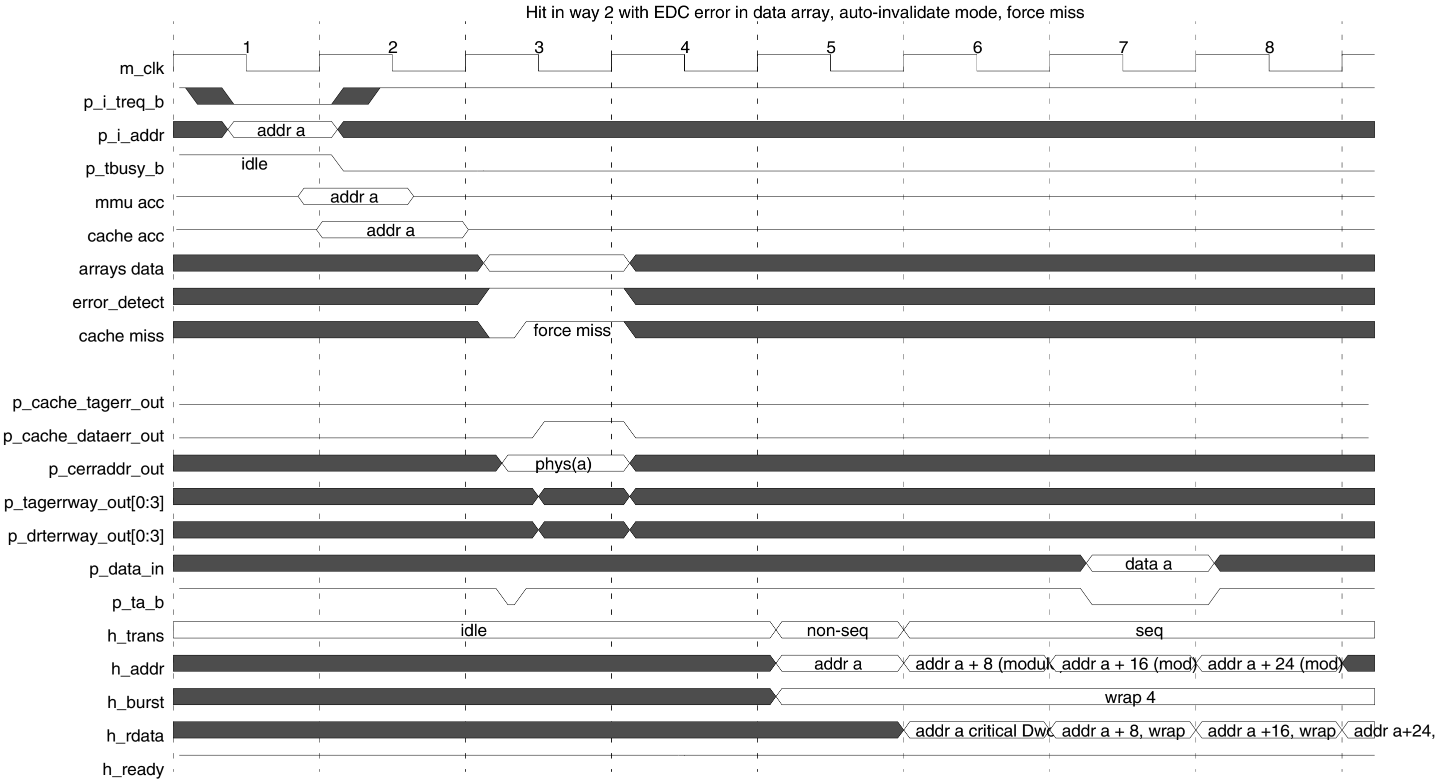


Figure 11-32. Cross-signaling Invalidation Output Operation—Data Error

Figure 11-33 illustrates functional timing for a cross-signaling operation by a CPU that encounters internal cache errors in the cache tag array with the error action indicating that an auto-invalidation/correction should be generated (ICEA = 01).

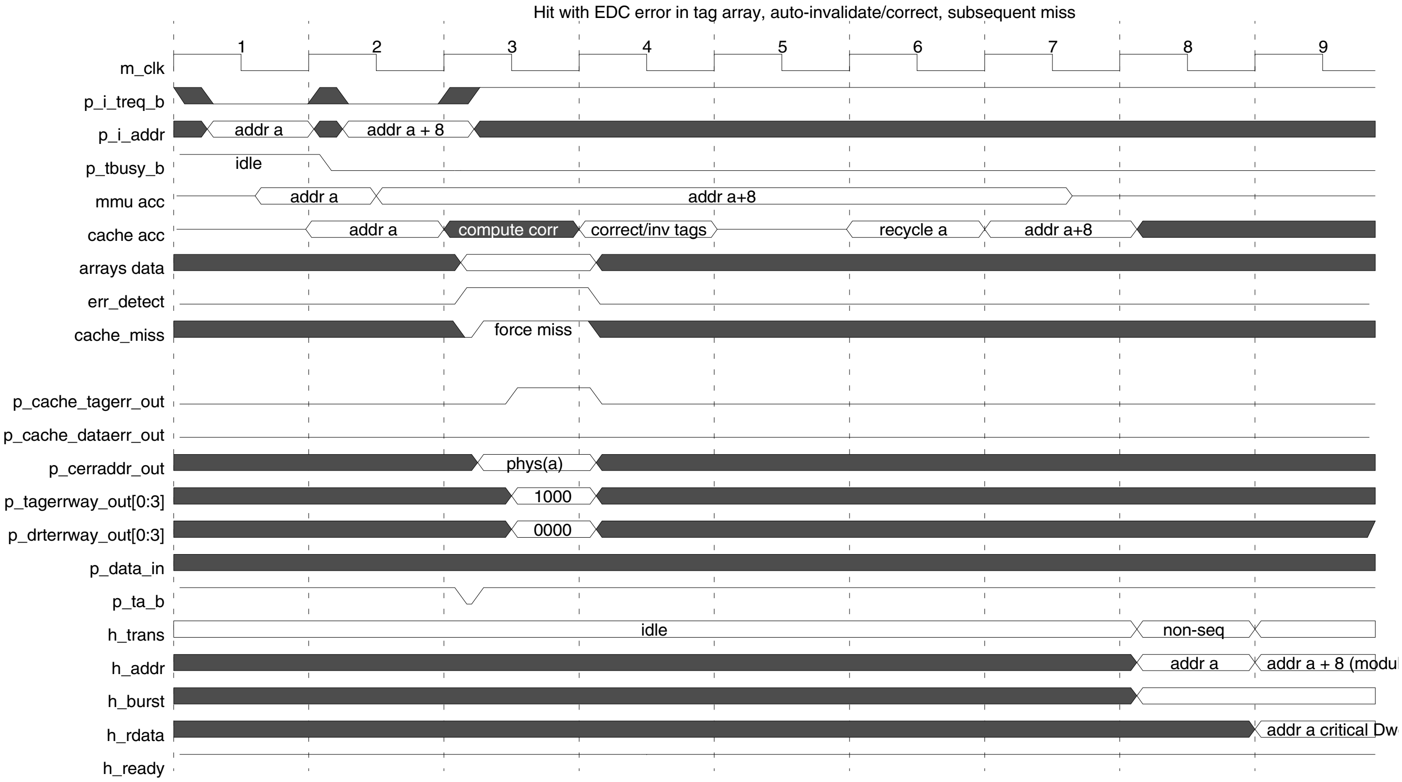


Figure 11-33. Cross-signaling Invalidation Output Operation—Tag Error, Miss

A cache tag array error is detected in cycle 3 and results in a cache correction/invalidation sequence being forced beginning in cycle 4. In this example, way 2 has a correctable error, and way 0 has an uncorrectable error and requires invalidation. The invalidation of way 0 is signaled in cycle 3 and performed in cycle 4. Way 0 would have resulted in a hit for access ‘a’, but the tag was uncorrectable. The cache is refilled beginning in cycle 8.

Note that the receiving CPU forces a miss and performs the same reload, even though it detected a hit for access ‘a’ to way 0 in cycle 3, because way 0 error is indicated on the **p_cache_tagerr_out** and **p_tagerrway_out[0:3]** outputs. Also note that way 2 is not indicated on the **p_tagerrway_out[0:3]** outputs because it is correctable.

Figure 11-34 illustrates functional timing for a cross-signaling operation by a CPU that encounters an internal cache error in the cache tag array with the error action indicating that an auto-invalidation/correction should be generated.

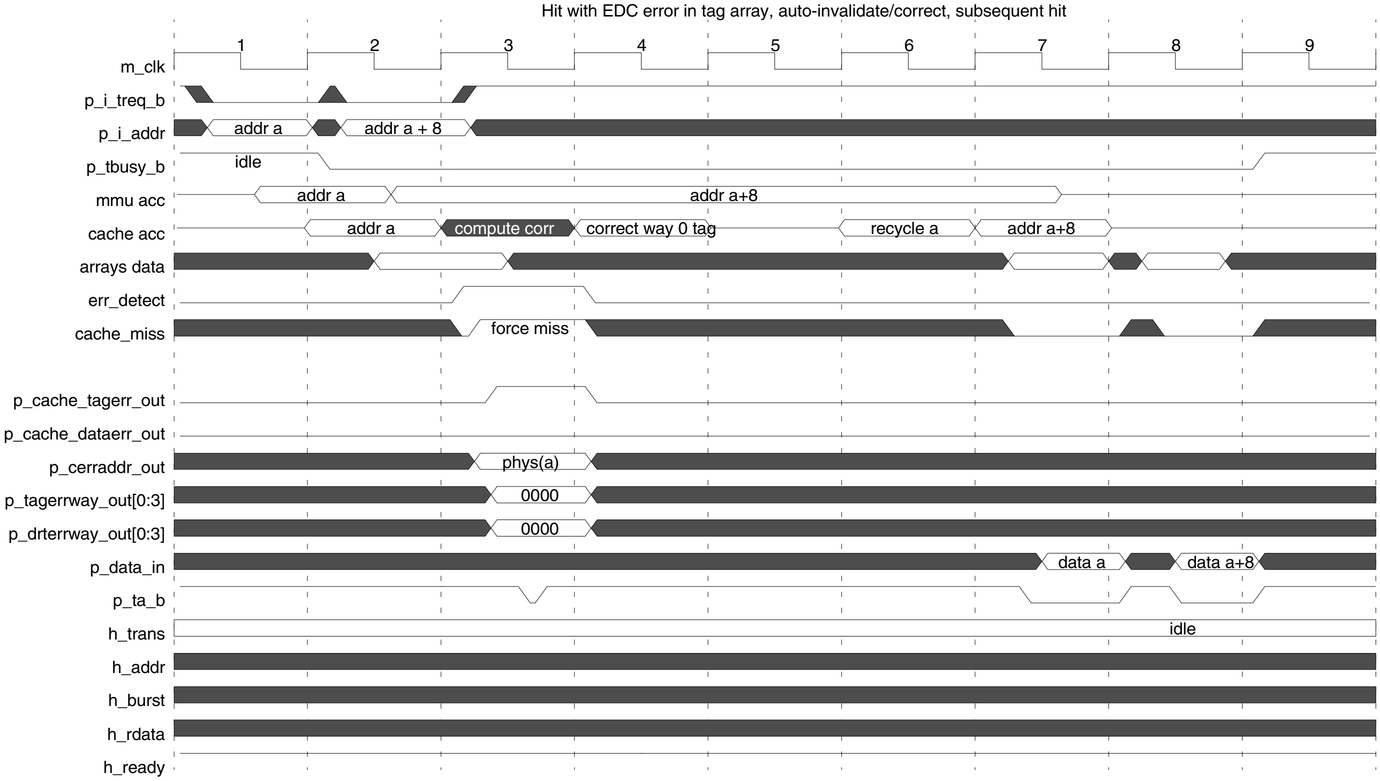


Figure 11-34. Cross-signaling Invalidation Output Operation—Tag Error, Hit

A cache tag array error is detected in cycle 3, which results in a cache correction/invalidation cycle being forced. Way 0 has a correctable error. The correction of way 0 is performed in cycle 4. The address ‘a’ access is recycled in cycle 6 and results in a hit.

Note that way 0 is not indicated on the **p_tagerrway_out[0:3]** outputs because it is correctable.

Figure 11-35 illustrates functional timing for a cross-signaling operation by a CPU that encounters an internal cache error in the cache tag array with the error action indicating that an auto-invalidation/correction should be generated.

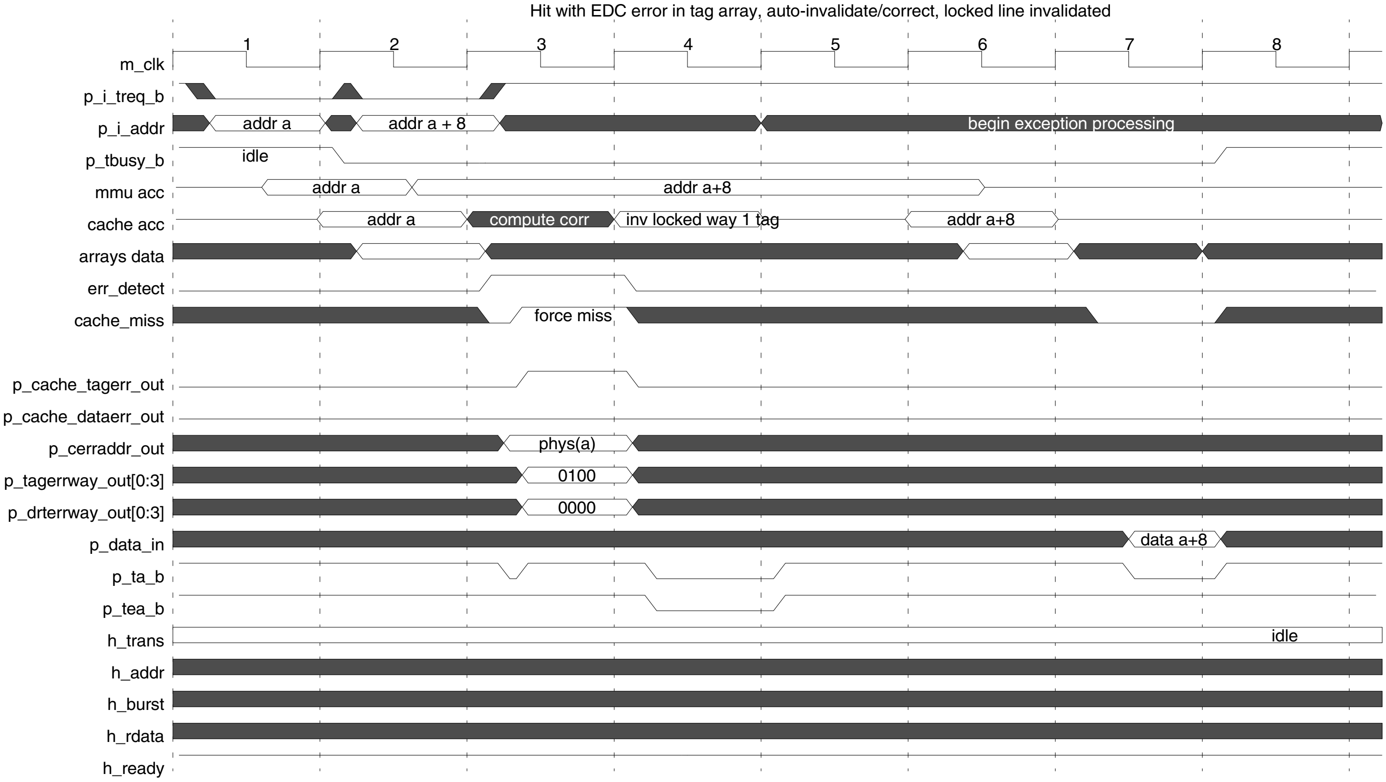


Figure 11-35. Cross-signaling Invalidation Output Operation—Tag Error, Locked Inv

A cache tag array error is detected in cycle 3, which results in a cache correction/invalidation cycle being forced. In this example, way 1 has an uncorrectable error and is locked. The invalidation of way 1 is performed in cycle 4, and a machine check is signaled.

Figure 11-36 illustrates functional timing for a cross-signaling operation by a CPU that encounters an internal cache error in the data cache dirty array. The error action indicates that an auto-invalidation/correction should be generated.

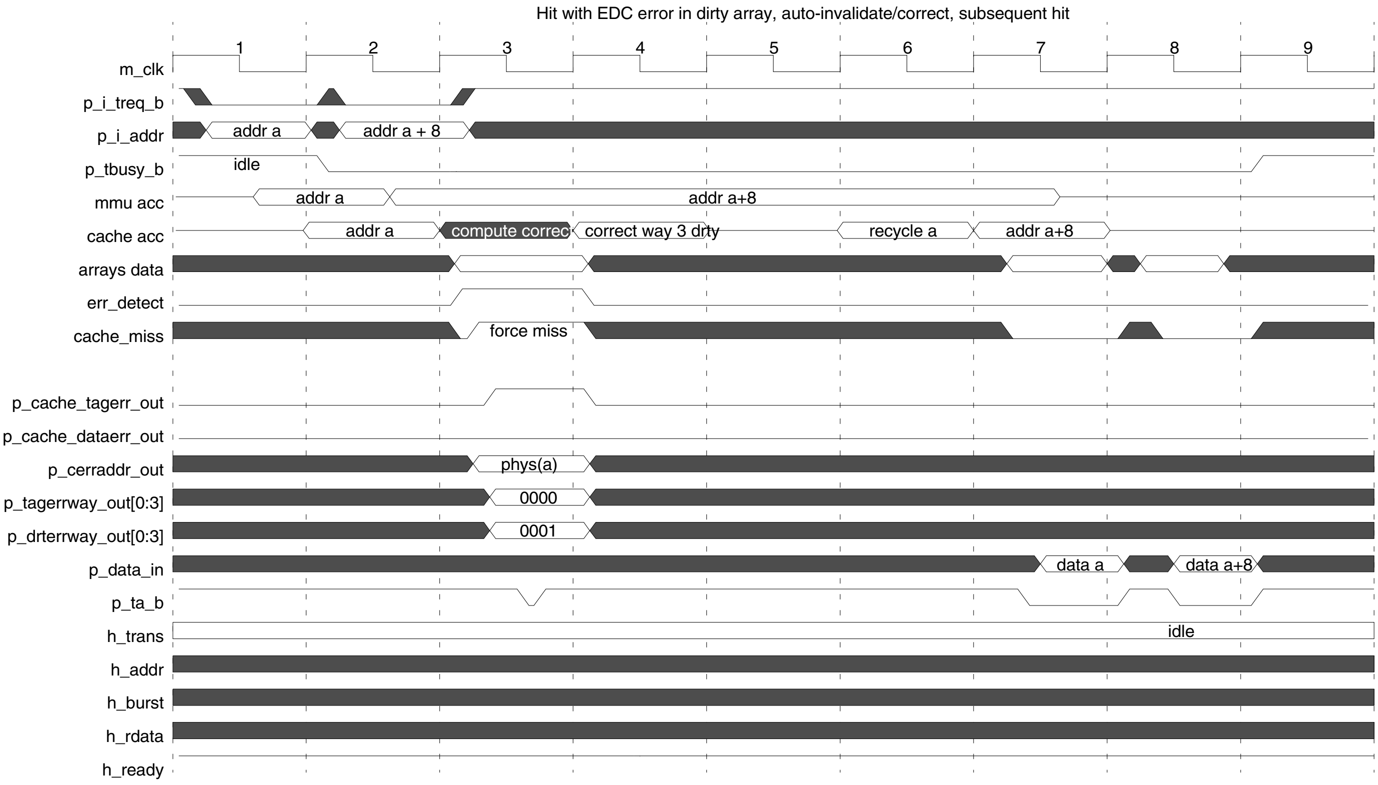


Figure 11-36. Cross-signaling Invalidation Output Operation—Dirty Error

A dirty array error is detected in cycle 3 due to a difference in one or more of the three dirty bits for a cache line, which results in a cache correction/invalidation cycle being forced.Way 3 has a dirty error and is corrected by rewriting all three dirty bits to a ‘1’. The dirty error is signaled on the **p_drterrway_out[0:3]** outputs. The re-write of way 3 dirty array is performed in cycle 4. The address ‘a’ access is recycled in cycle 6 and results in a hit.

Note that way 3 is not indicated on the **p_tagerrway_out[0:3]** outputs because it is correctable.

Figure 11-37 illustrates functional timing for a cross-signaling operation by a CPU which encounters internal cache errors in both the tag array and in the dirty array with the error action indicating that an auto-invalidation/correction should be generated.

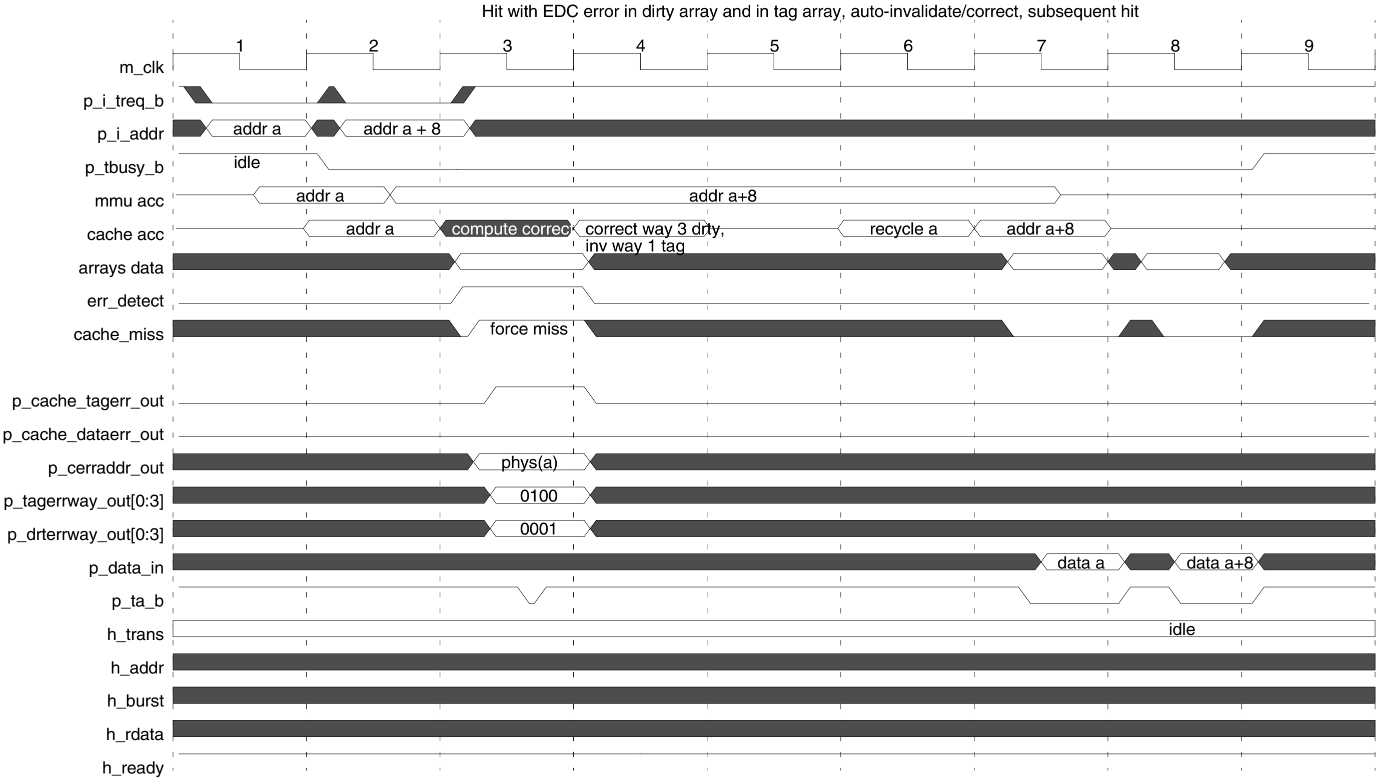


Figure 11-37. Cross-signaling Invalidation Output Operation—Tag Error, Dirty Error

The errors are detected in cycle 3 due to a difference in one or more of the three dirty bits for a cache line, and for the tag error and results in a cache correction/invalidation cycle being forced. Way 3 has a dirty error and is corrected by rewriting all three dirty bits to a ‘1’. Way 1 has an uncorrectable tag error and is auto-invalidated. The dirty error for way 3 is signaled on the **p_drterrway_out[0:3]** outputs, and the tag invalidation for way 1 is signaled on the **p_tagerrway_out[0:3]** outputs. The re-write of way 3 dirty array and the invalidation of the way 1 tag is performed in cycle 4, and the address ‘a’ access is recycled in cycle 6 and results in a hit.

Note that way 3 is not indicated on the **p_tagerrway_out[0:3]** outputs because it is correctable.

Figure 11-38 illustrates functional timing for a cross-signaling operation by a CPU which encounters internal cache errors in both the tag array, lock array, and in the dirty array with the error action indicating that an auto-invalidation/correction should be generated.

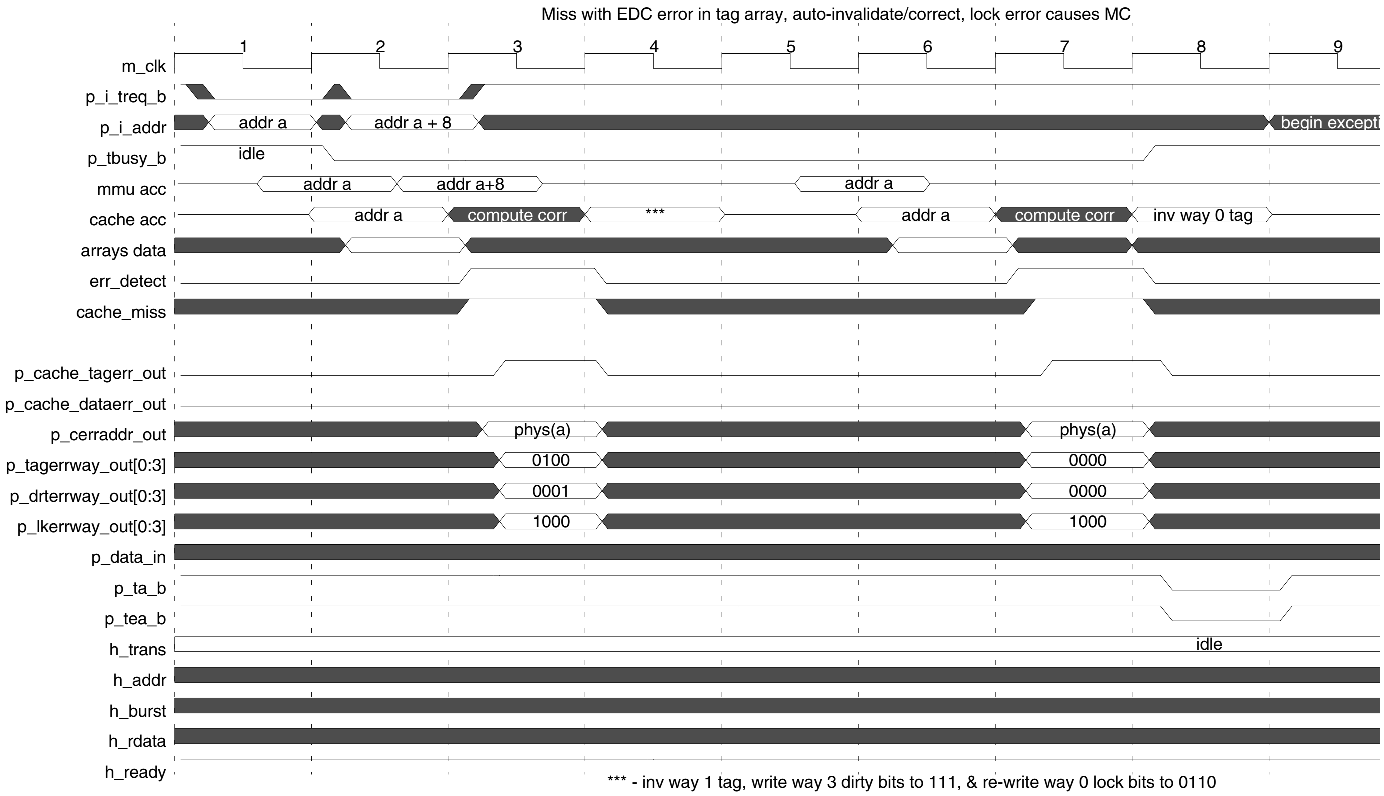


Figure 11-38. Cross-signaling Invalidation Output Operation—Tag Error, Dirty Error, and Lock Error

The errors are detected in cycle 3 due to a difference in one or more of the three dirty bits for a cache line, and for the tag error and lock error and results in a cache correction/invalidation cycle being forced. Way 4 has a dirty error and is corrected by rewriting all three dirty bits to a 1. Way 1 has an uncorrectable tag error and is auto-invalidated. Way 0 has an uncorrectable lock error which is signaled to the other CPU(s), and the lock bits are rewritten to 0110. The dirty error for way 4 is signaled on the **p_drterrway_out[0:3]** outputs; the lock error for way 0 is signaled on the **p_lkerrway_out[0:3]** outputs; and the invalidation for way 1 is signaled on the **p_tagerrway_out[0:3]** outputs.

Note that way 3 is not indicated on the **p_tagerrway_out[0:3]** outputs because it is correctable.

The re-write of way 4 dirty array, the rewrite of the lock bits to the double error value 0110, and the invalidation of the way 1 tag are performed in cycle 4. The address ‘a’ access is recycled in cycle 6 and results in a miss. The lock error is re-detected, re-signaled in cycle 7, and because of the miss, way 0 is invalidated in cycle 8 and a machine check is generated due to the invalidation of a locked way. Exception processing begins in cycle 9.

Figure 11-39 illustrates functional timing for a cross-signaling operation by a CPU that receives a cache error cross-signaling operation for the cache data array with the error action indicating that an auto-invalidation/correction should be generated for hitting way 2 by refilling the cache line. A cache data array error input signaling is detected in cycle 3 and results in a cache miss being forced. The hitting entry is refilled beginning in cycle 5.

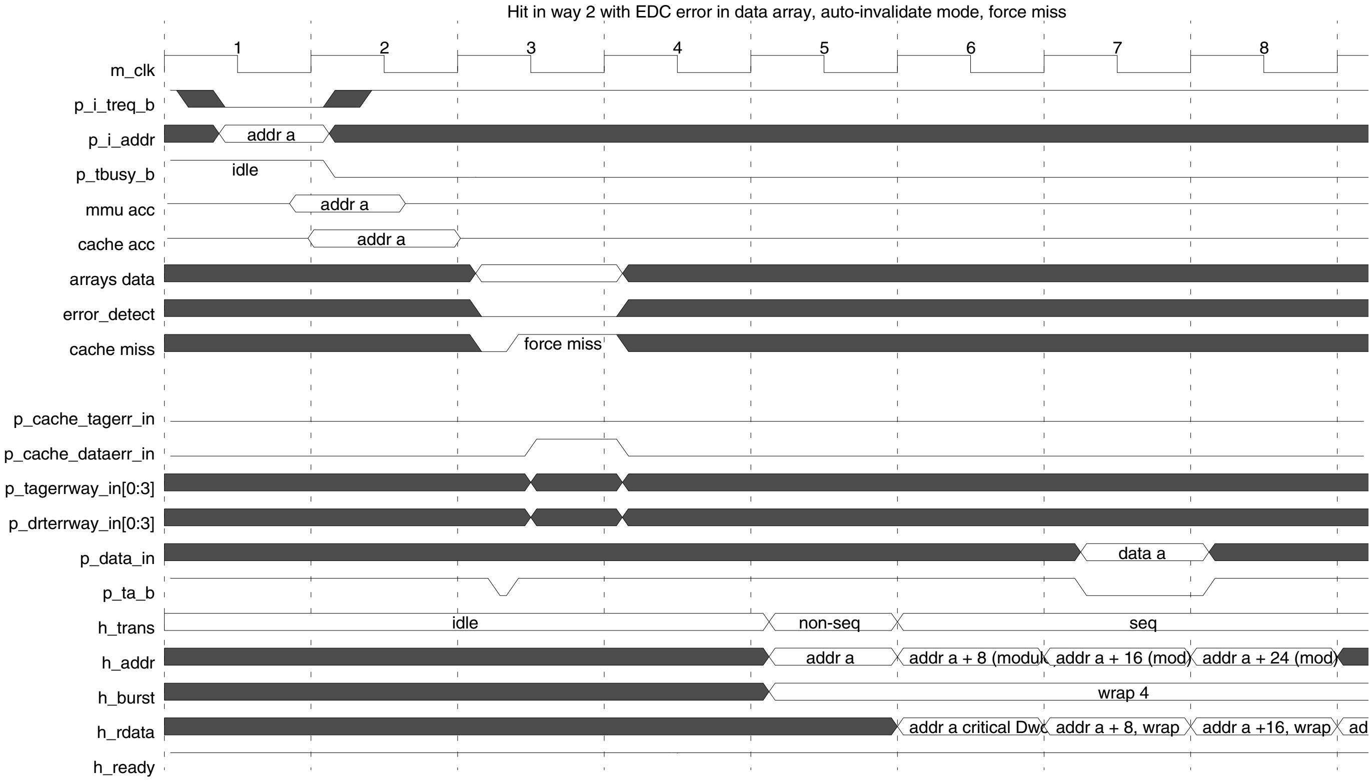


Figure 11-39. Cross-signaling Invalidation Input Operation—Data Error

Figure 11-40 illustrates functional timing for a cross-signaling operation by a CPU that receives a cache error cross-signaling operation for the cache tag array. The error action indicates that an auto-invalidation/correction should be generated (ICEA = 01).

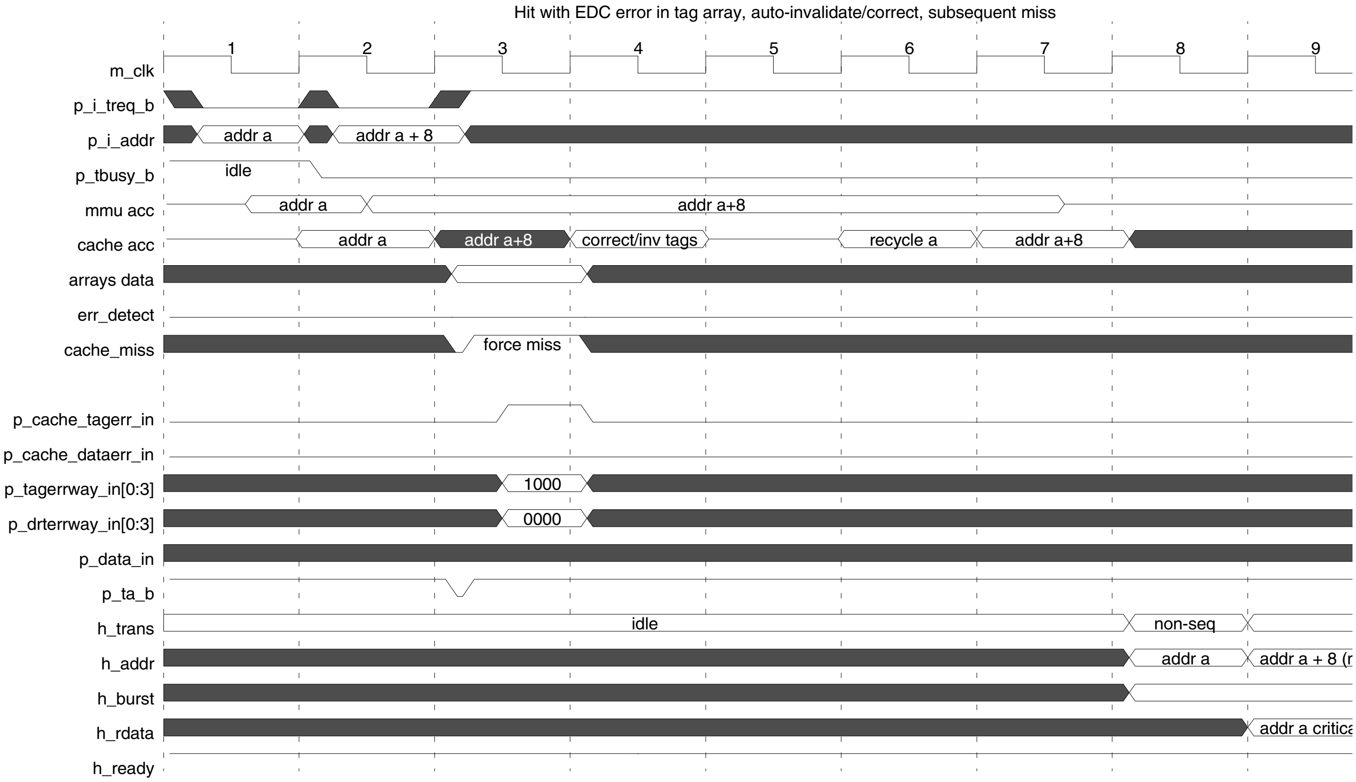


Figure 11-40. Cross-signaling Invalidation Input Operation—Tag Error, Miss

A cache tag array error signaling operation is detected in cycle 3 and results in a cache correction/invalidation sequence being forced beginning in cycle 4. In the external cache, way 0 has an uncorrectable error and requires invalidation. The invalidation of way 0 is signaled in cycle 3 and performed in cycle 4. In this example, way 0 would have resulted in a hit for access ‘a’, but the tag was invalidated. The cache is refilled beginning in cycle 8.

Note that the receiving CPU forces a miss and performs the same reload as the cache signaling the error even though it detected a hit for access ‘a’ to way 0 in cycle 3, since the way 0 error is indicated on the **p_cache_tagerr_in** and **p_tagerrway_in[0:3]** inputs.

Figure 11-41 illustrates functional timing for a cross-signaling operation by a CPU that receives a cache error cross-signaling operation for the cache tag array. The error action indicates that an auto-invalidation should be generated.

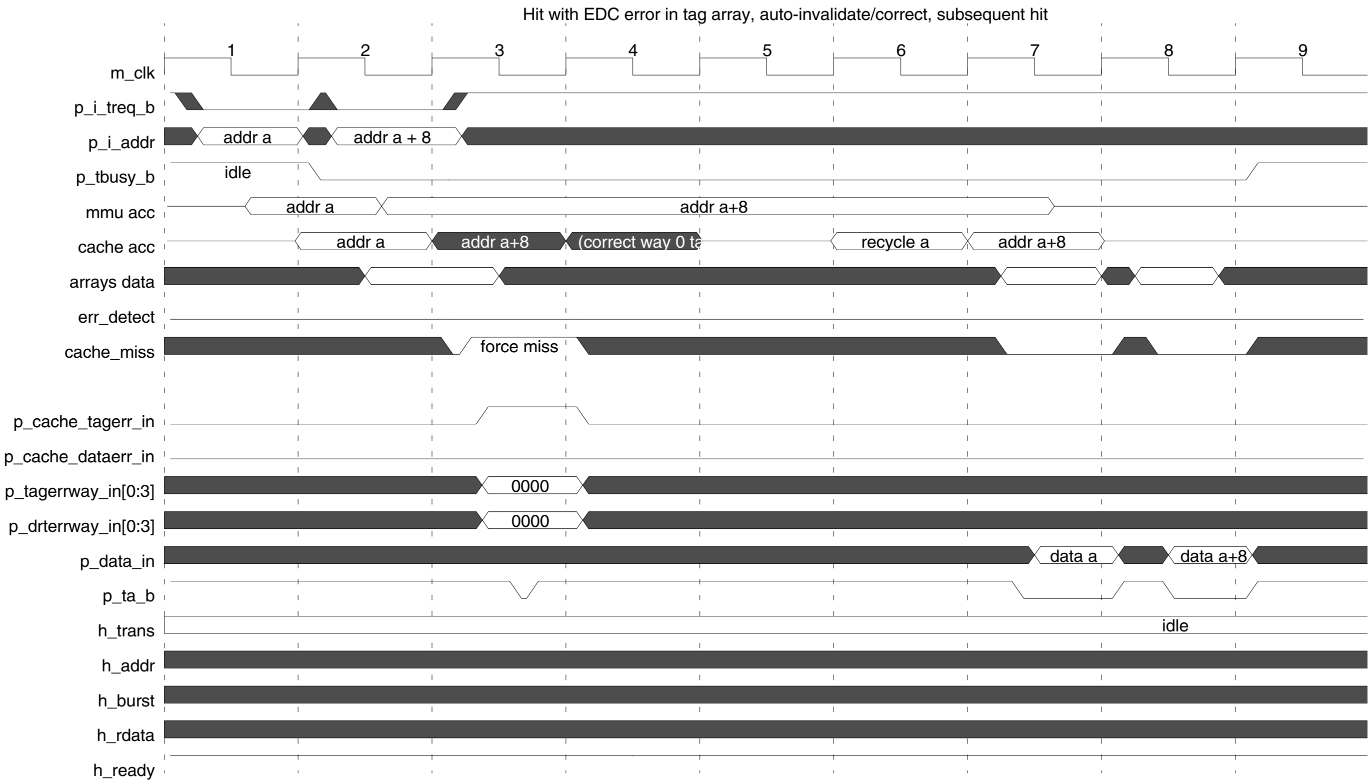


Figure 11-41. Cross-signaling Invalidation Input Operation—Tag Error, Hit

A cache tag array error is detected in cycle 3 by an external cache and results in a cache correction/invalidation cycle being forced. In the external cache, way 1 has a correctable error. Only the invalidations are signaled on the **p_tagerrway_in[0:3]** inputs. Because no invalidations are required, no access is performed in cycle 4. Cycle 4 corresponds to the correction cycle in the signaling cache. The address ‘a’ access is recycled in cycle 6 and results in a hit.

Figure 11-42 illustrates functional timing for a cross-signaling operation by a CPU that receives a cache error cross-signaling operation for the cache tag array. The error action indicates that an auto-invalidation should be generated.

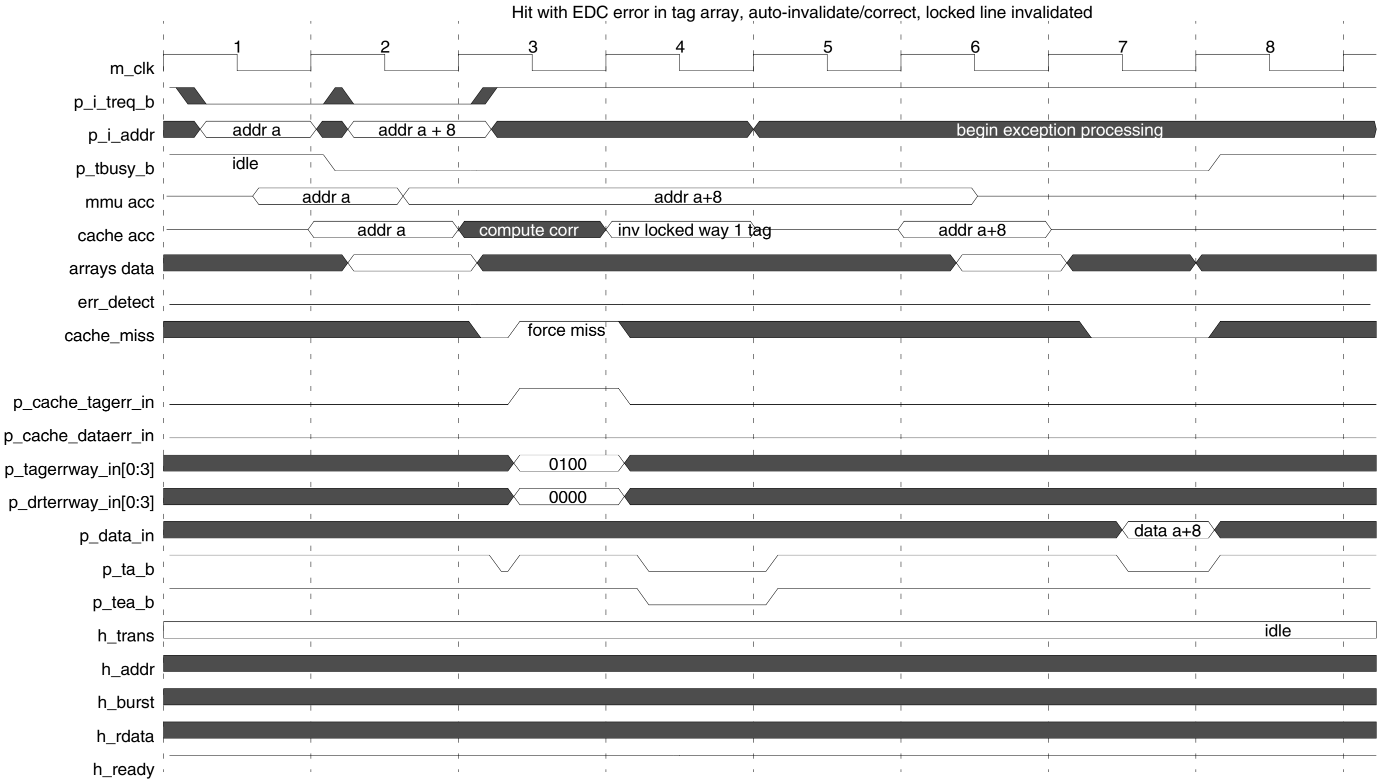


Figure 11-42. Cross-signaling Invalidation input Operation—Tag Error, Locked Inv

A cache tag array error is cross-signaled in cycle 3 and results in a cache correction/invalidation cycle being forced. Way 1 has an uncorrectable error in the external cache and is locked. The invalidation of way 1 is performed in cycle 4, and a machine check is signaled, since the way is also locked in the receiving cache.

Figure 11-43 illustrates functional timing for a cross-signaling operation by a CPU that receives a cache error cross-signaling operation for the data cache dirty array with the error action indicating that a correction should be generated.

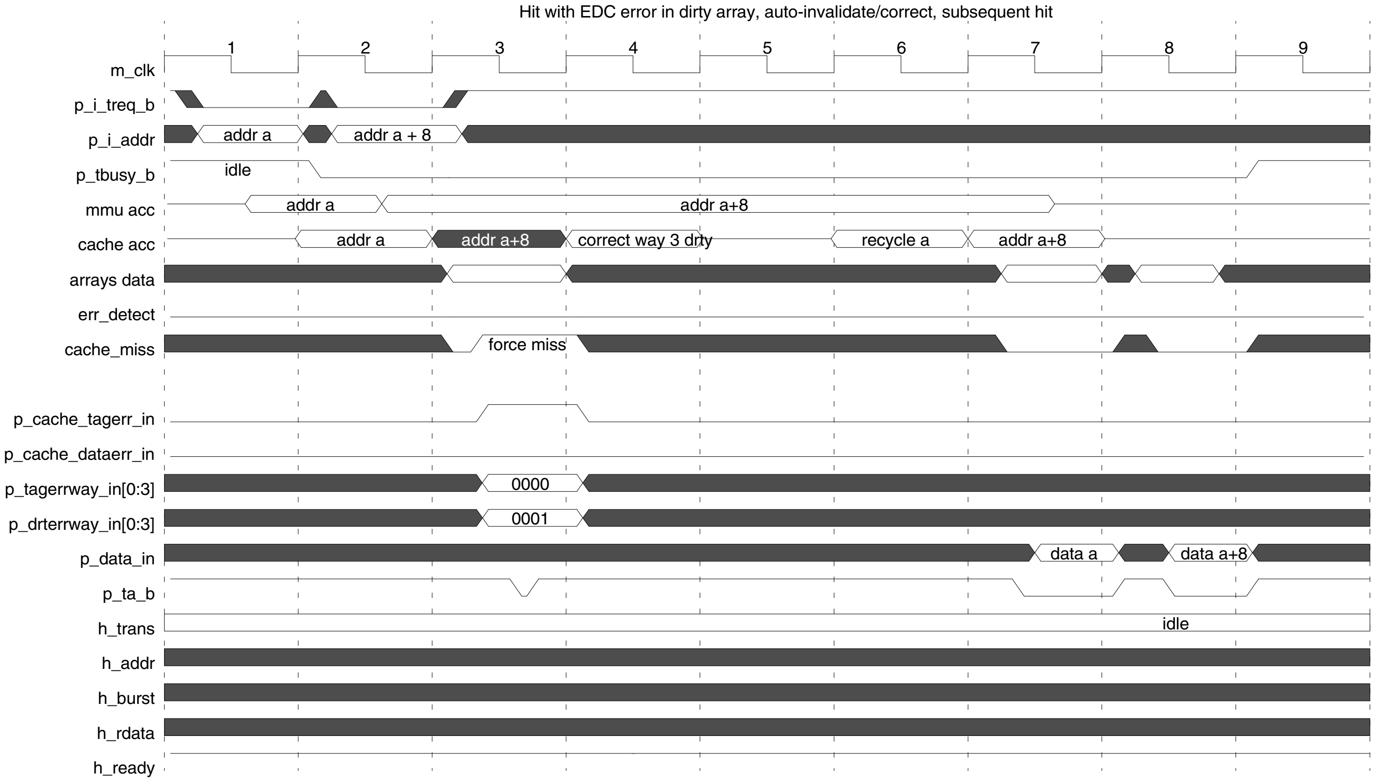


Figure 11-43. Cross-signaling Invalidation Input Operation—Dirty Error

A dirty array error is detected in cycle 3 in an external cache due to a difference in one or more of the three dirty bits for a cache line and results in a cache correction/invalidation operation being signaled. In this example, way 3 in the external cache has a dirty error, as indicated on the **p_drterrway_in[0:3]** inputs. The correction of the error is emulated in the receiving cache by rewriting all three dirty bits of the indicated way(s) (way 3 only in this example) to a ‘1’. The re-write of way 3 dirty bits in the dirty array is performed in cycle 4, and the address ‘a’ access is recycled in cycle 6 and results in a hit.

Note that way 3 is not indicated on the **p_tagerrway_in[0:3]** inputs since it is correctable. Also note that the result of this operation may be to cause a clean line to be marked as dirty, in order to replicate the state of the external cache. This can happen if the original state of the cache line is clean, but an error caused one or more of the dirty bits to be inadvertently set in the external cache for line(s) in the current set.

Figure 11-44 illustrates functional timing for a cross-signaling operation by a CPU which receives a cache error cross-signaling operation for both the tag array and in the dirty array. The error action indicates that an auto-invalidation/correction should be generated.

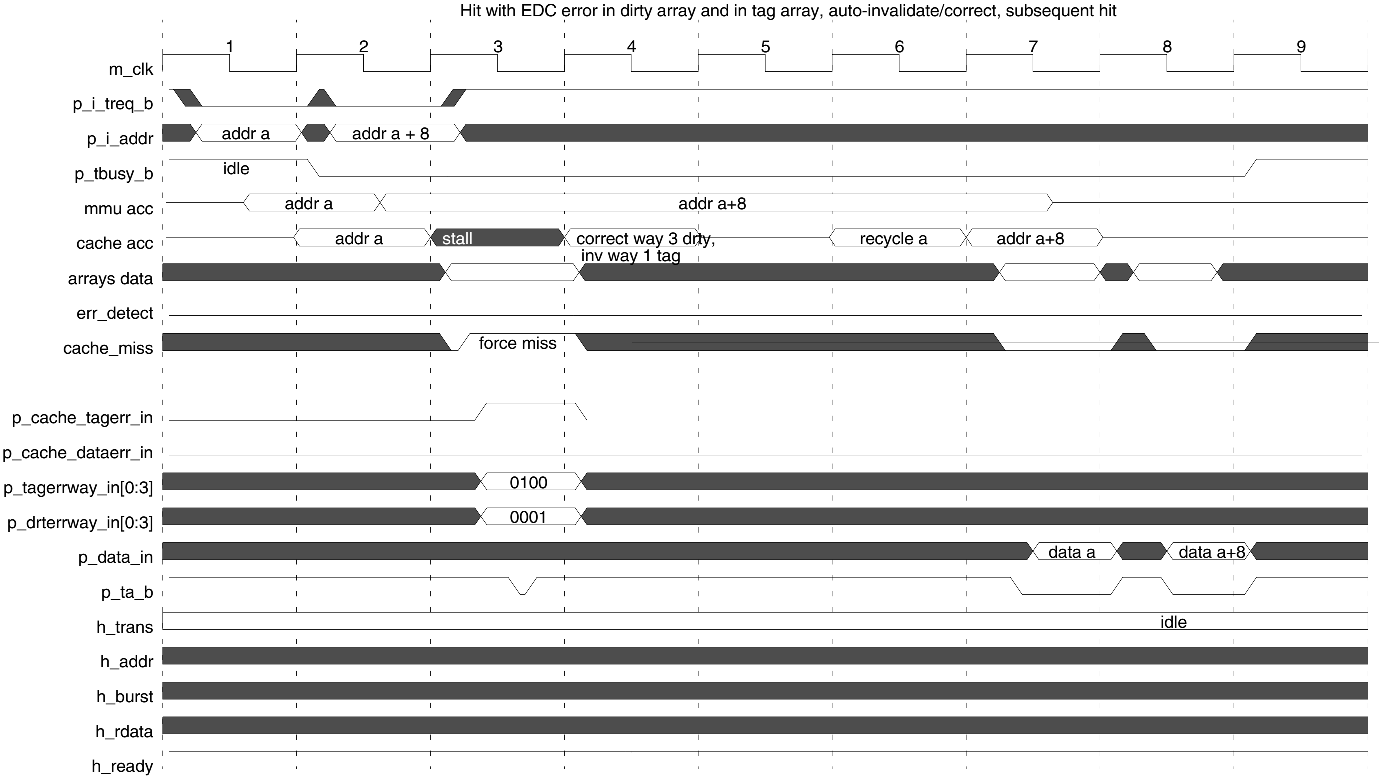


Figure 11-44. Cross-signaling Invalidation Input Operation—Tag Error, Dirty Error

The errors are detected in cycle 3 in an external cache due to a difference in one or more of the three dirty bits for a cache line and for the tag error, which results in a cache correction/invalidation cycle being forced. Way 3 has a dirty error, as indicated on the **p_drterrway_in[0:3]** inputs. Way 1 has an uncorrectable tag error and is auto-invalidated, as indicated on the **p_tagerrway_in[0:3]** inputs. The correction of the error is emulated in the receiving cache by a re-write of the dirty array for way 3 and the invalidation of the way 1 tag in cycle 4, and the address ‘a’ access is recycled in cycle 6 and results in a hit.

Note that way 3 is not indicated on the **p_tagerrway_in[0:3]** inputs because it is correctable.

Figure 11-45 illustrates functional timing for a cross-signaling operation by a CPU which receives a cache error cross-signaling operation for both the tag array, lock array, and the dirty array. The error action indicates that an auto-invalidation/correction should be generated.

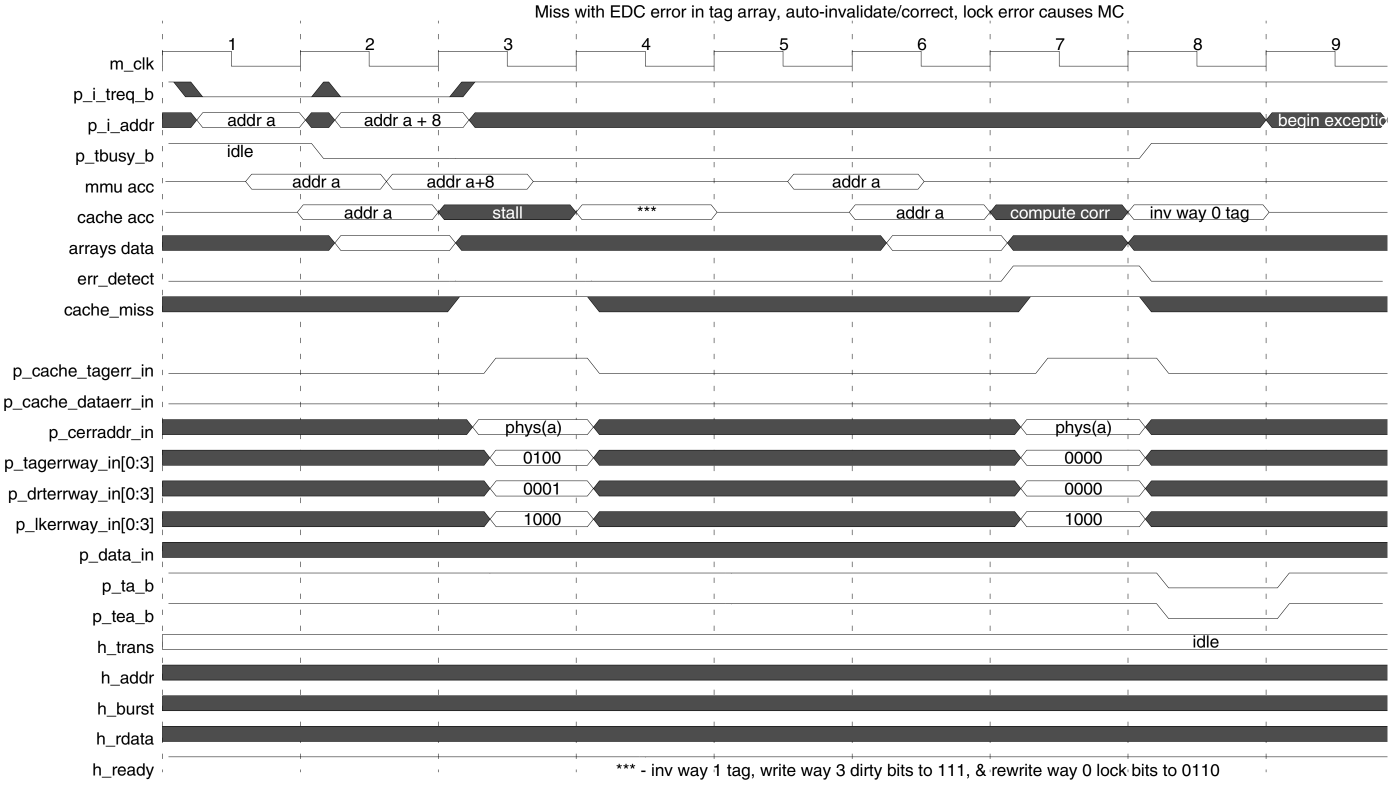


Figure 11-45. Cross-signaling Invalidation Input Operation—Tag Error, Dirty Error, and Lock Error

The errors are detected in cycle 3 in an external cache due to a difference in one or more of the three dirty bits for a cache line, and for the tag error and lock error, which results in a cache correction/invalidation cycle being forced. In this example, way 3 has a dirty error and is corrected by rewriting all three dirty bits to a 1. Way 1 has an uncorrectable tag error and is auto-invalidated. Way 0 has an uncorrectable lock error, which is signaled to cause the lock bits to be rewritten to 0110. The dirty error for way 4 is signaled on the **p_drterrway_in[0:3]** inputs, the lock error for way 0 is signaled on the **p_lkerrway_in[0:3]** inputs, and the invalidation for way 1 is signaled on the **p_tagerrway_in[0:3]** inputs.

Note that way 3 is not indicated on the **p_tagerrway_in[0:3]** outputs because it is correctable.

The re-write of way 3 dirty array, the rewrite of the way 0 lock bits to the double error value 0110, and the invalidation of the way 1 tag are emulated in the receiving cache in cycle 4. The address ‘a’ access is recycled in cycle 6 and results in a miss. The lock error is detected and re-signaled by both caches in cycle 7. Because of the miss, way 0 is invalidated in cycle 8, and a machine check is generated due to the invalidation of a locked way. Exception processing begins in cycle 9.

Figure 11-46 illustrates functional timing for a cross-signaling operation by a CPU that incurs a cache data parity error for a cache push operation.

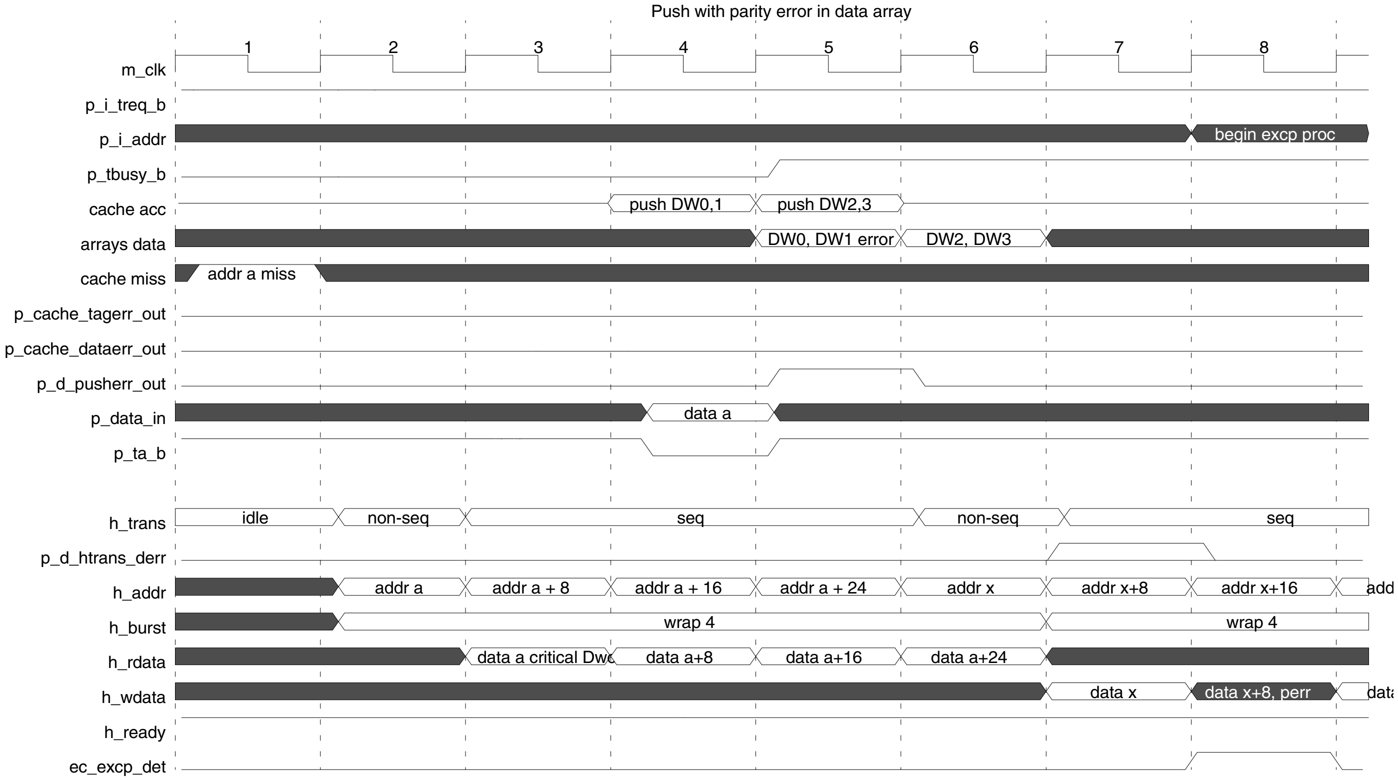


Figure 11-46. Cross-signaling Push Parity error Output Operation—Error on DW 1

A cache miss occurs for addr[a] in cycle 1, and a linefill operation is initiated. The linefill replaces a dirty line at addr[x], so a cache lookup for the dirty data is performed in cycles 4 and 5. DW1 has a data parity error that is detected in cycle 5. The push parity error is signaled on the **p_d_pusherr_out** output signal to an external cache for lockstep handshaking in cycle 5. The push begins in cycle 6 on the external bus. In cycle 7, during the address phase for double word 1, the **p_d_htrans_derr** output is asserted to indicate that the write data supplied for this beat of the push will contain erroneous data due to the parity error. The data for DW1 is driven in cycle 8. Also in cycle 8, exception processing for a machine check is initiated.

11.3.5 Cache Coherency Interface Operation

The cache coherency signaling interface is provided to support hardware cache coherency operations by the e200z760n3.

Figure 11-47 illustrates functional timing for a set of basic snoop request operations.

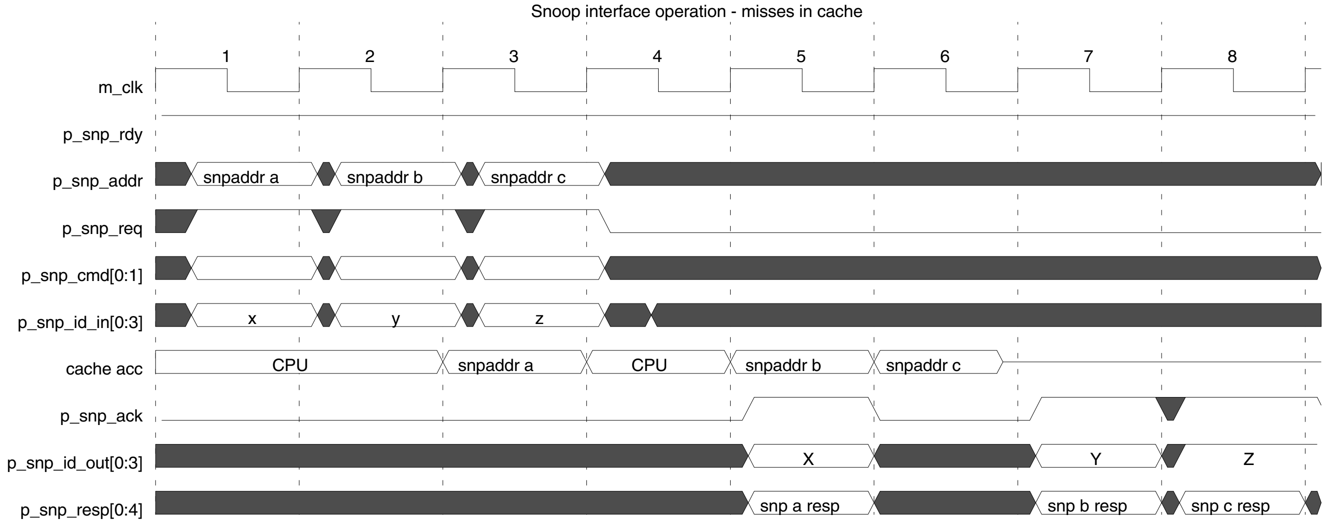


Figure 11-47. Basic Cache Coherency Interface Operation—Misses

Snoop requests are presented in cycles 1, 2, and 3, and enter the snoop queue. As requests are processed, they are acknowledged.

In this example, the snoops miss in the cache and require only a single cache access slot for lookup. The exact cycle the requests are acknowledged in varies and is not directly related to the cycle the requests occur.

Figure 11-48 illustrates functional timing for a snoop hit with invalidate. The exact cycle the requests are acknowledged in varies and is not directly related to the cycle the requests occur.

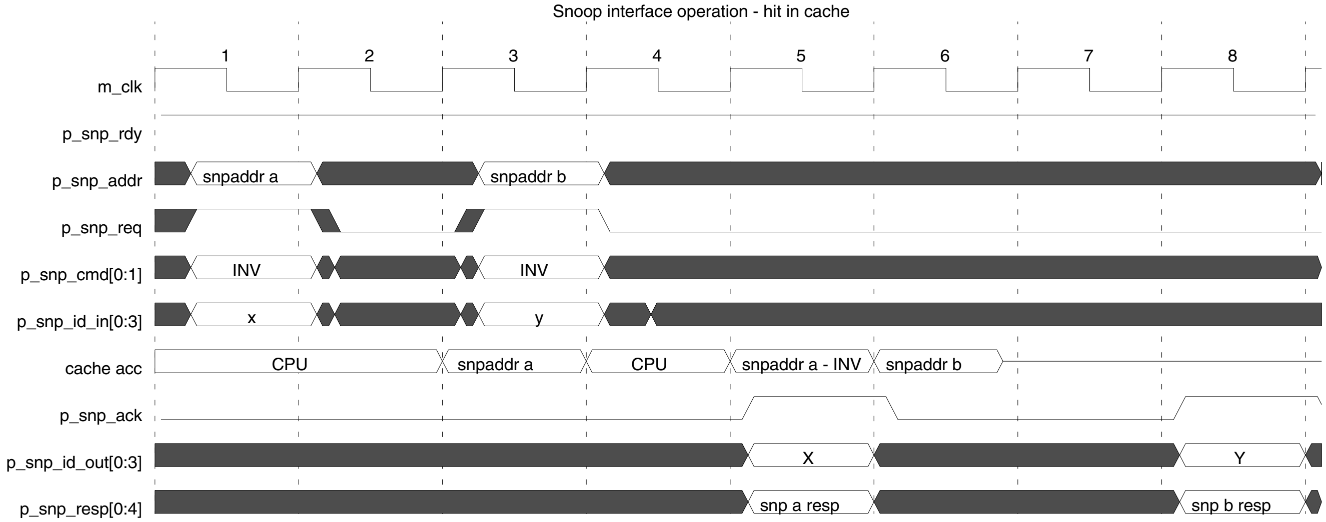


Figure 11-48. Basic Cache Coherency Interface Operation—Hit

Figure 11-49 illustrates another example of timing for a snoop request. This example shows the starvation control for a snoop which sits in the snoop queue until the snoop starvation counter expires due to blockage by a continuous stream of CPU requests.

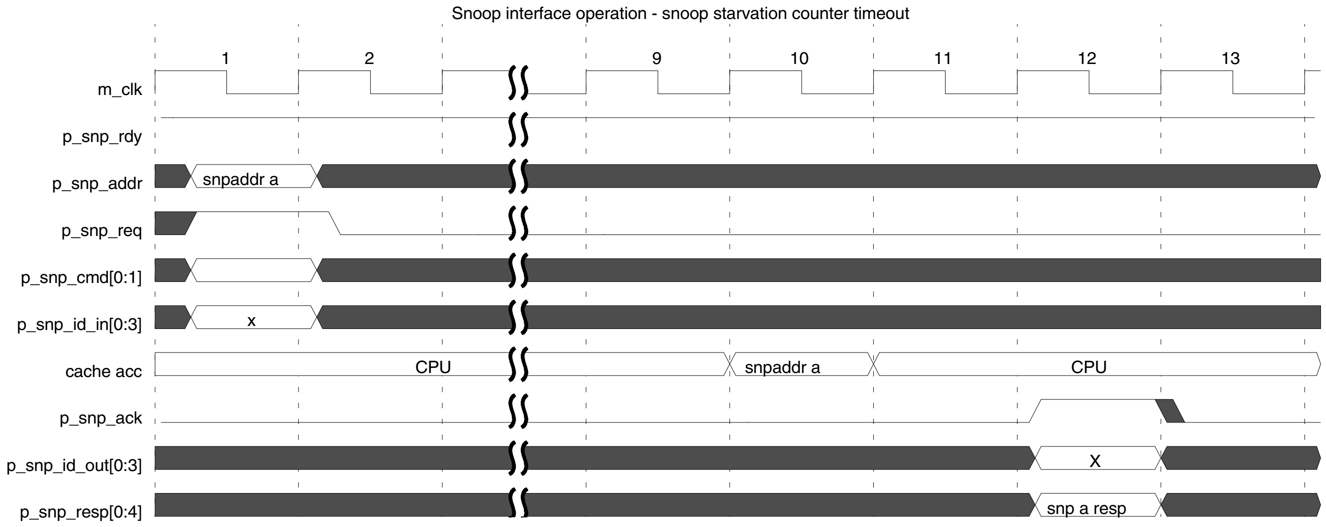


Figure 11-49. Cache Coherency Interface Operation—Snoop Starvation Timeout

Figure 11-50 illustrates operation of the **p_snp_rdy** output and snoop request acceptance.

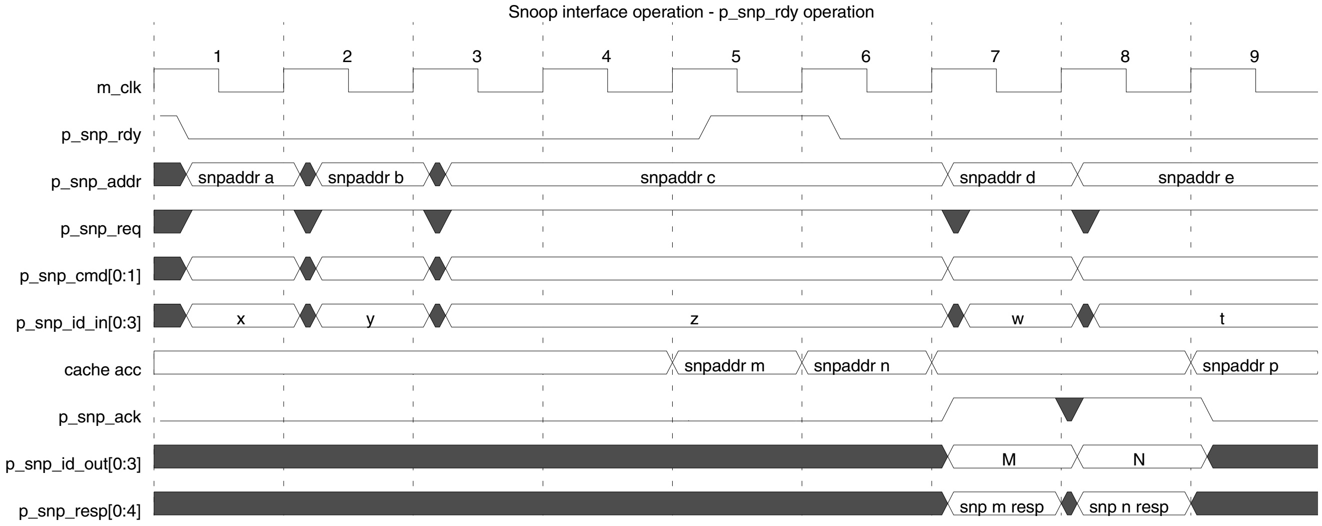


Figure 11-50. Cache Coherency Interface Operation—p_snp_rdy operation

In this example, **p_snp_rdy** is initially asserted, but in cycle 1 is negated due to the snoop queue filling. A snoop request for snpaddr ‘a’ is asserted in cycle 1. This request is taken and entered into the snoop queue at the end of cycle 1.

In cycle 2, **p_snp_rdy** is still negated, and a snoop request for snpaddr ‘b’ is presented. This request is also accepted and loaded into the snoop queue at the end of cycle 2 to allow for systems to use

p_snp_rdy from one CPU as a control qualifier to drive the **p_stall_bus_gwrite** input control of another CPU.

Following this, in cycle 3 another snoop request is presented for snpaddr ‘c’. This request is not accepted and must remain pending until the cycle after **p_snp_rdy** re-asserts to be recognized.

In cycle 5, **p_snp_rdy** is reasserted, indicating that the snoop queue can begin to store additional requests starting in the next cycle. Due to the protocol on **p_snp_rdy**, a minimum of two snoop queue entries must be available before **p_snp_rdy** can be re-asserted. Since a snoop request was pending at the end of cycle 4 (**p_snp_req** was asserted), the **p_snp_rdy** output will re-assert for one cycle once two free queue entries are available. The request for snpaddr ‘c’ will be queued at the end of cycle 6.

In cycle 6, **p_snp_rdy** is again negated, due to limited available snoop queue entries. This negation occurs in cycle 6 since **p_snp_rdy** was asserted during cycle 5 with only two free entries in the queue. When no pending snoop request is presented (**p_snp_req** is negated), **p_snp_rdy** will not be re-asserted until three queue entries are available. This is so that the **p_snp_rdy** signal does not alternate between asserted and negated, which must happen when only two queue entries are available. The re-assertion of **p_snp_rdy** in cycle 5 allows the pending request for snpaddr ‘c’ to be accepted at the end of cycle 6.

A new snoop request to snpaddr ‘d’ is made in cycle 7 and is accepted even though **p_snp_rdy** was negated in cycle 6, according to the **p_snp_rdy** protocol. A subsequent snoop request to snpaddr ‘e’ presented in cycle 8 must remain pending to be accepted until **p_snp_rdy** re-asserts after two free queue entries are once again available. Note that in cycles 5 and 6, earlier snoop requests to snpaddr ‘m’ and ‘n’ are processed, and the completion of these requests are signaled in cycles 7 and 8.

Figure 11-51 illustrates another example of operation of the **p_snp_rdy** output and snoop request acceptance.

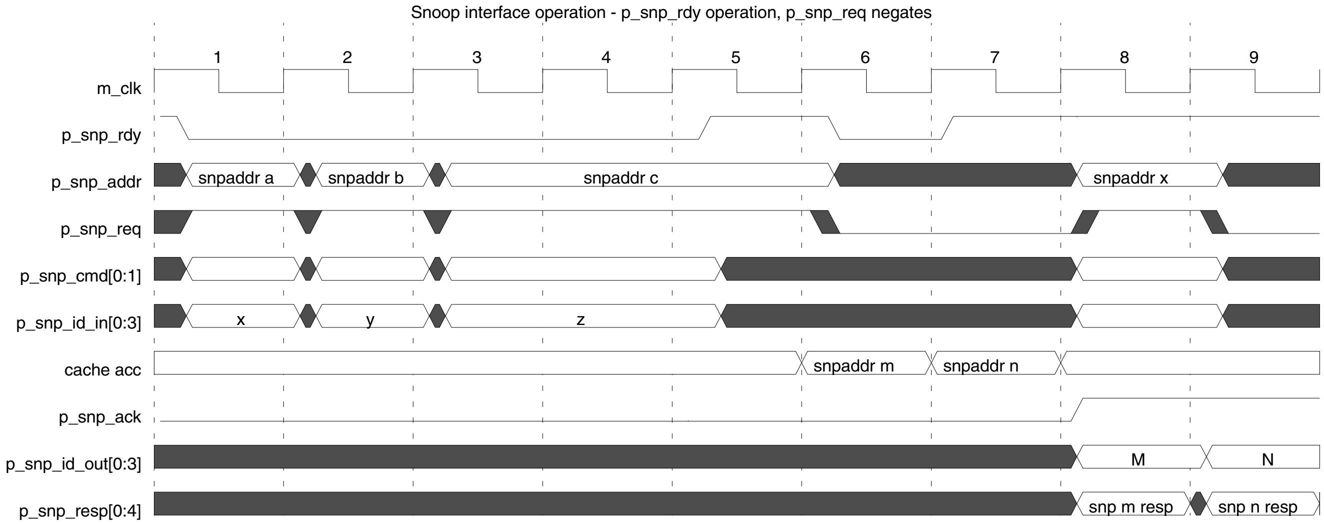


Figure 11-51. Cache Coherency Interface Operation—p_snp_rdy operation, p_snp_req negation prior to acceptance

In this example, **p_snp_rdy** is initially asserted, but in cycle 1 is negated due to the snoop queue filling. A snoop request for snpaddr ‘a’ is asserted in cycle 1. This request is taken and entered into the snoop queue at the end of cycle 1.

In cycle 2, **p_snp_rdy** is still negated, and a snoop request for snpaddr ‘b’ is presented. This request is also accepted and loaded into the snoop queue at the end of cycle 2 to allow for systems using **p_snp_rdy** from one CPU as a control qualifier to drive the **p_stall_bus_gwrite** input control of another CPU.

Following this, in cycle 3 another snoop request is presented for snpaddr ‘c’. This request is not accepted. It must remain pending until the cycle after **p_snp_rdy** re-asserts to be recognized.

In cycle 5, **p_snp_rdy** is reasserted, indicating that the snoop queue can begin to store additional requests starting in the next cycle. Due to the protocol on **p_snp_rdy**, a minimum of two snoop queue entries must be available before **p_snp_rdy** can be re-asserted. Since a snoop request was pending at the end of cycle 4 (**p_snp_req** was asserted), the **p_snp_rdy** output re-asserts for one cycle once two free queue entries are available. However, the request for snpaddr ‘c’ is not be queued at the end of cycle 6 because the request is no longer present.

In cycle 6, **p_snp_rdy** negates, since three queue entries have not yet become available. In cycle 7, **p_snp_rdy** can be re-asserted indicating at least three queue entries are available, and in cycle 8, a new snoop request is presented and accepted for snpaddr ‘x’. Note that in cycles 6 and 7, earlier snoop requests to snpaddr ‘m’ and ‘n’ are processed, freeing up the needed queue entries for the re-assertion of **p_snp_rdy**, and the completion of these requests are signaled in cycles 8 and 9.

Figure 11-52 illustrates another example of operation of the **p_snp_rdy** output and snoop request acceptance.

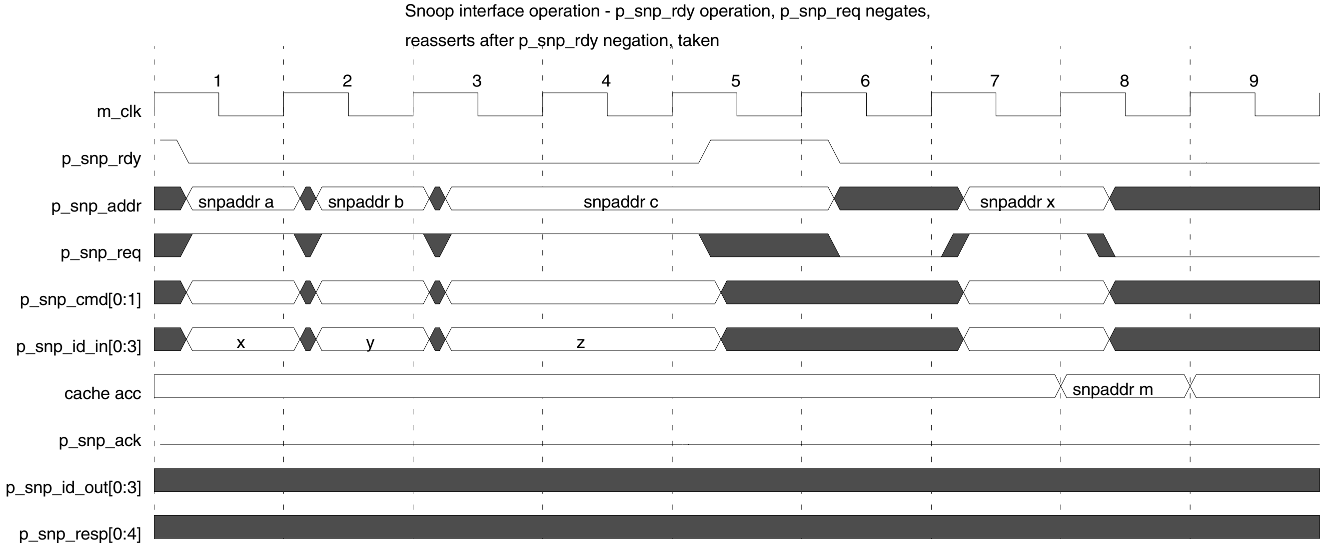


Figure 11-52. p_snp_rdy Operation, p_snp_req Negation Prior to Acceptance, Reasserted Later in Ready Window

In this example, **p_snp_rdy** is initially asserted, but in cycle 1 is negated due to the snoop queue filling. A snoop request for snpaddr ‘a’ is asserted in cycle 1. This request is taken and entered into the snoop queue at the end of cycle 1.

In cycle 2, **p_snp_rdy** is still negated, and a snoop request for snpaddr ‘b’ is presented. This request is also accepted and loaded into the snoop queue at the end of cycle 2, to allow for systems to use **p_snp_rdy** from one CPU as a control qualifier to drive the **p_stall_bus_gwrite** input control of another CPU.

Following this, in cycle 3 another snoop request is presented for snpaddr ‘c’. This request is not accepted. It must remain pending until the cycle after **p_snp_rdy** re-asserts to be recognized.

In cycle 5, **p_snp_rdy** is reasserted, indicating that the snoop queue can begin to store additional requests starting in the next cycle. Due to the protocol on **p_snp_rdy**, a minimum of two snoop queue entries must be available before **p_snp_rdy** can be re-asserted. Since a snoop request was pending at the end of cycle 4 (**p_snp_req** was asserted), the **p_snp_rdy** output re-asserts for one cycle once two free queue entries are available. However, the request for snpaddr ‘c’ is not queued at the end of cycle 6 because the request is no longer present.

In cycle 6, **p_snp_rdy** negates, since three queue entries have not yet become available. In cycle 8, a new snoop request is presented and accepted for snpaddr ‘x’. Note that since the **p_snp_rdy** output was asserted in cycle 5, a snoop request present in ether or both of cycles 6 and 7 will be accepted as per the protocol.

11.3.5.1 Stop Mode Entry/Exit and Snoop Ready Signaling

When a request is made to enter stop mode via the assertion of **p_stop**, the **p_snp_rdy** output is negated. While the core complex is in the stopped (power-down) state, bus snooping is disabled, and the **p_snp_rdy** output is held negated. Snoop requests will be processed around the assertion of the stop mode entry request (assertion of **p_stop**) per the normal protocol associated with **p_snp_rdy** negation, including acceptance of a snoop request during a small interval around **p_snp_rdy** negation, thus additional snoop operations may need to occur prior to entering the stopped state. All snoop queue entries are processed prior to the assertion of **p_stopped**.

Figure 11-53 illustrates an example of operation of the **p_snp_rdy** output when entering the stopped state and snoop request acceptance.

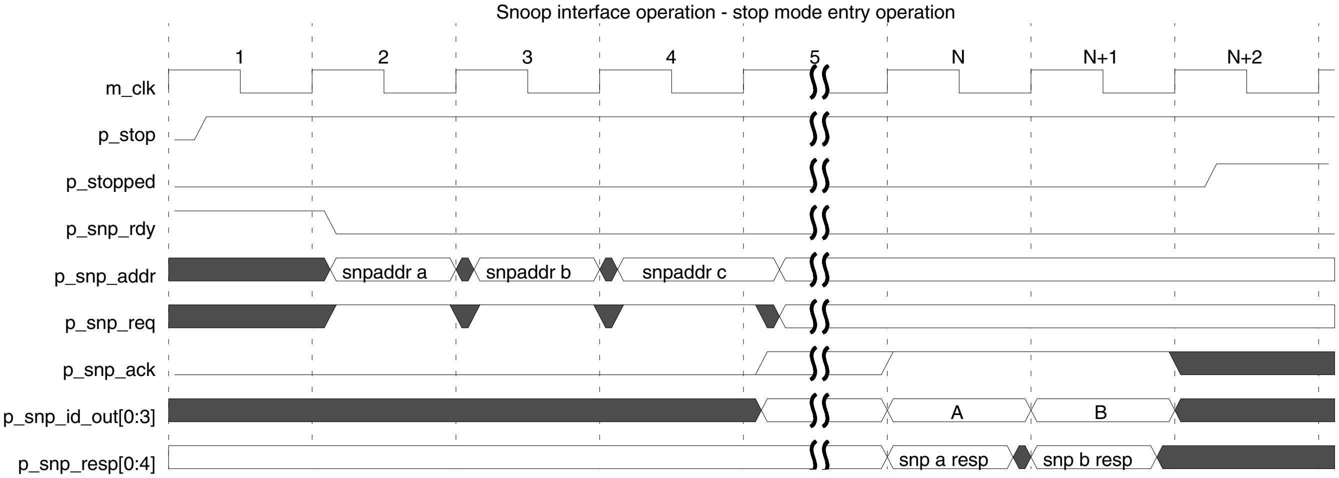


Figure 11-53. Stop mode Entry, p_snp_rdy Operation

In cycle 1, p_stop is asserted, indicating a request to enter the stopped state. In cycle 2 the **p_snp_rdy** signal negates due to the stop request. Snoop requests for snpaddr ‘a’ and ‘b’ are taken in cycles 2 and 3 according to the **p_snp_rdy** protocol, although the system logic should typically stop generating new requests based on the **p_stop** input assertion. The request for snpaddr ‘c’ is not taken in cycle 4, again based on the snoop ready protocol.

In cycle(s) 5, the snoop control logic continues to process any previously queued snoop requests, and in cycle N, and N+1, the final snoop responses for snoops A and B occur. Following the snoop responses for these final queued snoop requests, **p_stopped** asserts in cycle N+2. No further snoop requests will be accepted while the CPU is stopped.

Figure 11-54 illustrates an example of operation of the **p_snp_rdy** output when exiting the stopped state and snoop request acceptance.

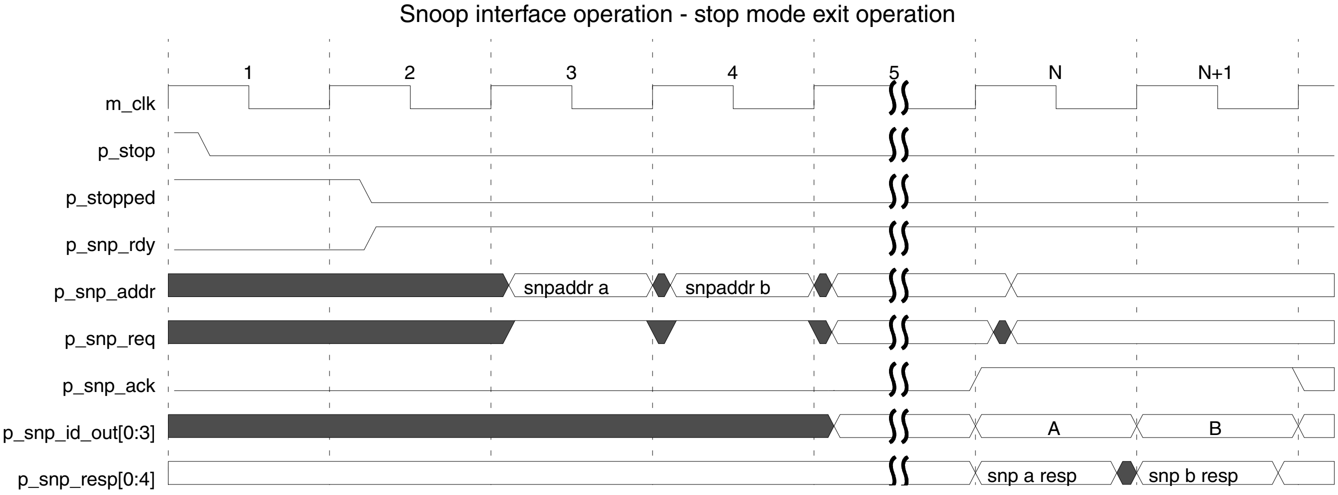


Figure 11-54. Stop Mode Exit, p_snp_rdy Operation

In cycle 1, p_stop is negated, indicating a request to exit the stopped state. In cycle 2, the **p_stopped** output signal negates due to the negated stop request. Also in cycle 2, the p_snp_rdy output is asserted, indicating that snoop requests will begin to be accepted on the next clock cycle. Snoop requests for snpaddr 'a' and 'b' are taken in cycles 3 and 4 according to the **p_snp_rdy** protocol. In cycle N and N+1, the snoop responses for snoops A and B occur.

11.3.6 Debug Lockstep Cross-signaling Operation

The debug lockstep cross-signaling interface is provided to allow lockstep operation of two or more CPUs. It is used when performing external debug mode (EDM) operations in which the CPUs must maintain lockstep operation in the presence of asynchronous debug operations that cause the CPU to enter or exit a debug halted mode. The interface permits signaling that a debug request has been received and that other CPUs in lockstep operation should emulate the same debug-entry point. Similar signaling is provided for exiting debug mode, either in response to a single-step operation (a go + noexit OCMD operation) or in response to exiting debug mode back to normal operating mode (go + exit OCMD operation). Because the debug logic associated with the OnCE JTAG controller operates asynchronously to the processor **m_clk** clock, the exact edge on which a debug request generated from the OnCE **tclk** domain is recognized is not always deterministic, and the same issue exists when exiting debug mode via a **tclk** domain generated OCMD “go” command.

In addition, debug lockstep cross-signaling is provided to handshake updates to the Nexus 3 control registers such that various aspects of Nexus 3 are controlled in a lockstep fashion. This is done by providing handshaking of synchronized Update_DR TAP controller states, so that register updates due to entering the Update_DR state are delayed until the Update_DR state has been seen by all lockstep processors. Because the OnCE JTAG controller operates asynchronously to the processor **m_clk** clock, the exact edge on which an Update_DR state generated from the OnCE **tclk** domain is recognized is not always deterministic. The cross-signaling interface provides a way to ensure lockstep updates of register resources.

11.3.6.1 Debug Entry Cross-signaling

Figure 11-55 illustrates functional timing for debug entry cross-signaling operation with lockstep operation disabled.

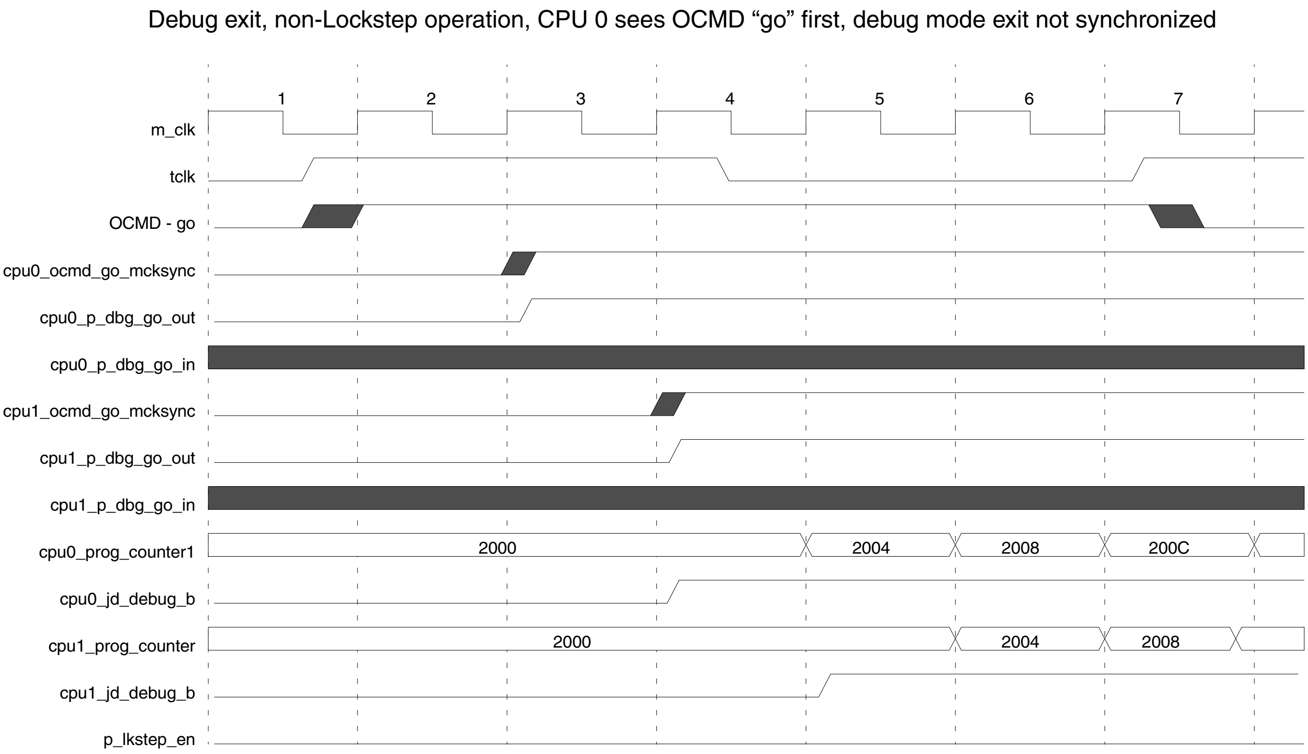


Figure 11-55. Debug Entry Cross-Signaling Interface, non-Lockstep Mode

In this example, entry into debug mode is requested by setting OCR[DR] simultaneously in CPU0 and CPU1. The OCR register is updated in the Update_DR state by the OnCE controller, using **tclk** clocking, and the value of the DR bit is synchronized to the **m_clk** clock domain in each processor. Since the relationship between **tclk** and **m_clk** is not fixed, it is possible for the synchronized version of the DR bit to differ in the two CPUs. In the example in the timing diagram, the DR bit is updated at the rise of **tclk**, and the synchronized version (**ocr_dr_mcksync**) in CPU0 is asserted in clock cycle 3. Due to differences in synchronizer outputs, the version of this signal in CPU1 is not seen asserted until clock cycle 4. Since the lockstep control signal **p_lkstep_en** is not asserted for this example, the cross-signaling interface signals **cpu0_p_dbgrq_edm_in** and **cpu1_p_dbgrq_edm_in** are ignored and do not condition the entry into debug mode by the CPUs. CPU0 enters debug mode in cycle 4 (**cpu0_jd_debug_b** asserted) with a program counter value of 100C, while CPU1 enters debug mode in cycle 5 (**cpu1_jd_debug_b** asserted) with a program counter value of 1010. The two CPUs are thus no longer in sync.

Figure 11-56 illustrates functional timing for debug entry cross-signaling operation with lockstep operation enabled.

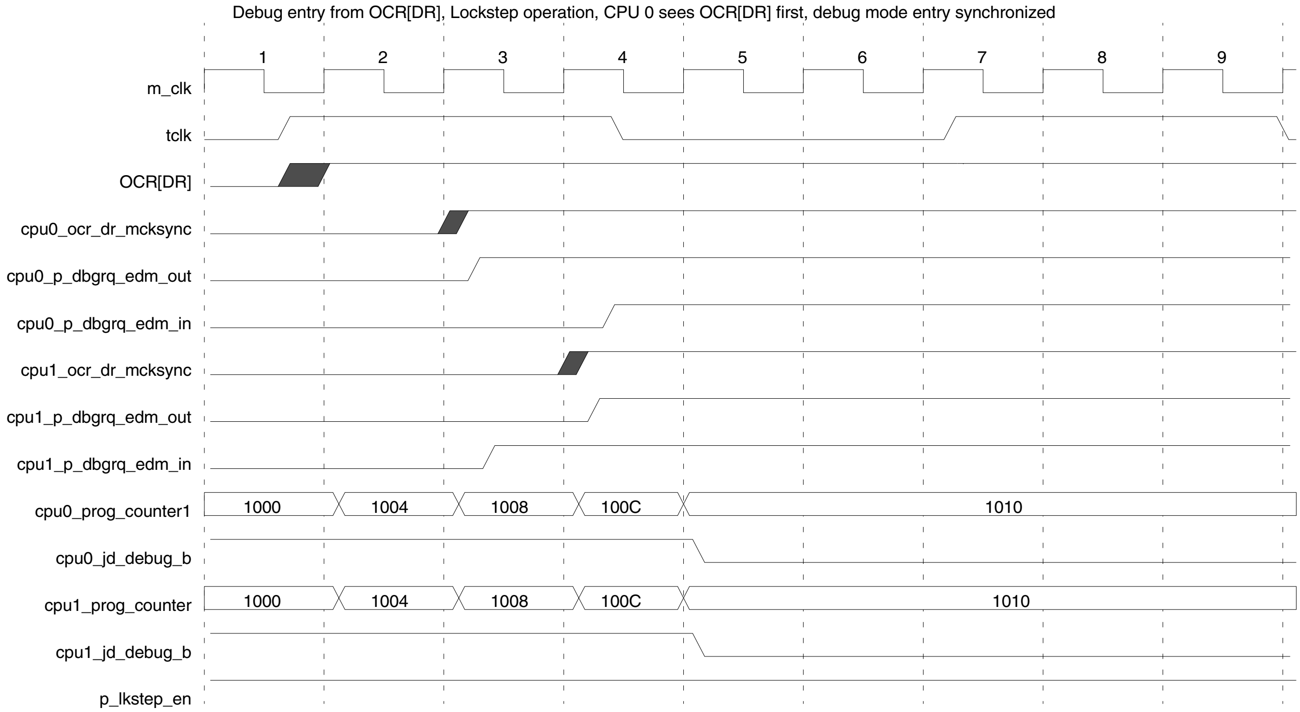


Figure 11-56. Debug Entry Cross-Signaling Interface, lockstep mode

In this example, entry into debug mode is requested by setting OCR[DR] simultaneously in CPU0 and CPU1. The OCR register is updated in the Update_DR state by the OnCE controller, using **tclk** clocking, and the value of the DR bit is synchronized to the **m_clk** clock domain in each processor. Because the relationship between **tclk** and **m_clk** is not fixed, it is possible for the synchronized version of the DR bit to differ in the two CPUs. The DR bit is updated at the rise of **tclk**, and the synchronized version (**ocr_dr_mcksync**) in CPU0 is asserted in clock cycle 3. Due to differences in synchronizer outputs, the version of this signal in CPU1 is not seen asserted until clock cycle 4.

Because the lockstep control signal **p_lkstep_en** is asserted, the cross-signaling interface signals **cpu0_p_dbgrq_edm_in** and **cpu1_p_dbgrq_edm_in** are used to handshake entry into debug mode and condition the entry into debug mode by the CPUs. Based on the internal recognition of the asserted DR bit (**cpu0_ocr_dr_mcksync**) in cycle 3, CPU0 output **cpu0_p_dbgrq_edm_out** is asserted in cycle 3 and drives the corresponding input signal **cpu1_p_dbgrq_edm_in** of CPU1 in cycle 3. Since CPU0 does not have an asserted **cpu0_p_dbgrq_edm_in** signal, debug entry is delayed. Based on the internal recognition of the asserted DR bit (**cpu1_ocr_dr_mcksync**) in cycle 4, CPU1 output **cpu1_p_dbgrq_edm_out** is asserted in cycle 4 and drives the corresponding input signal **cpu0_p_dbgrq_edm_in** of CPU0 in cycle 4.

At this point, both CPUs have received the proper cross-signaling handshakes to allow synchronized entry into debug mode. CPU0 and CPU1 both enter debug mode in cycle 5 (**cpu0,1_jd_debug_b** asserted) with a program counter value of 1010. The two CPUs are thus properly in sync.

Figure 11-57 illustrates functional timing for debug entry cross-signaling operation with lockstep operation enabled.

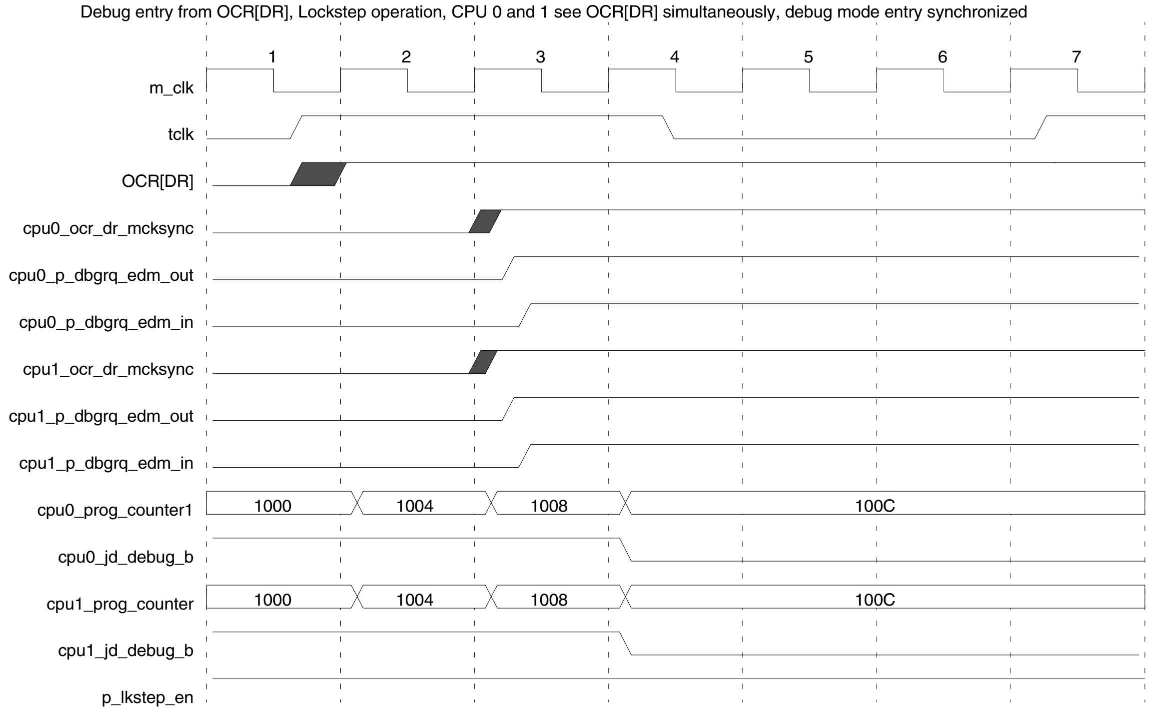


Figure 11-57. Debug Entry Cross-Signaling Interface, lockstep mode (2)

In this example, entry into debug mode is requested by setting OCR[DR] simultaneously in CPU0 and CPU1. The OCR register is updated in the Update_DR state by the OnCE controller, using **tclk** clocking, and the value of the DR bit is synchronized to the **m_clk** clock domain in each processor. Since the relationship between **tclk** and **m_clk** is not fixed, it is possible for the synchronized version of the DR bit to differ in the two CPUs. In the example in the timing diagram, the DR bit is updated at the rise of **tclk**, and the synchronized version (**ocr_dr_mcksync**) in CPU0 and in CPU1 is asserted in clock cycle 3. Since the lockstep control signal **p_lkstep_en** is asserted for this example, the cross-signaling interface signals **cpu0_p_dbgrq_edm_in** and **cpu1_p_dbgrq_edm_in** are used to handshake entry into debug mode, and condition the entry into debug mode by the CPUs.

Based on the internal recognition of the asserted DR bit (**cpu0_ocr_dr_mcksync**) in cycle 3, CPU0 output **cpu0_p_dbgrq_edm_out** is asserted in cycle 3 and drives the corresponding input signal **cpu1_p_dbgrq_edm_in** of CPU1 in cycle 3. Similarly, in cycle 3, CPU1 output **cpu1_p_dbgrq_edm_out** is asserted and drives the corresponding input signal **cpu0_p_dbgrq_edm_in** of CPU0 in cycle 3. Since CPU0 has an asserted **cpu0_p_dbgrq_edm_in** signal, debug entry is not delayed. Based on the internal recognition of the asserted DR bit (**cpu1_ocr_dr_mcksync**) in cycle 3, CPU1 output **cpu1_p_dbgrq_edm_out** is asserted in cycle 3 and drives the corresponding input signal **cpu0_p_dbgrq_edm_in** of CPU0 in cycle 3.

At this point, both CPUs have received the proper cross-signaling handshakes to allow synchronized entry into debug mode. CPU0 and CPU1 both enter debug mode in cycle 4 (**cpu0,1_jd_debug_b** asserted) with a program counter value of 100C. The two CPUs are thus properly in sync.

11.3.6.2 Debug Exit Cross-signaling

Figure 11-58 illustrates functional timing for debug exit cross-signaling operation with lockstep operation disabled.

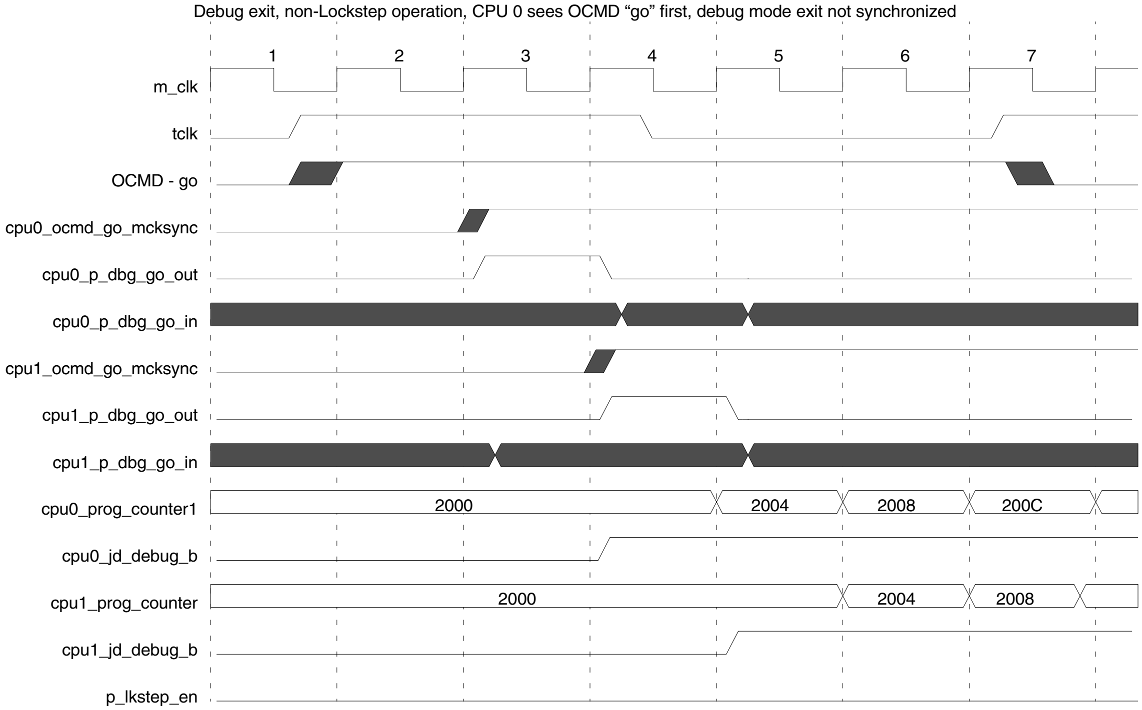


Figure 11-58. Debug Exit Cross-Signaling Interface, non-lockstep mode

In this example, exit from debug mode is requested by setting OCMD[GO] simultaneously in CPU0 and CPU1. The OCMD register is updated in the Update_DR state by the OnCE controller, using **tclk** clocking, and the value of the GO bit is synchronized to the **m_clk** clock domain in each processor. Since the relationship between **tclk** and **m_clk** is not fixed, it is possible for the synchronized version of the GO bit to differ in the two CPUs. The GO bit is updated at the rise of **tclk** in cycle 1, and the synchronized version (**ocmd_go_mcksync**) in CPU0 is asserted in clock cycle 3. Due to differences in synchronizer outputs, the version of this signal in CPU1 is not seen asserted until clock cycle 4. Since the lockstep control signal **p_lkstep_en** is not asserted, the cross-signaling interface signals **cpu0_p_dbg_go_in** and **cpu1_p_dbg_go_in** are ignored and do not condition the exit from debug mode by the CPUs.

CPU0 exits debug mode in cycle 4 (**cpu0_jd_debug_b** negated) and begins execution, while CPU1 exits debug mode in cycle 5 (**cpu1_jd_debug_b** negated). The two CPUs are thus no longer in sync.

Figure 11-59 illustrates functional timing for debug exit cross-signaling operation with lockstep operation enabled.

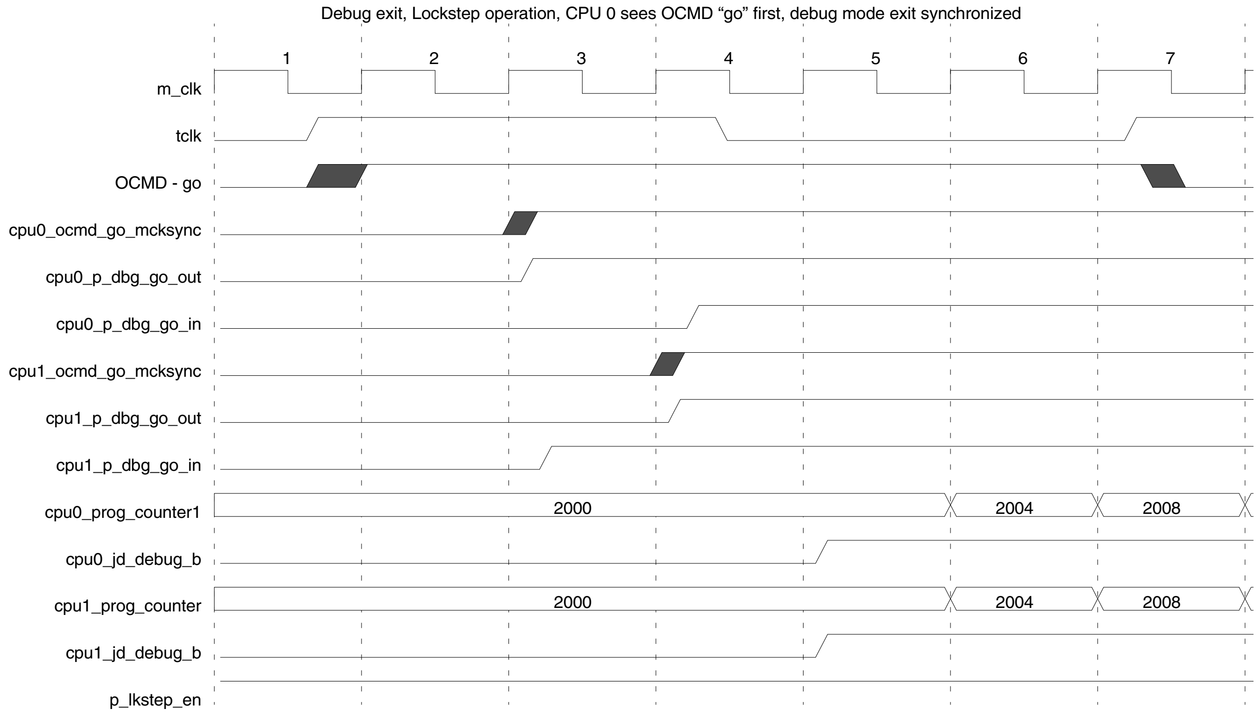


Figure 11-59. Debug Exit Cross-Signaling Interface, lockstep mode

In this example, exit from debug mode is requested by setting OCMD[GO] simultaneously in CPU0 and CPU1. The OCMD register is updated in the Update_DR state by the OnCE controller, using **tclk** clocking, and the value of the GO bit is synchronized to the **m_clk** clock domain in each processor. Since the relationship between **tclk** and **m_clk** is not fixed, it is possible for the synchronized version of the GO bit to differ in the two CPUs. In this example, the GO bit is updated at the rise of **tclk** in cycle 1, and the synchronized version (**ocmd_go_mcksync**) in CPU0 is asserted in clock cycle 3. Due to differences in synchronizer outputs, the version of this signal in CPU1 is not seen asserted until clock cycle 4.

Since the lockstep control signal **p_lkstep_en** is asserted for this example, the cross-signaling interface signals **cpu0_p_dbg_go_in** and **cpu1_p_dbg_go_in** are used to qualify exiting debug mode. CPU0 signals an exit condition in cycle 3 by asserting **cpu0_p_dbg_go_out** which drives the **cpu1_p_dbg_go_in** input of CPU1. Since CPU0’s **cpu0_p_dbg_go_in** input is not yet asserted, CPU0 delays exiting debug mode. CPU1 signals an exit condition in cycle 4 by asserting **cpu1_p_dbg_go_out** which drives the **cpu0_p_dbg_go_in** input of CPU0.

Because CPU0 and CPU1 now have their respective **p_dbg_go_in** input asserted, exiting from debug mode may now proceed. CPU0 and CPU1 exit debug mode in cycle 5 (**jd_debug_b** negated) and being execution. The two CPUs are thus kept in sync.

Figure 11-60 illustrates functional timing for debug exit cross-signaling operation with lockstep operation enabled.

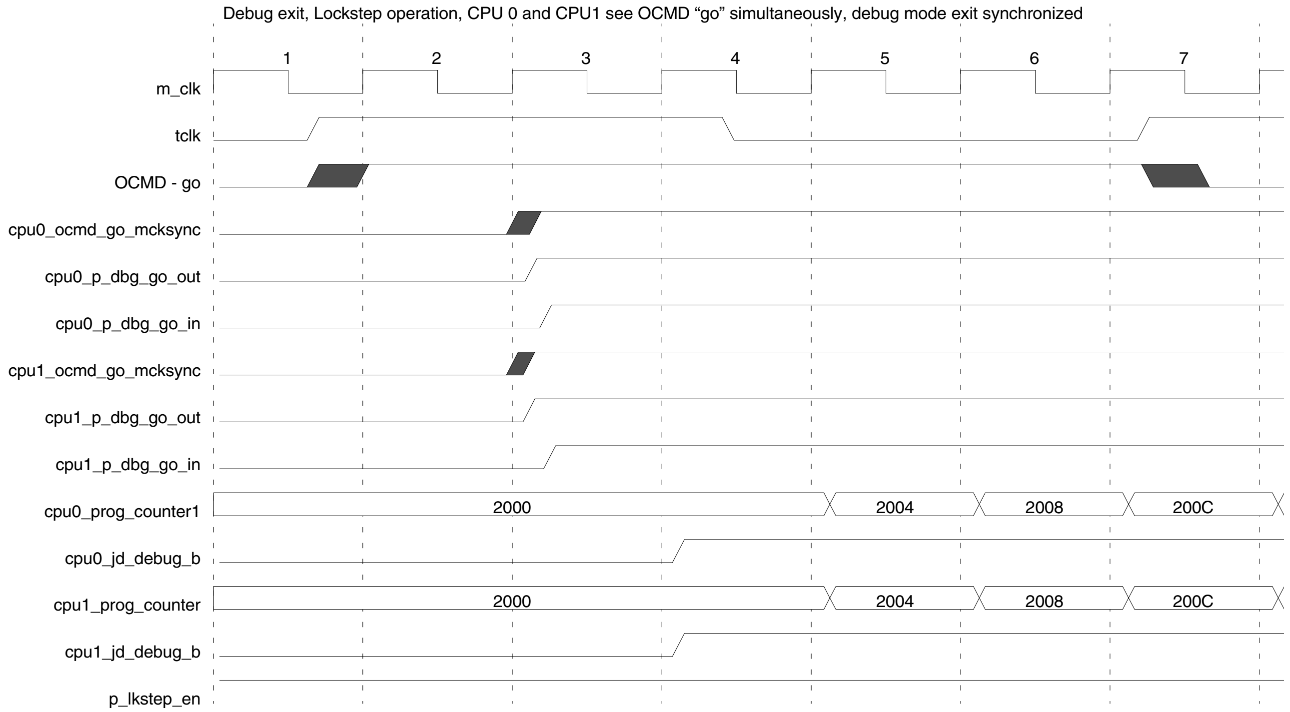


Figure 11-60. Debug Exit Cross-Signaling Interface, lockstep mode (2)

In this example, exit from debug mode is requested by setting OCMD[GO] simultaneously in CPU0 and CPU1. The OCMD register is updated in the Update_DR state by the OnCE controller, using **tclk** clocking, and the value of the GO bit is synchronized to the **m_clk** clock domain in each processor. Since the relationship between **tclk** and **m_clk** is not fixed, it is possible for the synchronized version of the GO bit to differ in the two CPUs. The GO bit is updated at the rise of **tclk** in cycle 1, and the synchronized version (**ocmd_go_mcksync**) in CPU0 and CPU1 is asserted in clock cycle 3.

Because the lockstep control signal **p_lkstep_en** is asserted, the cross-signaling interface signals **cpu0_p_dbg_go_in** and **cpu1_p_dbg_go_in** are used to qualify exiting debug mode. CPU0 signals an exit condition in cycle 3 by asserting **cpu0_p_dbg_go_out** which drives the **cpu1_p_dbg_go_in** input of CPU1. CPU1 also signals an exit condition in cycle 3 by asserting **cpu1_p_dbg_go_out** which drives the **cpu0_p_dbg_go_in** input of CPU0.

CPU0 and CPU1 now have their respective **p_dbg_go_in** input asserted and exiting from debug mode may proceed. CPU0 and CPU1 exit debug mode in cycle 4 (**jd_debug_b** negated) and being execution. The two CPUs are kept in sync.

11.3.6.3 Update_DR State Cross-signaling

Figure 11-61 illustrates functional timing for Update_DR cross-signaling operation with lockstep operation enabled.

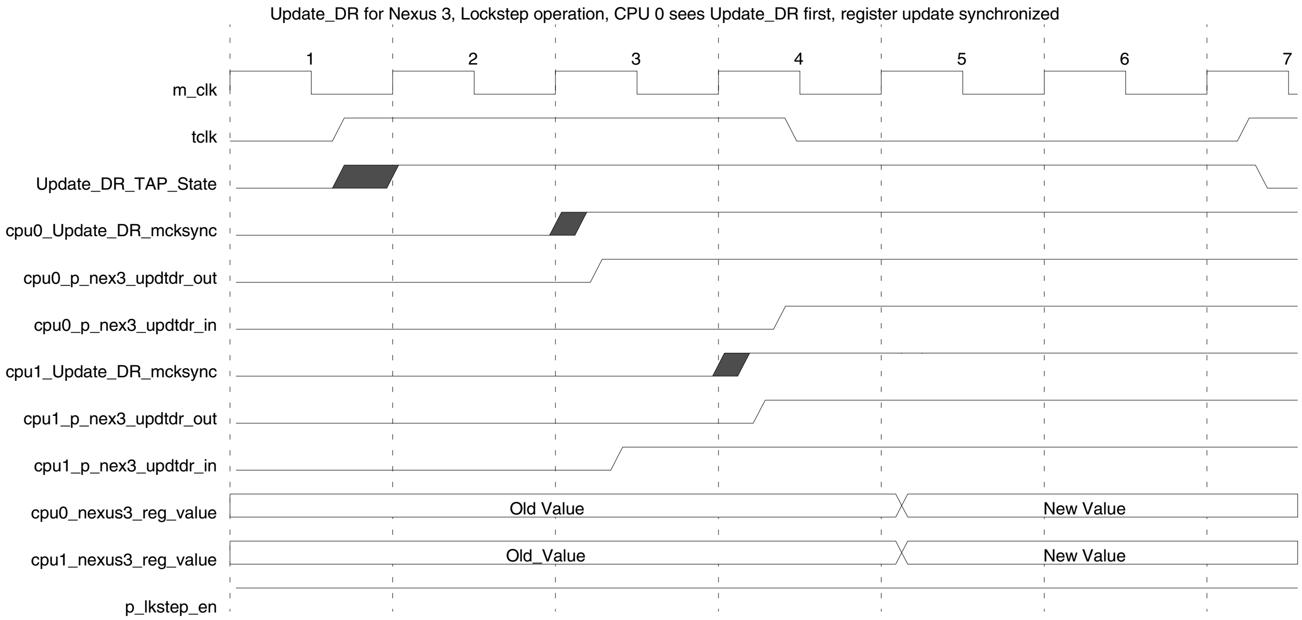


Figure 11-61. Debug Update_DR State Cross-Signaling Interface, lockstep mode

In this example, an update to one of the Nexus 3 register is requested by entering the Update_DR state simultaneously in CPU0 and CPU1 in the **tclk** domain. The Update_DR state is reached by the OnCE controller, using **tclk** clocking, and the Update_DR state is synchronized to the **m_clk** clock domain in each processor.

Because the relationship between **tclk** and **m_clk** is not fixed, it is possible for the synchronized version of the Update_DR state to differ in the two CPUs. The Update_DR state is reached at the rise of **tclk**, and the synchronized version in CPU0 is asserted in clock cycle 3. Due to differences in synchronizer outputs, the version of this signal in CPU1 is not asserted until clock cycle 4.

Because the lockstep control signal **p_lkstep_en** is asserted, the cross-signaling interface signals **cpu0_p_nex3_updtdr_in** and **cpu1_p_nex3_updtdr_in** are used to handshake actual register updates by the CPUs. Based on the internal recognition of the synchronized version of the asserted Update_DR state in cycle 3, CPU0 output **cpu0_p_nex3_updtdr_out** is asserted in cycle 3 and drives the corresponding input signal **cpu1_p_nex3_updtdr_in** of CPU1 in cycle 3. Since CPU0 does not have an asserted **cpu0_p_nex3_updtdr_in** signal, the Nexus 3 register update is delayed. Based upon reaching the synchronized version of the Update_DR state in cycle 4, CPU1 output **cpu1_p_nex3_updtdr_out** is asserted in cycle 4 and drives the corresponding input signal **cpu0_p_nex3_updtdr_in** of CPU0 in cycle 4.

At this point, both CPUs have received the proper cross-signaling handshakes to allow the Nexus 3 register update to occur. CPU0 and CPU1 both update the Nexus 3 register in cycle 5. The two CPUs are properly in sync.

Figure 11-62 illustrates functional timing for Update_DR cross-signaling operation with lockstep operation enabled.

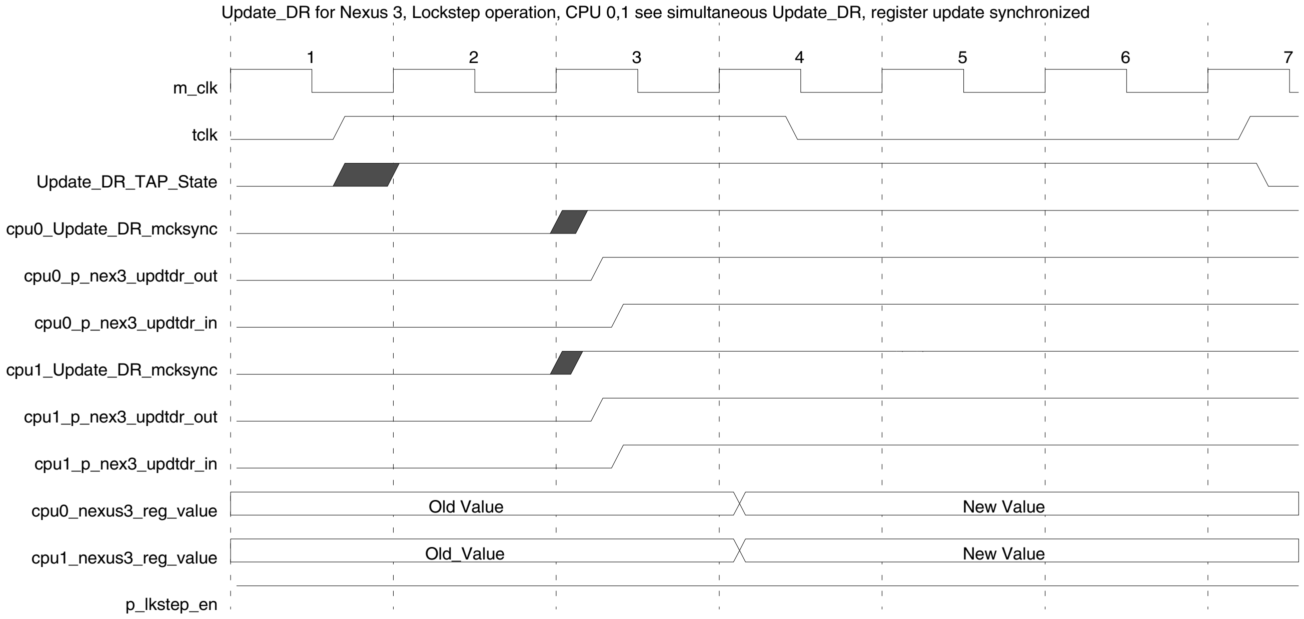


Figure 11-62. Debug Update_DR State Cross-Signaling Interface, lockstep mode (2)

In this example, an update to one of the Nexus 3 register is requested by entering the Update_DR state simultaneously in CPU0 and CPU1 in the **tclk** domain. The Update_DR state is reached by the OnCE controller, using **tclk** clocking, and the Update_DR state is synchronized to the **m_clk** clock domain in each processor. Since the relationship between **tclk** and **m_clk** is not fixed, it is possible for the synchronized version of the Update_DR state to differ in the two CPUs. The Update_DR state is reached at the rise of **tclk**, and the synchronized version in CPU0 is asserted in clock cycle 3. The version of this signal in CPU1 is also asserted in clock cycle 3.

Because the lockstep control signal **p_lkstep_en** is asserted, the cross-signaling interface signals **cpu0_p_nex3_updtdr_in** and **cpu1_p_nex3_updtdr_in** are used to handshake actual register updates by the CPUs. Based on the internal recognition of the synchronized version of the asserted Update_DR state in cycle 3, CPU0 output **cpu0_p_nex3_updtdr_out** is asserted in cycle 3 and drives the corresponding input signal **cpu1_p_nex3_updtdr_in** of CPU1 in cycle 3. Based upon reaching the synchronized Update_DR state in cycle 3, CPU1 output **cpu1_p_nex3_updtdr_out** is asserted in cycle 3 and drives the corresponding input signal **cpu0_p_nex3_updtdr_in** of CPU0 in cycle 3.

At this point, both CPUs have received the proper cross-signaling handshakes to allow the Nexus 3 register update to occur. CPU0 and CPU1 both update the Nexus 3 register in cycle 4. The two CPUs are thus properly in sync.

11.3.7 Power Management

Figure 11-63 shows the relationship of the wakeup control signal **p_wakeup** to the relevant input signals.

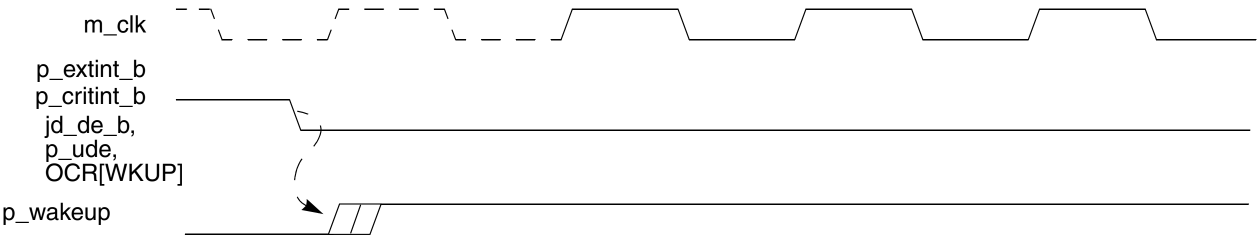


Figure 11-63. Wakeup Control Signal (p_wakeup)

11.3.8 Interrupt Interface

The following diagram shows the relationship of the interrupt input signals to the CPU clock. The **p_avec_b**, **p_extint_b**, **p_critint_b** and **p_voffset[0:15]** inputs as well as the **p_nmi_b** input must meet setup and hold timing relative to the rising edge of the **m_clk**. In addition, during each clock cycle in which either of the interrupt request inputs **p_extint_b** or **p_critint_b** are asserted, **p_avec_b** and **p_voffset[0:15]** are required to be in a valid state for the highest priority non-masked interrupt being requested.

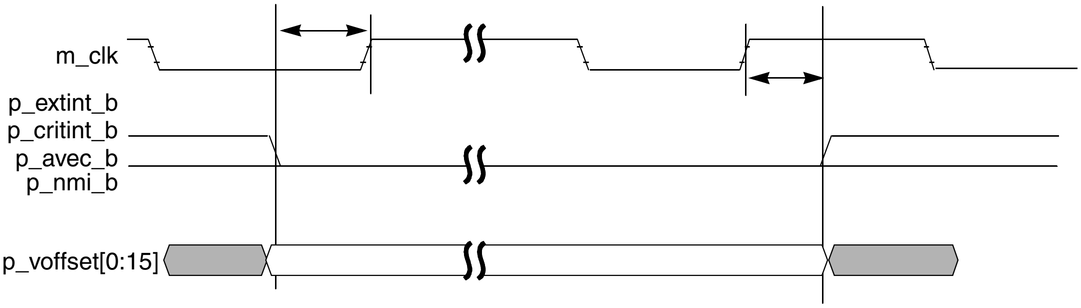


Figure 11-64. Interrupt Interface Input Signals

Figure 11-65 shows the relationship of the interrupt pending signal to the interrupt request inputs. Note that **p_ipend** is asserted combinationally from the **p_extint_b**, **p_critint_b**, and **p_nmi_b** inputs, and the MCSR[NMI] syndrome bit.

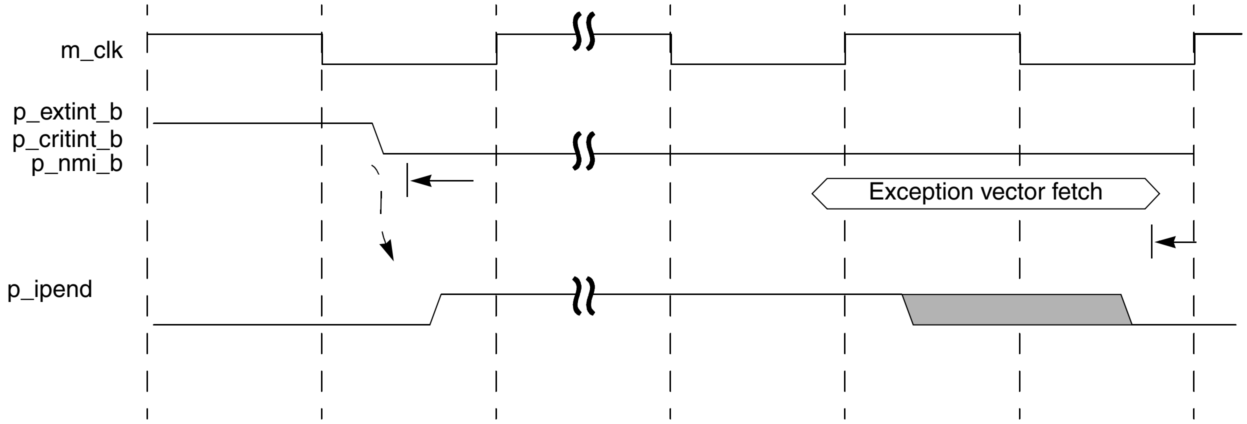


Figure 11-65. Interrupt Pending Operation

Figure 11-66 shows the relationship of the interrupt acknowledge signal to the interrupt request inputs and exception vector fetching.

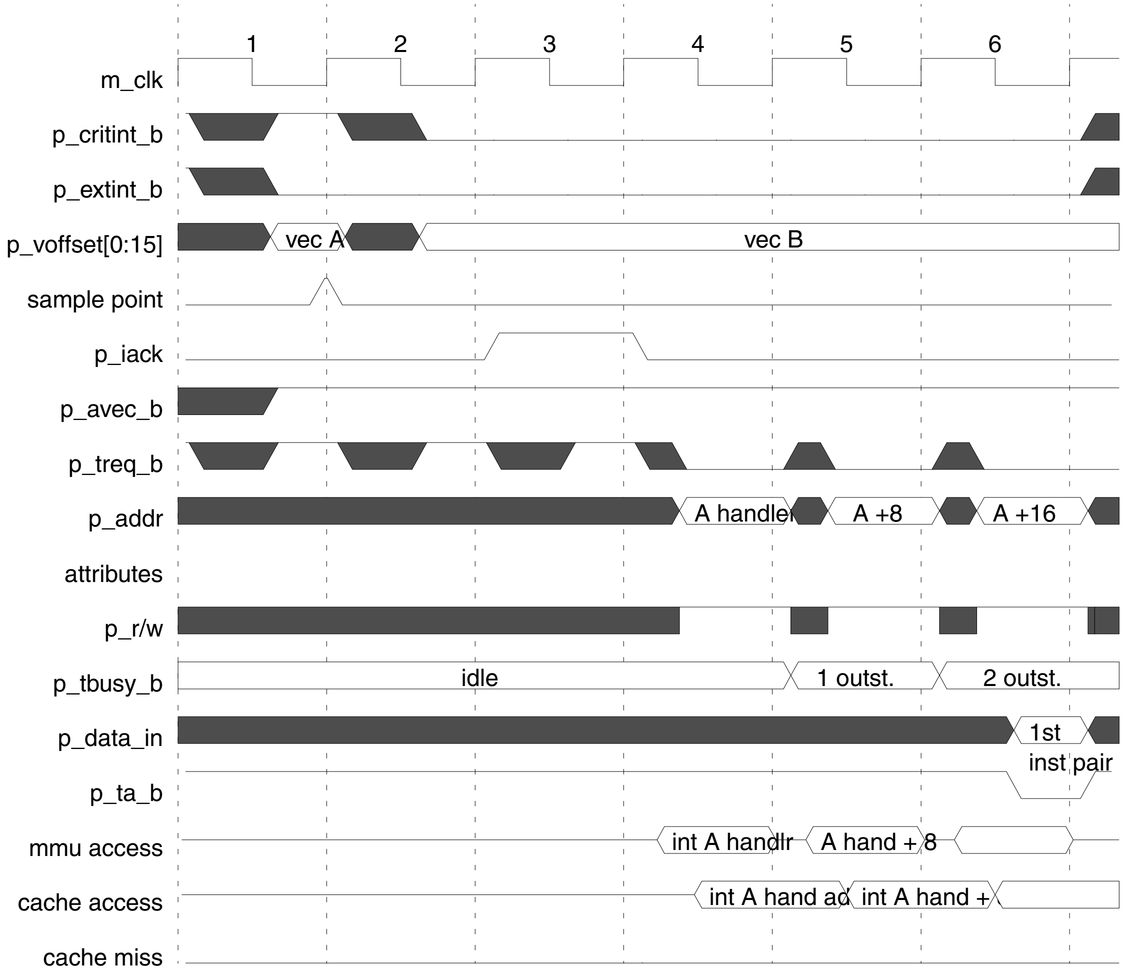


Figure 11-66. Interrupt Acknowledge Operation—1

In this example, an external input interrupt is requested in cycle 1. The **p_voffset[0:15]** inputs are driven with the vector offset for ‘A’, and **p_avec_b** is negated, indicating vectoring is desired. The bus is idle at the time of assertion. The CPU may sample a requested interrupt as early as the cycle it is initially requested, and does so in this example. The interrupt request and the vector offset and auto-vector input are sampled at the end of cycle 1.

In cycle 3, the interrupt is acknowledged by the assertion of the **p_iack** output, indicating that the values present on interrupt inputs at the beginning of cycle 2 have been internally latched and committed to for servicing. Note that the interrupt vector lines have changed to a value of ‘B’ during cycle 2, and the **p_critint_b** input has been asserted by the interrupt controller. The vector number/auto-vector signals must be consistent with the higher priority critical input request, thus must change at the same time the state of the interrupt request inputs change. Because the **p_iack** output asserts in cycle 3, it indicates that the values present at the rise of cycle 2 (vector ‘A’) have been committed to.

During cycle 4, the CPU begins instruction fetching of the handler for vector ‘A’. The new request for a subsequent critical interrupt ‘B’ was not received in time to be acted upon first. It will be acknowledged after the fetch for the external input interrupt handler has been completed and has entered decode.

Note that the time between assertion of an interrupt request input and the acknowledgment of an interrupt may be multiple cycles, and the interrupt inputs may change during that interval. The CPU asserts the **p_iack** output to indicate which cycle an interrupt is committed to.

Figure 11-67 shows an example in which the CPU was unable to acknowledge the external input interrupt during cycle 2 due to internal or external execution conditions, and the critical input request was therefore sampled.

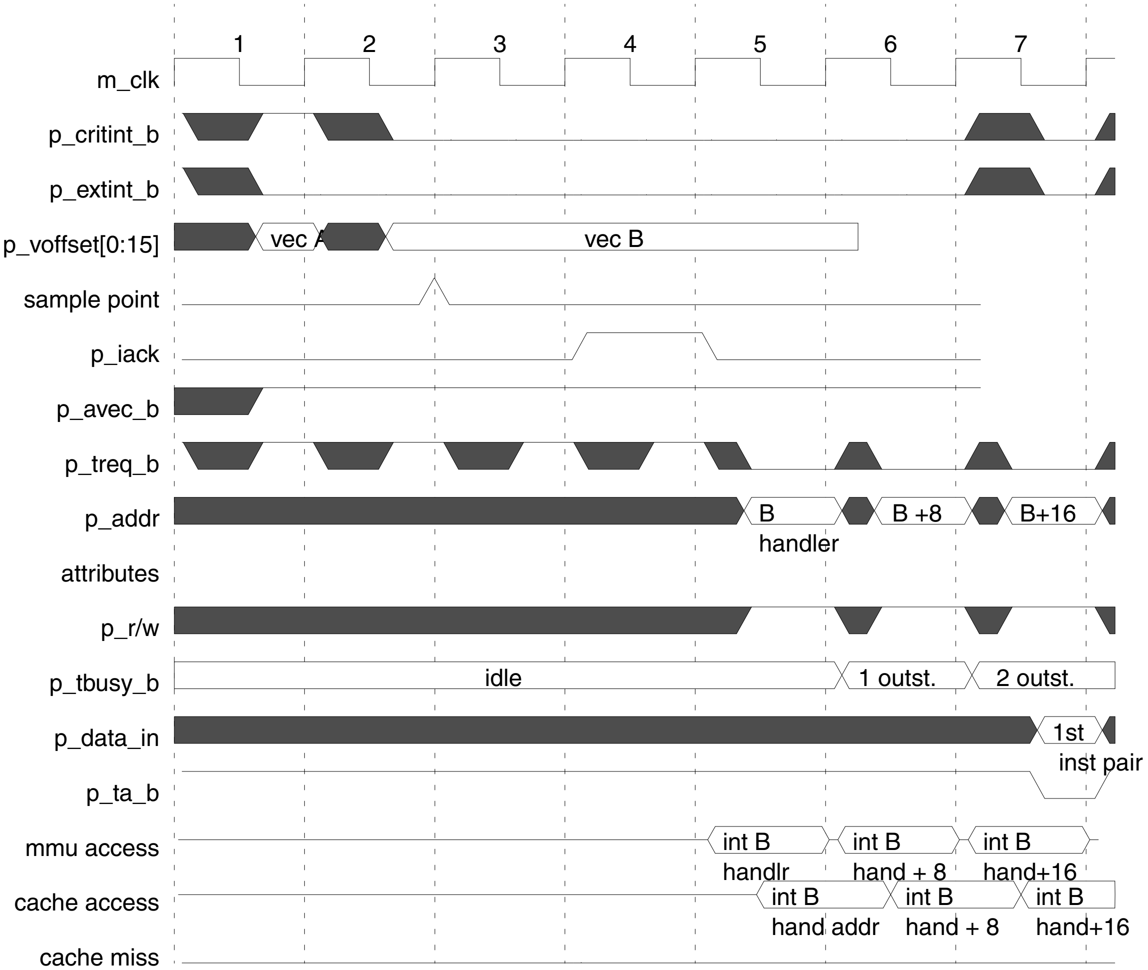


Figure 11-67. Interrupt Acknowledge Operation—2

11.3.9 Time Base Interface

Figure 11-68 shows the required relationships of the time base inputs.

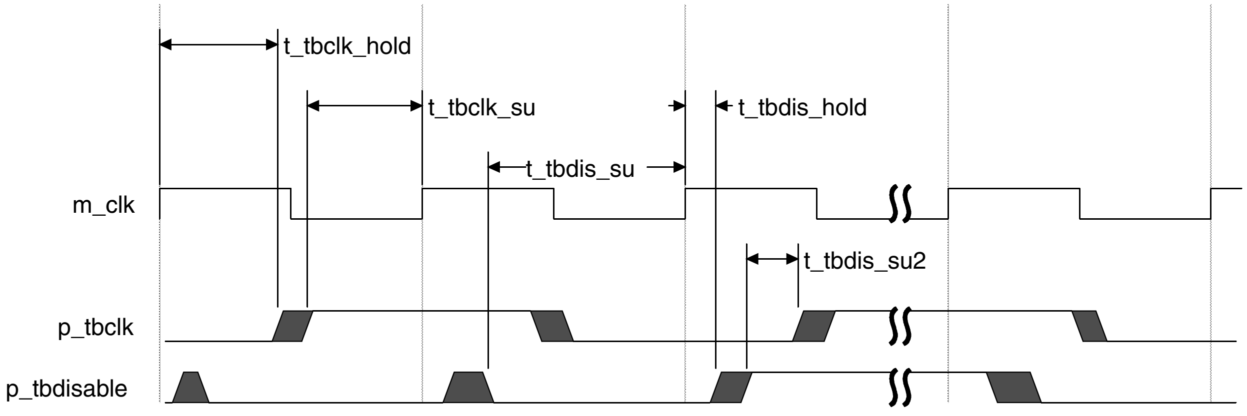


Figure 11-68. Time Base Input Timing

11.3.10 JTAG Test Interface

Figure 11-69, Figure 11-70, and Figure 11-71 show the relationships of the various JTAG related signals to the **j_tclk** input.

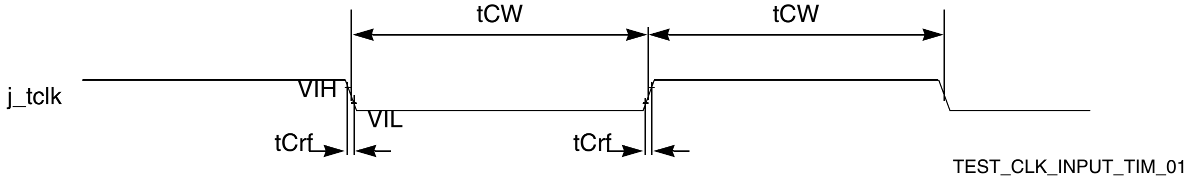


Figure 11-69. Test Clock Input Timing

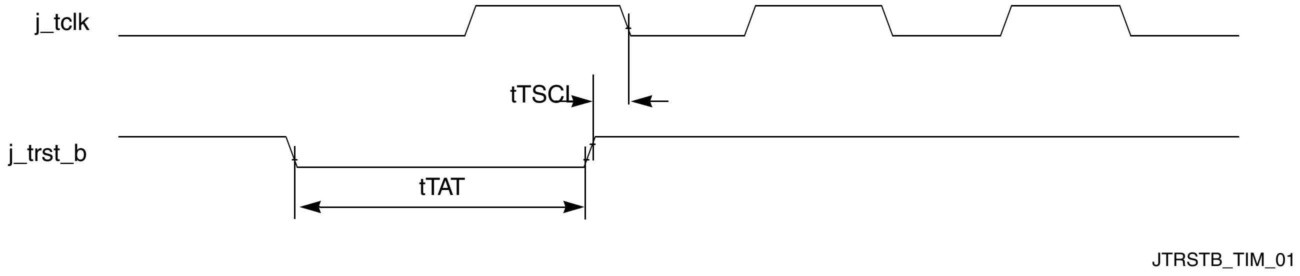


Figure 11-70. j_trst_b Timing

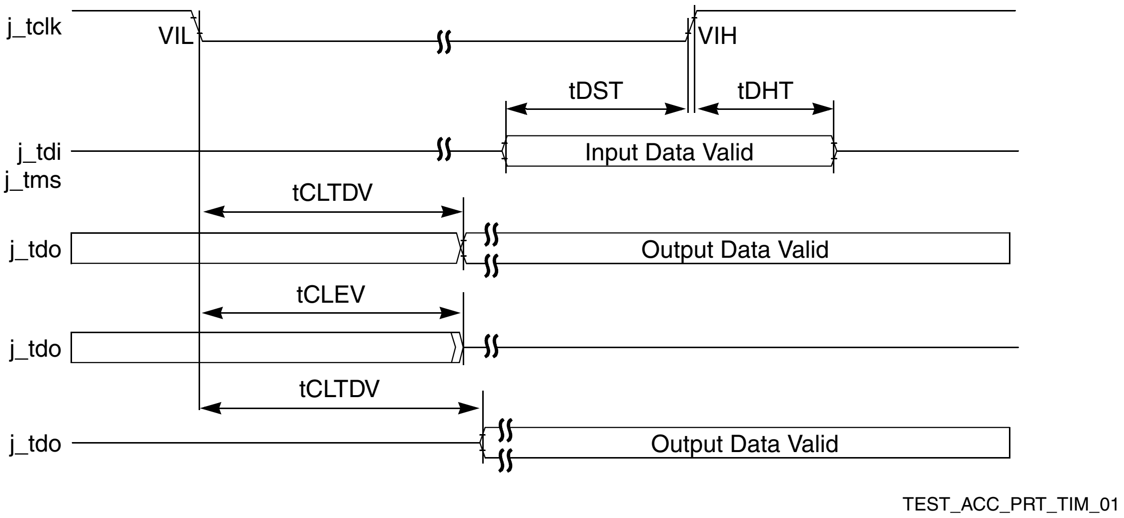


Figure 11-71. Test Access Port Timing

Chapter 12
Power Management

12.1 Power Management

The e200 cores support power management to minimize overall system power consumption. The e200z7 core provides the ability to initiate power management from external sources as well as through software techniques. The power states on the e200 core are described below.

NOTE

> Be aware that some core power-management modes and SoC power-management modes have the same names, but are not the same mode. There is a difference between core and SoC power-management modes.

12.1.1 Active State

The active state is the default state for the e200 core in which all of its internal units operate at full processor clock speed. In this state, the e200 core still provides dynamic power management in which individual internal functional units may stop clocking automatically whenever they are idle.

12.1.2 Waiting State

The e200 core enters the waiting state as a result of executing a **wait** instruction. Following entry into the waiting state, instruction execution and bus activity is suspended. Most internal clocks are gated off in this state. The e200 core asserts *p_waiting* to indicate it is in the waiting state.

Prior to entering the waiting state, all outstanding instructions and bus transactions are completed, and the cache's store and push buffers are flushed. The *m_clk* input should remain running while in the waiting state to allow for interrupt sampling, to allow further transitions into the halted or stopped state if requested, and to keep the time base operational if it is using *m_clk* as the clock source.

In the waiting state, the core is waiting for a valid unmasked pending interrupt request. Once a pending interrupt request is received, the core exits the waiting state and begins interrupt processing. The return program counter value points to the next instruction after the **wait** instruction. The interrupt can be an external input interrupt, various critical interrupts, a debug interrupt (based on ICMP), a nonmaskable interrupt, or a machine check interrupt (*p_mcp_b* assertion, etc.). Once the interrupt processing begins, the core will not return to the waiting state until another **wait** instruction is executed.

The waiting state can be temporarily exited and returned to if a request is made to enter hardware debug mode (various mechanisms), the halted state, or the stopped state. After exiting one of these states, the processor returns to the waiting state. While temporarily exited, the *p_waiting* output negates, and it will be re-asserted once the CPU returns to the waiting state.

12.1.3 Halted State

Instruction execution and bus activity is suspended in the halted state. Most internal clocks are gated off in this state. The e200 core asserts *p_halted* to indicate it is in the halted state. Prior to entering the halted state, all outstanding bus transactions are completed, and the cache's store and push buffers are flushed. The *m_clk* input should remain running while in the halted state to ensure the following:

- Snoop requests continue to be processed.
- Further transitions are allowed into the stopped state if requested.
- The time base stays operational if it is using *m_clk* as the clock source.

12.1.4 Stopped State

In the stopped state, all internal functional units of the e200 core are stopped except the time base unit and the clock control state machine logic. The internal *m_clk* may be kept running to keep the time base active and to allow quick recovery to the full on state. Clocks are not running to functional units in this state except for the time base. The stopped state is reached after transitioning through the halted state with the *p_stop* input asserted. The *p_stopped* output signal will be asserted once the stopped state is reached.The CPU does not enter the stopped state until all snoops have been processed and the snoop queue is empty. System logic is responsible for ensuring that snoop requests are no longer generated once the **p_stop** input is asserted, in order to allow a transition from the halted to the stopped state.

While in the stopped state, further power savings may be achieved by disabling the time base by asserting *p_tbdisable* or by stopping the *m_clk* input. This is done externally by the system after the e200 core is safely in the stopped state and has asserted the *p_stopped* output signal. To exit from the stopped state, the system must first restart the *m_clk* input.

Because the time base unit is off during the stopped state either if it is using *m_clk* as the clock source and *m_clk* is stopped or if the time base clocking is disabled by the assertion of *p_tbdisable*, system software usually needs to access an external time base source after returning to the full on state in order to re-initialize the time base unit. In addition, it is not possible to use a time base related interrupt source to exit low power states.

The e200 also provides the capability of clocking the time base from an independent (but externally synchronized) clock source, which allows the time base to be maintained during the stopped state and generation of a time base related interrupt to indicate an exit condition from the stopped state.

Figure 12-1 shows the power management state diagram.

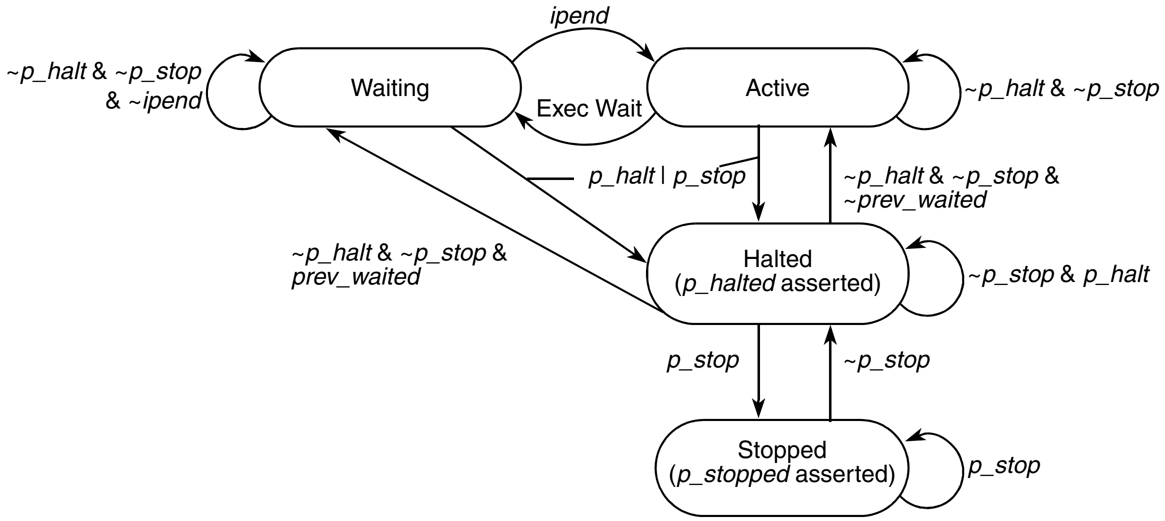


Figure 12-1. Power Management State Diagram

12.1.5 Power Management Pins

The power management pins are as follows:

- *p_waiting*—Output pin asserted when the e200 core is in the waiting state.
- *p_halt*—Input pin is asserted by system logic to request the core to go into the halted state. Negating this pin causes the e200 core to transition back into the active or waiting state if *p_stop* is also negated.
- *p_halted*—Output pin asserted when the e200 core is in the halted state.
- *p_stop*—Input pin is asserted by system logic to request that the e200 core go into the stopped state. Negating this pin causes the e200 core to transition back into the halted state from the stopped state.
- *p_stopped*—Output pin asserted when the e200 core is in the stopped state.
- *p_tbdisable*—Input pin is asserted by system logic when clocking of the time base should be disabled.
- *p_tbint*—Output pin is asserted when an internal time base interrupt request is signaled.
- *p_doze, p_nap*, and *p_sleep*—Output pins that reflects the state of HID0[DOZE], HID0[NAP], and HID0[SLEEP], respectively. These pins are qualified with MSR[WE] = 1. Interpretation of these signals is done by the system logic.
- *p_wakeup*—Output pin asserted when an interrupt is pending or other condition which requires the clock to be running.

12.1.6 Power Management Control Bits

The following bits are used by software to generate a request to enter a power-saving state and to choose the state to be entered:

- MSR[WE]—The WE bit is used to qualify assertion of the *p_doze*, *p_nap*, and *p_sleep* output pins to the system logic. When MSR[WE] is cleared, these pins are negated. When MSR[WE] is set, these pins reflect the state of their respective control bits in the HID0 register.
- HID0[DOZE]—The interpretation of the doze mode bit is done by the external system logic. Doze mode on the e200 core is intended to be the halted state with the clocks running.
- HID0[NAP]—The interpretation of the nap mode bit is done by the external system logic. Nap mode on the e200 core may be used for a power-down state with the time base enabled.
- HID0[SLEEP]—The interpretation of the sleep mode bit is done by the external system logic. Sleep mode on the e200 core may be used for a power-down state with the time base disabled.

12.1.7 Software Considerations for Power Management using Wait Instructions

Executing a **wait** instruction causes the e200 core to complete instruction fetch and execution activity and await an interrupt. The *p_waiting* output is asserted once the waiting state is entered. External system hardware may interpret the state of this signal and activate the *p_halt* and/or *p_stop* inputs to cause the e200 core to enter a quiescent state in which clocks may be disabled for low power operation. Alternatively, system hardware may utilize some other clock control mechanism while the processor is in the waiting state, and *p_wakeup* remains negated.

12.1.8 Software Considerations for Power Management using Doze, Nap, or Sleep

Setting MSR[WE] generates a request to enter a power saving state. The power saving state (doze, nap, or sleep) must be previously determined by setting the appropriate HID0 bit. Setting MSR[WE] has no direct effect on instruction execution, but is simply reflected on *p_doze*, *p_nap*, and *p_sleep* depending on the setting of HID0[DOZE], HID0[NAP], and HID0[SLEEP], respectively. Note that the e200 core is not affected by assertion of these pins directly. External system hardware may interpret the state of these signals and activate the *p_halt* and/or *p_stop* inputs to cause the e200 core to enter a quiescent state in which clocks may be disabled for low power operation.

To ensure a clean transition into and out of a power saving mode, the following program sequence is recommended:

```
         sync
         mtmsr (WE)
         isync
loop:    br loop  (optionally use a wait instruction)
```

An interrupt is typically used to exit a power saving state. The *p_wakeup* output is used to indicate to the system logic that an interrupt (or a debug request) has become pending. System logic uses this output to re-enable the clocks and exit a low power state. The interrupt handler is responsible for determining how to exit the low power loop if one is used. Wait instructions will be exited automatically. The vectored interrupt capability provided by the core may be useful in assisting the determination if an external hardware interrupt is used to perform the wake-up.

12.1.9 Debug Considerations for Power Management

When a debug request is presented to the e200 core while it is in either the waiting, halted, or stopped state, the *p_wakeup* signal is asserted. When *m_clk* is provided to the CPU, it temporarily exist the waiting, halted, or stopped state and enters debug mode regardless of the assertion of *p_halt* or *p_stop*. The *p_waiting*, *p_halted*, and *p_stopped* outputs are negated for the duration of the time the CPU remains in a debug session (*jd_debug_b* asserted). When the debug session is exited, the CPU re-samples the *p_halt* and *p_stop* inputs and re-enters the halted or stopped state as appropriate. If the CPU was previously waiting, and no interrupt was received while in the debug session, it re-enters the waiting state and re-asserts *p_waiting*.

Chapter 13
Debug Support

This chapter describes the debug features of the e200z7 core.

13.1 Overview

Internal debug support in the e200z7 core allows for software and hardware debug by providing debug functions, such as instruction and data breakpoints and program trace modes. For software based debugging, debug facilities consisting of a set of software accessible debug registers and interrupt mechanisms are provided. These facilities are also available to a hardware based debugger which communicates using a modified IEEE 1149.1 test access port (TAP) controller and pin interface. When hardware debug is enabled, the debug facilities controlled by hardware are protected from software modification.

Software debug facilities are defined as part of the Power ISA embedded category. The e200z7 supports a subset of these defined facilities. In addition to the facilities defined in the Power ISA embedded category, the e200z7 provides additional flexibility and functionality in the form of debug event counters, linked instruction and data breakpoints, and sequential debug event detection. These features are also available to a hardware-based debugger.

The e200z7 core also provides support for run-time integrity checking via a parallel signature unit, which is capable of monitoring the internal CPU data read and data write buses and accumulating a pair of 32-bit MISR signatures of the data values transferred over these buses.

13.1.1 Software Debug Facilities

The e200z7 provides debug facilities to enable hardware and software debug functions, such as instruction and data breakpoints and program single stepping. The debug facilities consist of the following:

- Set of debug control registers (DBCR0–6, DBERC0)
- Set of address compare registers (IAC1–8, DAC1, and DAC2)
- Set of data value compare registers (DVC1, DVC2)
- Configurable debug counter
- Debug status register (DBSR) for enabling and recording various kinds of debug events
- Special debug interrupt type built into the interrupt mechanism (see Section 7.6.16, “Debug Interrupt (IVOR15)”)

The debug facilities also provide a mechanism for software-controlled processor reset and for controlling the operation of the timers in a debug environment.

Software debug facilities are enabled by setting the internal debug mode bit in debug control register 0 (DBCR0[IDM]). When internal debug mode is enabled, debug events can occur and be enabled to record exceptions in the debug status register (DBSR). If enabled by MSR[DE], these recorded exceptions cause debug interrupts to occur. When DBCR0[IDM] and DBCR0[EDM] are cleared, no debug events occur, and no status flags are set in DBSR unless already set. When DBCR0[IDM] is cleared or is overridden by DBCR0[EDM] being set and DBERC0 indicating no resource is owned by software, no debug interrupts occur, regardless of the contents of DBSR.

A software debug interrupt handler may access all system resources and perform necessary functions appropriate for system debug.

13.1.1.1 Power ISA Embedded Category Compatibility

The e200z7 core implements a subset of the Power ISA embedded category internal debug features. The following restrictions on functionality are present:

- Instruction address compares do not support compare on physical (real) addresses.
- Data address compares do not support compare on physical (real) addresses.

13.1.2 Additional Debug Facilities

In addition to the debug functionality defined in Power ISA embedded category, the e200z7 provides capability to link instruction and data breakpoints, provides a configurable debug event counter to allow debug exception generation capability, and also provides a sequential breakpoint control mechanism.

The e200z7 also defines two new debug events (CIRPT, CRET) for debugging around critical interrupts.

In addition, the e200z7 implements the debug unit, which when enabled allows debug interrupts to utilize a dedicated set of save/restore registers (DSRR0, DSRR1) for saving state information when a debug interrupt occurs, and for restoring this state information at the end of a debug interrupt handler by means of the **rfdi** or **se_rfdi** instructions.

The e200 also provides the capability of sharing resources between hardware and software debuggers. See Section 13.1.4, “Software/Hardware Debug Resource Sharing.”

13.1.3 Hardware Debug Facilities

The e200z7 core contains facilities that allow for external test and debugging. A modified IEEE 1149.1 control interface is used to communicate with the core resources. This interface is implemented through a standard 1149.1 TAP (test access port) controller.

By using public instructions, the external debugger can freeze or halt the e200z7 core, read, and write internal state and debug facilities, single-step instructions, and resume normal execution.

Hardware debug is enabled by setting the external debug mode enable bit in debug control register 0 (DBCR0[EDM]), which is also aliased to EDBCR0[EDM]. Setting DBCR0[EDM] overrides the internal debug mode enable bit DBCR0[IDM] unless resources are provided back to software via the settings in DBERC0. When the hardware debug facility is enabled, software is blocked from modifying the hardware-owned debug facilities. In addition, since the hardware debugger owns the resources, inconsistent values may be present if software attempts to read hardware-owned debug-related resources.

When hardware debug is enabled by setting EDBCR0[EDM] = 1, the control registers and resources described in Section 13.3, “Debug Registers,” are reserved for use by the external debugger. The same events described in Section 13.2, “Software Debug Events and Exceptions,” are also used for external debugging, but exceptions are not generated to running software. Hardware-owned debug events enabled in the respective DBCR0–6 registers are recorded in EDBSR0—not the DBSR—regardless of MSR[DE]. No debug interrupts are generated unless DBERC0 settings grant the resource back to software, and the corresponding event bit in EDBSRMSK0 does not mask debug mode entry. Instead, the CPU enters debug mode when an enabled event causes an EDBSR0 bit to become set. DBCR0[EDM], EDBSR0, EDBSRMSK0, and DBERC0 may only be written through the OnCE port.

A program trace PC FIFO provides to support program change of flow capture.

Access to most debug resources (registers) requires that the CPU clock (**m_clk**) be running in order to perform write accesses from the external hardware debugger.

13.1.4 Software/Hardware Debug Resource Sharing

A a hardware debugger and software debug may share debug resources based on the debug control register DBERC0’s settings. When DBCR0[EDM] is set, DBERC0 settings determine which debug resources are allocated to software and which resources remain under exclusive hardware control. Software-owned resources that set DBSR bits when DBCR0[IDM] = 1 cause a debug interrupt to occur when enabled with MSR[DE]. Hardware-owned resources that set EDBSR0 bits when EDBCR0[EDM] = 1 cause an entry into debug mode if EDBSRMSK0 does not mask the event. DBERC0 is read-only by software.

When resource sharing is enabled (DBCR0[EDM] = 1 and DBERC0[IDM] = 1), software may only modify software-owned resources. Hardware always has full access to all registers and all register fields through the OnCE register access mechanism. It is up to the debug firmware to properly implement modifications to these registers, using read-modify-write operations to implement any control sharing with software. Hardware-owned resources set status bits in EDBSR0 instead of in DBSR. Settings in DBERC0 should be considered by the debug firmware in order to preserve software settings of control and status registers as appropriate when hardware modifications to the debug registers are performed.

13.1.4.1 Simultaneous Hardware and Software Debug Event Handing

Because it is possible for a hardware-owned resource to produce a debug event at the same time that a software-owned resource produces a different debug event, a priority ordering mechanism guarantees that the hardware event is handled as soon as possible while preserving the software event. The CPU gives highest priority to the software event initially in order to reach a recoverable boundary, but then gives highest priority to the hardware event so that it enters debug mode as near the point of event occurrence as possible.

This is implemented by allowing software exception handling to begin internally to the CPU. It continues until it reaches the point where the current program counter and MSR values have been saved into DSRR0/1, and the new PC points to the debug interrupt handler along with the new MSR updates. At this point, hardware priority takes over, and the CPU enters debug mode.

Figure 13-1 shows the e200z7 debug resources.

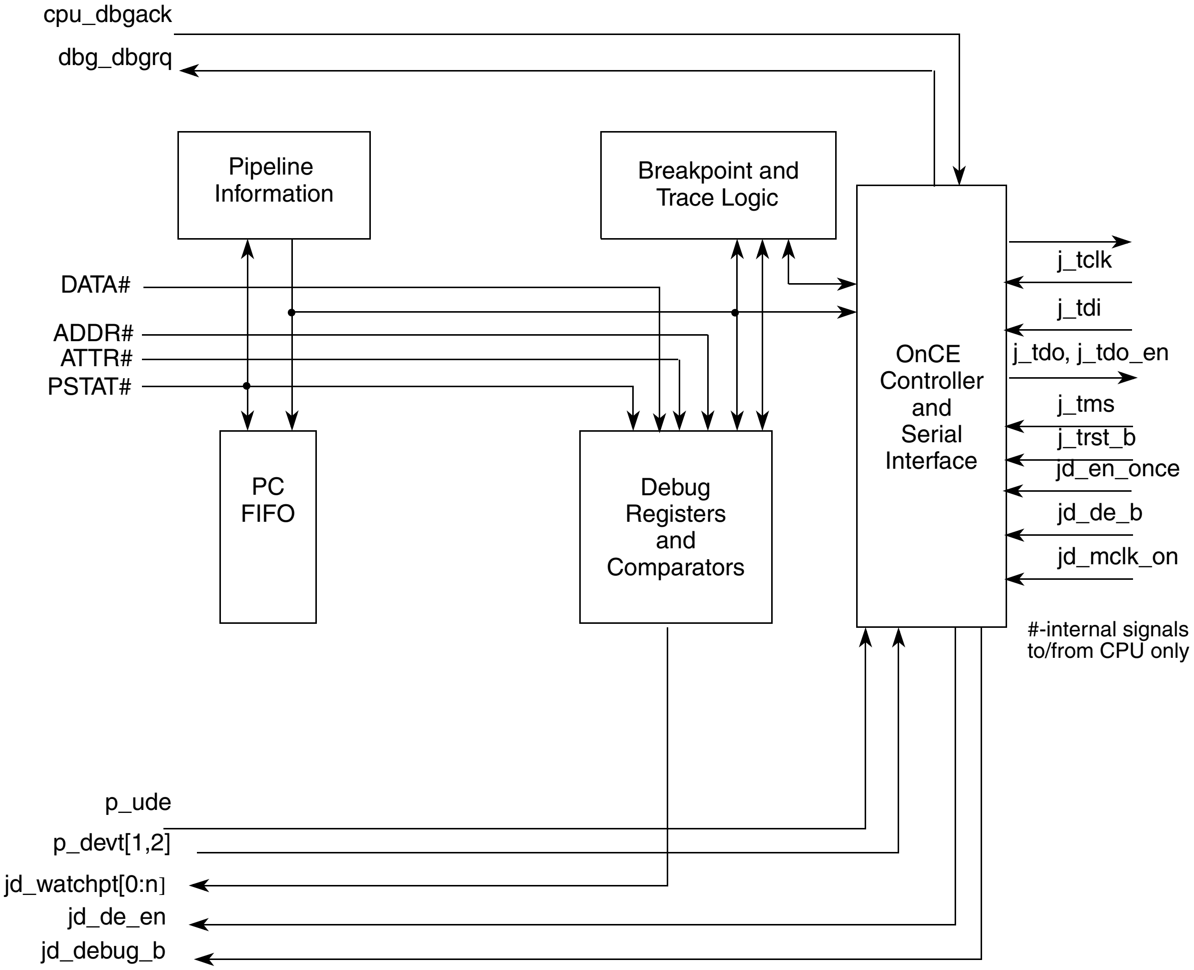


Figure 13-1. e200z7 Debug Resources

13.2 Software Debug Events and Exceptions

Software debug events and exceptions are available when internal debug mode is enabled (DBCR0[IDM] = 1) and not overridden by external debug mode (DBCR0[EDM] must either be cleared or corresponding resources must be allocated to software debug by the settings in DBERC0). When enabled, debug events cause debug exceptions to be recorded in the debug status register. Specific event types are enabled by the debug control registers (DBCR0–6). The unconditional debug event (UDE) is an exception to this rule; it is always enabled. A debug interrupt is generated once a debug resource that is owned by software sets a debug status register (DBSR) bit (other than MRR and CNT1TRG) if debug interrupts are enabled by MSR[DE]. The debug interrupt handler is responsible for ensuring that multiple repeated debug interrupts do not occur by clearing the DBSR as appropriate.

Certain debug events are not allowed to occur when MSR[DE] = 0 and DBCR0[IDM] = 1. In such situations, no debug exception occurs and thus no DBSR bit is set. Other debug events may cause debug exceptions and set DBSR bits regardless of the state of MSR[DE]. A debug interrupt is delayed until MSR[DE] is later set.

When a debug status register bit is set while MSR[DE] = 0, an imprecise debug event flag (DBSR[IDE]) is also set to indicate that an exception bit in the debug status register was set while debug interrupts were disabled. Debug interrupt handler software can use this bit to determine whether the address recorded in debug save/restore register 0 is an address associated with the instruction causing the debug exception, or the address of the instruction which enabled a delayed debug interrupt by setting MSR[DE]. An **mtmsr** or **mtdbcr0** that causes both MSR[DE] and DBCR0[IDM] to become set, enabling precise debug mode, may cause an imprecise (delayed) debug exception to be generated due to an earlier recorded event in the debug status register.

There are eight types of debug events defined by Power ISA embedded category:

1. Instruction address compare debug events
2. Data address compare debug events
3. Trap debug events
4. Branch taken debug events
5. Instruction complete debug events
6. Interrupt taken debug events
7. Return debug events
8. Unconditional debug events

These events are described in detail in the *EREF.*

In addition, e200z7 defines the following additional debug events:

- The debug counter debug events DCNT1 and DCNT2, which are described in Section 13.2.11, “Debug Counter Debug Event.”
- The external debug events DEVT1 and DEVT2, which are described in Section 13.2.12, “External Debug Event.”
- The critical interrupt taken debug event CIRPT, which is described in Section 13.2.8, “Critical Interrupt Taken Debug Event.”
- The critical return debug event CRET, which is described in Section 13.2.10, “Critical Return Debug Event.”

The e200z7 debug configuration supports most of these event types. Unsupported Power ISA embedded category functionality is as follows:

- Instruction address compare and data address compare *Real address* mode are not supported.

A brief description of each of the event types follows.

NOTE

In these descriptions, DSRR0 and DSRR1 are used, assuming that the debug unit is enabled. If it is disabled, use CSRR0 and CSRR1 respectively.

13.2.1 Instruction Address Compare Event

Instruction address compare debug events occur when enabled and execution is attempted of an instruction at an address that meets the criteria specified in the DBCR0, DBCR1, DBCR5, DBCR6, and IAC1–8 registers. Instruction address compares may specify user/supervisor mode and instruction space (MSR[IS]), along with an effective address, masked effective address, or range of effective addresses for comparison (range compares are not supported for IAC5–8). This event can occur and be recorded in DBSR regardless of the setting of MSR[DE]. IAC events do not occur when an instruction would not have normally begun execution due to a higher priority exception at an instruction boundary.

IAC compares perform a 31-bit compare for VLE instruction pages, and 30-bit compares for Power ISA instruction pages. Each half word fetched by the instruction fetch unit are marked with a set of bits indicating whether an instruction address compare occurred on that half word. Debug exceptions occur if enabled and either a 16-bit instruction or the first half word of a 32-bit instruction is tagged with an IAC hit. For instruction fetches that miss in the TLB, Power ISA pages are assumed, and a 30-bit compare is performed.

13.2.2 Data Address Compare Event

Data address compare debug events occur when enabled and execution of a load or store class instruction or a cache maintenance instruction results in a data access that meets the criteria specified in the DBCR0, DBCR2, DBCR4, DAC1, DAC2, DVC1, and DVC2 registers. Data address compares may specify user/supervisor mode and data space (MSR[DS]), along with an effective address, masked effective address, or range of effective addresses for comparison. This event can occur and be recorded in DBSR regardless of the setting of MSR[DE]. Two address compare values (DAC1, DAC2) are provided.

NOTE

In contrast to the Power ISA embedded category definition, data address compare events on the e200z7 do not prevent the load or store class instruction from completing. If a load or store class instruction completes successfully without a data TLB or data storage interrupt, data address compare exceptions are reported at the completion of the instruction. If the exception results in a precise debug interrupt, the address value saved in DSRR0 (or CSRR0 if the debug unit is disabled) is the address of the instruction following the load or store class instruction. For DVC DAC events, the exception can be imprecisely reported even further past the load or store class instruction generating the event (without necessarily affecting DBSR[IDE]) and the saved address value can point to a subsequent instruction past the next instruction. This occurrence is indicated in the DBSR[DAC_OFST] field.

If a load or store class instruction does not complete successfully due to a data TLB or data storage exception or a machine check condition for the load or store, and a data address compare debug exception also occurs, or a Debug counter event based on a counted DAC occurs, the result is an imprecise debug interrupt, the address value saved in DSRR0 (or CSRR0 if the debug unit is disabled) is the address of the load or store class instruction, and the DBSR[IDE] is set. In addition to occurring when DBCR0[IDM] = 1, this circumstance can also occur when DBCR0[EDM] = 1.

- DAC events are not recorded or counted if a load multiple word or store multiple word instruction is interrupted prior to completion by a critical input or external input interrupt.
- DAC events are not signaled on the second portion of a misaligned load or store that is broken up into two separate accesses.
- DAC events are not signaled on the **tlbre**, **tlbwe**, **tlbsx**, or **tlbivax** instructions.
- DAC[1,2] events are not signaled if DVC[1,2]M is non-zero and a DSI or DTLB exception occurs on the load or store, since the load or store access is not performed. For a **lmw** or **stmw** transfer however, if a DVC successfully occurs on a transfer and a later transfer encounters a DSI or DTLB exception, the DAC event will be reported, since a successful data value compare took place.

13.2.2.1 Data Address Compare Event Status Updates

Data address compare debug events with data value compares can be reported ambiguously in several circumstances involving issuing a sequence of load or store class instructions. Due to the CPU pipeline and the delay in performing the data value compare following completion of the access, if the first load or store class instruction generates a DVC DAC, a second and possibly third load or store class instruction may also generate a DAC or DVC DAC event, or may generate a DTLB or DSI exception with or without a simultaneous DAC event.

Also, since non-load/store instructions may be dual-issued in combination with a load/store instruction, the actual number of additional instructions which are completed following a recognized DVC DAC on a load/store instruction may vary from 0 to 5. This value will be reported in the DBSR[DAC_OFST] field when the DVC DAC status is recorded.

Table 13-1 outlines the settings of the DBSR, DSRR0 saved value, and potential updating of the ESR and MMU MASx registers for various exception cases on sequences of load/store class instructions. Not all exception combinations are covered in the table, such as IAC, ITLB, ISI, or alignment exceptions on subsequent instructions. In general these exceptions cause further instruction issue to be halted, execution of the excepting instruction to be aborted, and reporting of these exceptions to be masked. The saved DSRR0 value points to this excepting instruction, and the exception(s) may be regenerated after returning from the debug interrupt handler and attempting to re-execute the instruction pointed to by DSRR0. In addition, in the examples in Table 13-1, the DAC_OFST and DSRR0 values assume no dual issue occurs.

If dual-issue occurs with the first, second, or third column, the DAC_OFST and DSRR0 values point beyond the values shown.

Table 13-1. DAC Events and Resultant Updates

1st load/store class instruction	2nd instruction (load/store class unless otherwise specified)	3rd instruction (load/store class unless otherwise specified)	Result
DTLB Error, no DAC	—	—	Take DTLB exception, no DBSR update, update MASx registers for 1st load/store class instruction. Update ESR.
DSI, no DAC	—	—	Take DSI exception, no DBSR update, no MASx register update. Update ESR.
DTLB Error, with DACx	—	—	Take Debug exception, DBSR update setting DACx and IDE, DAC_OFST not set. No MASx register update for 1st load/store class instruction. DSRR0 points to 1st load/store class instruction. No ESR update.
DSI, with DACx	—	—	Take Debug exception, DBSR update setting DACx and IDE, DAC_OFST not set. DSRR0 points to 1st load/store class instruction. No MASx register update. No ESR update.
DACx	—	—	Take Debug exception, DBSR update setting DACx, DAC_OFST not set. DSRR0 points to 2nd load/store class instruction. No MASx register update. No ESR update.
DVC DACx	No exceptions, any instruction	No exceptions, Non-ldst instruction	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 0b001. DSRR0 points to 3rd instruction. No MASx register update. No ESR update.
DVC DACx	No exceptions	No exceptions, Ldst instruction	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 0b010. DSRR0 points to instruction after 3rd instruction. No MASx register update. No ESR update.
DVC DACx	DTLB Error, no DAC	—	Take Debug exception, DBSR update setting DACx, DAC_OFST not set. DSRR0 points to 2nd load/store class instruction. no MASx register update. No ESR update. No debug counter updates for 2nd ld/st instruction. **Note:** In this case the 2nd ld/st exception is masked. This behavior is implementation dependent and may differ on other CPUs.
DVC DACx	DSI, no DAC	—	Take Debug exception, DBSR update setting DACx, DAC_OFST not set. DSRR0 points to 2nd load/store class instruction. No MASx register update. No ESR update. No debug counter updates for 2nd ld/st instruction. **Note:** In this case the 2nd ld/st exception is masked. This behavior is implementation dependent and may differ on other CPUs.
DVC DACx	DTLB Error, with DACy	—	Take Debug exception, DBSR update setting DACx. DAC_OFST not set. DSRR0 points to 2nd load/store class instruction. No MASx register update. No ESR update. No debug counter update occurs for the 2nd ld/st. **Note:** In this case the 2nd ld/st exception is masked. This behavior is implementation dependent and may differ on other CPUs.
DVC DACx	DSI, with DACy	—	Take Debug exception, DBSR update setting DACx. DAC_OFST not set. DSRR0 points to 2nd load/store class instruction. No MASx register update. No ESR update. No debug counter update occurs for the 2nd ld/st. **Note:** In this case the 2nd ld/st exception is masked. This behavior is implementation dependent and may differ on other CPUs.

Table 13-1. DAC Events and Resultant Updates (continued)

1st load/store class instruction	2nd instruction (load/store class unless otherwise specified)	3rd instruction (load/store class unless otherwise specified)	Result
DVC DACx	DACy	—	Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 0b001. DSRR0 points to 3rd instruction. Debug counter update occurs for the 2nd ld/st as appropriate. **Note:** In this case debug counter updates can occur for the 2nd ld/st even though the 1st ld/st has a DVC DAC exception[1]. **Note:** In this case if x = y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the “no DACy” case.
DVC DACx	DVC DACy, Normal Ldst	Non-Ldst instruction	Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 0b001. DSRR0 points to the 3rd instruction. Debug counter update occurs for the 2nd ld/st as appropriate. **Note:** In this case debug counter updates occur for the 2nd ld/st even though the 1st ld/st has a DVC DAC exception[1]. **Note:** In this case if x = y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the “no DACy” case.
DVC DACx	DVC DACy, Normal Ldst	Ldst instruction, no exception	Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 0b010. DSRR0 points to instruction after the 3rd load/store class instruction. Debug counter update occurs for the 2nd and 3rd ld/st as appropriate. **Note:** In this case debug counter updates occur for the 2nd and 3rd ld/st even though the 1st ld/st has a DVC DAC exception[2]. **Note:** In this case if x = y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the “no DACy” case.
DVC DACx	DVC DACy, Normal Ldst	DSI Error, with or without DAC	Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 0b001. No ESR update. DSRR0 points to 3rd instruction. Debug counter update occurs for the 2nd ld/st as appropriate. **Note:** In this case debug counter updates occur for the 2nd ld/st even though the 1st ld/st has a DVC DAC exception[1]. **Note:** In this case if x = y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the “no DACy” case. **Note:** In this case the 3rd ld/st exception is masked. This behavior is implementation dependent and may differ on other CPUs.
DVC DACx	DVC DACy, Normal Ldst	DTLB, with or without DAC	Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 0b001. No ESR update. No MASx register updates. DSRR0 points to 3rd instruction. Debug counter update occurs for the 2nd ld/st as appropriate. **Note:** In this case debug counter updates occur for the 2nd ld/st even though the 1st ld/st has a DVC DAC exception[1]. **Note:** In this case if x = y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the “no DACy” case. **Note:** In this case the 3rd ld/st exception is masked. This behavior is implementation dependent and may differ on other CPUs.

Table 13-1. DAC Events and Resultant Updates (continued)

1st load/store class instruction	2nd instruction (load/store class unless otherwise specified)	3rd instruction (load/store class unless otherwise specified)	Result
DVC DACx	DVC DACy, Normal Ldst	DACy, or DVC DAC_y Normal Ldst or multiple word Ldst	Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 3’b010. DSRR0 points to instruction after the 3rd load/store class instruction. Debug counter update occurs for the 2nd and 3rd ld/st as appropriate. **Note:** In this case debug counter updates occur for the 2nd and 3rd ld/st even though the 1st ld/st has a DVC DAC exception[2]. **Note:** In this case if x = y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the “no DACy” case.
DVC DACx	DVC DACy, Ldst multiple (lmw, stmw)	Any instruction including ld/st	Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 3’b001. DSRR0 points to the 3rd instruction. Debug counter update occurs for the 2nd ld/st multiple as appropriate. **Note:** In this case debug counter updates occur for the 2nd ld/st multiple even though the 1st ld/st has a DVC DAC exception[1]. **Note:** In this case if x = y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the “no DACy” case.
DVC DACx	Any instruction (no exception)	DSI, with or without DAC, Normal Ldst or multiple word Ldst	Take Debug exception, DBSR update setting DACx. DAC_OFST set to 3’b001. DSRR0 points to the 3rd instruction. No MASx register update. No ESR update. No debug counter update occurs for the 3rd instruction. Debug counter update occurs for the 2nd instruction as appropriate. **Note:** In this case debug counter updates occur for the 2nd instruction even though the 1st ld/st has a DVC DAC exception[1]. **Note:** In this case the 3rd ld/st exception is masked. This behavior is implementation dependent and may differ on other CPUs.
DVC DACx	Any instruction (no exception)	DACy, or DVC DAC_y Normal Ldst or multiple word Ldst	Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 3’b010. DSRR0 points to instruction after the 3rd class instruction. Debug counter update occurs for the 2nd and 3rd instruction as appropriate. **Note:** In this case debug counter updates occur for the 2nd and 3rd instructions even though the 1st ld/st has a DVC DAC exception[2]. **Note:** In this case if x = y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the “no DACy” case.

[1] The 2nd instruction may cause DAC, ICMP or IAC events to be counted.

[2] The 2nd and 3rd instructions may cause DAC, ICMP or IAC events to be counted.

Table 13-2–Table 13-5 show some example updates for specific code sequences of dual issuing of load/store class instructions with non-load/store class instructions and the results of DAC and DVC events on selected ones of the load/store instructions.

Table 13-2 shows the first example case.

Table 13-2. DAC Events and Resultant Updates, Dual-Issue Case 1

Instruction Sequence: The following pairs dual issue: • (1) load/store (2) alu • (3) load/store (4) alu • (5) load/store (6) alu	**Event(s)**	**Result**
	Instruction (1): DTLB Error, no DAC	Take DTLB exception, no DBSR update, update MASx registers for 1st load/store instruction. Update ESR.
	Instruction (1): DSI, no DAC	Take DSI exception, no DBSR update, no MASx register update. Update ESR.
	Instruction (1): DTLB Error, with DACx	Take Debug exception, DBSR update setting DACx and IDE, DAC_OFST set to 0b000. DSRR0 points to instruction 1. No MASx register update. No ESR update.
	Instruction (1): DSI, with DACx	
	Instruction (1): DACx	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 0b000. DSRR0 points to instruction 2. No MASx register update. No ESR update.
	Instruction (1): DVC DACx No other exceptions	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 0b100. DSRR0 points to instruction 6. No MASx register update. No ESR update. Debug counter update occurs for instructions 1–5 as appropriate. No debug counter or event updates for instruction 6.
	Instruction (1): DVC DACx Instruction (3): DTLB Error, with or without DAC	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 0b001. DSRR0 points to instruction (3). no MASx register update. No ESR update. Debug counter update occurs for instructions 1–2 as appropriate. No debug counter or event updates for instructions 3–6. **Note:** In this case the 2nd ld/st exception is masked. This behavior is implementation dependent and may differ on other CPUs.
	Instruction (1): DVC DACx Instruction (3): DSI, with or without DAC	
	Instruction (1): DVC DACx Instruction (3): DACy	Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 0b010. DSRR0 points to instruction 4. Debug counter update occurs for instructions 1–3 as appropriate. No debug counter or event updates for instructions 4–6. **Note:** In this case if x = y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the "no DACy" case.

Table 13-2. DAC Events and Resultant Updates, Dual-Issue Case 1 (continued)

Instruction Sequence: The following pairs dual issue: • (1) load/store (2) alu • (3) load/store (4) alu • (5) load/store (6) alu	**Event(s)**	**Result**
	Instruction (1): DVC DACx Instruction (3): DVC DACy	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 3’b100. DSRR0 points to instruction 6. No MASx register update. No ESR update. Debug counter update occurs for instructions 1–5 as appropriate. No debug counter or event updates for instruction 6. **Note:** In this case debug counter updates can occur for instructions 2–5 even though the 1st ld/st has a DVC DAC exception. **Note:** Note: in this case if x = y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the “no DACy” case.
	Instruction (1): DVC DACx Instruction (3): DVC DACy Instruction (5): DSI, with or without DAC	Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 0b010. No ESR update. DSRR0 points to instruction 4. Debug counter update occurs for instructions 1–3 as appropriate. No debug counter or event updates for instructions 4–6. **Note:** In this case if x = y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the “no DACy” case. **Note:** In this case the 3rd ld/st exception is masked. This behavior is implementation dependent and may differ on other CPUs.
	Instruction (1): DVC DACx Instruction (3): DVC DACy Instruction (5): DTLB Error, with or without DAC	
	Instruction (1): DVC DACx Instruction (3): DVC DACy Instruction (5): DACy or DVC DACy	Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 0b100. No ESR update. DSRR0 points to instruction 6. Debug counter update occurs for instructions 1–5 as appropriate. **Note:** In this case if x = y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the “no DACy” case.

Table 13-3 shows the second example case.

Table 13-3. DAC Events and Resultant Updates, Dual-Issue Case 2

Instruction Sequence: The following pairs dual issue: • (1) load/store (2) alu • (3) load/store (4) alu • (5) alu (6) load/store	**Event(s)**	**Result**
	Instruction (1): DTLB Error, no DAC	Take DTLB exception, no DBSR update, update MASx registers for 1st load/store instruction. Update ESR.
	Instruction (1): DSI, no DAC	Take DSI exception, no DBSR update, no MASx register update. Update ESR.
	Instruction (1): DTLB Error, with DACx	Take Debug exception, DBSR update setting DACx and IDE, DAC_OFST set to 0b000. DSRR0 points to instruction 1. No MASx register update. No ESR update.
	Instruction (1): DSI, with DACx	
	Instruction (1): DACx	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 0b000. DSRR0 points to instruction 2. No MASx register update. No ESR update.
	Instruction (1): DVC DACx No other exceptions	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 0b101. DSRR0 points to instruction after instruction 6. No MASx register update. No ESR update.Debug counter update occurs for instructions 1–6 as appropriate.
	Instruction (1): DVC DACx Instruction (3): DTLB Error, with or without DAC	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 0b001. DSRR0 points to instruction 3. No MASx register update. No ESR update. Debug counter update occurs for instructions 1–2 as appropriate. No debug counter or event updates for instructions 3–6. **Note:** In this case the 2nd ld/st exception is masked. This behavior is implementation dependent and may differ on other CPUs.
	Instruction (1): DVC DACx Instruction (3): DSI, with or without DAC	
	Instruction (1): DVC DACx Instruction (3): DACy	Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 0b010. DSRR0 points to instruction 4. Debug counter update occurs for instructions 1–3 as appropriate. No debug counter or event updates for instructions 4–6. **Note:** In this case if x = y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the “no DACy” case.
	Instruction (1): DVC DACx Instruction (3): DVC DACy	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 0b101. DSRR0 points to instruction 7. No MASx register update. No ESR update. Debug counter update occurs for instructions 1–6 as appropriate. **Note:** In this case if x = y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the “no DACy” case.

Table 13-3. DAC Events and Resultant Updates, Dual-Issue Case 2 (continued)

Instruction Sequence: The following pairs dual issue: • (1) load/store (2) alu • (3) load/store (4) alu • (5) alu (6) load/store	**Event(s)**	**Result**
	Instruction (1): DVC DACx Instruction (3): DVC DACy Instruction (6): DSI, with or without DAC	Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 0b010. No ESR update. DSRR0 points to instruction 4. Debug counter update occurs for instructions 1–3 as appropriate. No debug counter or event updates for instruction 4. **Note:** In this case if x = y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the “no DACy” case. **Note:** In this case the 3rd ld/st exception is masked. This behavior is implementation dependent and may differ on other CPUs.
	Instruction (1): DVC DACx Instruction (3): DVC DACy Instruction (6): DTLB Error, with or without DAC	
	Instruction (1): DVC DACx Instruction (3): DVC DACy Instruction (6): DACy or DVC DACy	Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 0b101. No ESR update. DSRR0 points to instruction 7. Debug counter update occurs for instructions 1–6 as appropriate. No debug counter or event updates for instruction 7. **Note:** In this case if x = y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the “no DACy” case.

Table 13-4 shows the third example case.

Table 13-4. DAC Events and Resultant Updates, Dual-Issue Case 3

Instruction Sequence: The following pairs dual issue: • (1) load/store (2) alu • (3) alu (4) alu • (5) load/store (6) alu	**Event(s)**	**Result**
	Instruction (1): DTLB Error, no DAC	Take DTLB exception, no DBSR update, update MASx registers for 1st load/store instruction. Update ESR.
	Instruction (1): DSI, no DAC	Take DSI exception, no DBSR update, no MASx register update. Update ESR.

Table 13-4. DAC Events and Resultant Updates, Dual-Issue Case 3 (continued)

Instruction Sequence: The following pairs dual issue: • (1) load/store (2) alu • (3) alu (4) alu • (5) load/store (6) alu	**Event(s)**	**Result**
	Instruction (1): DTLB Error, with DACx	Take Debug exception, DBSR update setting DACx and IDE, DAC_OFST set to 0b000. DSRR0 points to instruction 1. No MASx register update. No ESR update.
	Instruction (1): DSI, with DACx	
	Instruction (1): DACx	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 0b000. DSRR0 points to instruction 2. No MASx register update. No ESR update.
	Instruction (1): DVC DACx No other exceptions	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 0b100. DSRR0 points to instruction 6. No MASx register update. No ESR update.Debug counter update occurs for instructions 1–5 as appropriate. No debug counter or event updates for instruction 6.
	Instruction (1): DVC DACx Instruction (5): DTLB Error, with or without DAC	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 0b011. DSRR0 points to instruction 5. No MASx register update. No ESR update. Debug counter update occurs for instructions 1–4 as appropriate. No debug counter or event updates for instructions 5–6. **Note:** In this case the 2nd ld/st exception is masked. This behavior is implementation dependent and may differ on other CPUs.
	Instruction (1): DVC DACx Instruction (5): DSI, with or without DAC	
	Instruction (1): DVC DACx Instruction (5): DACy	Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 0b100. DSRR0 points to instruction 6. Debug counter update occurs for instructions 1–5 as appropriate. No debug counter or event updates for instruction 6. **Note:** In this case if x = y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the “no DACy” case.
	Instruction (1): DVC DACx Instruction (5): DVC DACy	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 0b100. DSRR0 points to instruction 6. No MASx register update. No ESR update. Debug counter update occurs for instructions 1–5 as appropriate. No debug counter or event updates for instruction 6. **Note:** In this case if x = y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the “no DACy” case.

Table 13-5 shows the fourth example update.

Table 13-5. DAC Events and Resultant Updates, Dual-issue Case 4

Instruction Sequence: The following pairs dual-issue: • (1) load/store (2) alu • (3) load/store (4) alu • (5) alu (6) load/store	**Event(s)**	**Result**
	Instruction (1): DTLB Error, no DAC	Take DTLB exception, no DBSR update, update MASx registers for 1st load/store instruction. Update ESR.
	Instruction (1): DSI, no DAC	Take DSI exception, no DBSR update, no MASx register update. Update ESR.
	Instruction (1): DTLB Error, with DACx	Take Debug exception, DBSR update setting DACx and IDE, DAC_OFST set to 0b000. DSRR0 points to instruction 1. No MASx register update. No ESR update.
	Instruction (1): DSI, with DACx	
	Instruction (1): DACx	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 0b000. DSRR0 points to instruction 2. No MASx register update. No ESR update.
	Instruction (1): DVC DACx No other exceptions	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 0b011. DSRR0 points to instruction 5. No MASx register update. No ESR update. Debug counter update occurs for instructions 1–4 as appropriate. No debug counter or event updates for instructions 5–6.
	Instruction (1): DVC DACx Instruction (3): DTLB Error, with or without DAC	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 3’b001. DSRR0 points to instruction 3. No MASx register update. No ESR update. Debug counter update occurs for instructions 1–2 as appropriate. No debug counter or event updates for instructions 3–6. **Note:** In this case the 2nd ld/st exception is masked. This behavior is implementation dependent and may differ on other CPUs.
	Instruction (1): DVC DACx Instruction (3): DSI, with or without DAC	
	Instruction (1): DVC DACx Instruction (3): DACy	Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 0b010. DSRR0 points to instruction 4. Debug counter update occurs for instructions 1–3 as appropriate. No debug counter or event updates for instructions 4–6. **Note:** In this case if x = y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the “no DACy” case.
	Instruction (1): DVC DACx Instruction (3): DVC DACy	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 0b011. DSRR0 points to instruction 5. No MASx register update. No ESR update. Debug counter update occurs for instructions 1–4 as appropriate. No debug counter or event updates for instructions 5–6. **Note:** In this case if x = y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the “no DACy” case.

13.2.3 Linked Instruction Address and Data Address Compare Event

Data address compare debug events may be linked with an instruction address compare event by setting the DAC1LNK and/or DAC2LNK control bits in DBCR2 to further refine when a data address compare debug event is generated. DAC1 may be linked with IAC1, and DAC2 (when not used as a mask or range bounds register) may be linked with IAC3. When linked, a DAC1 (or DAC2) debug event occurs when the same instruction which generates the DAC1 (or DAC2) hit also generates an IAC1 (or IAC3) hit. When linked, the IAC1 (or IAC3) event is not recorded in the debug status register, regardless of whether a corresponding DAC1 (or DAC2) event occurs, or whether the IAC1 (or IAC3) event enable is set.

When enabled and execution of a load or store class instruction results in a data access with an address that meets the criteria specified in the DBCR0, DBCR2, DBCR4, DAC1, DAC2, DVC1, and DVC2 registers, and the instruction also meets the criteria for generating an instruction address compare event, a linked data address compare debug event occurs. This event can occur and be recorded in DBSR regardless of the setting of MSR[DE]. The normal DAC1 and DAC2 status bits in the DBSR are used for recording these events. The IAC1 and IAC3 status bits are not set if the corresponding instruction address compare register is linked.

Linking is enabled using control bits in DBCR2. If data address compare debug events are used to control or modify operation of the debug counter, linking is also available, even though DBCR0 may not have enabled IAC or DAC events. Also, instruction address compare events which are linked may still affect the debug counter (if enabled to), thus may be used to either trigger a counter, or be counted, in contrast to being blocked from affecting the DBSR.

NOTE

Linked DAC events will not be recorded or counted if a load multiple word or store multiple word type instruction is interrupted prior to completion by a critical input or external input interrupt.

13.2.4 Trap Debug Event

A trap debug event (TRAP) occurs if trap debug events are enabled (DBCR0[TRAP] = 1), a trap instruction (**tw**, **twi**) is executed, and the conditions specified by the instruction for the trap are met. This event can occur and be recorded in DBSR regardless of the setting of MSR[DE]. When a trap debug event occurs, DBSR[TRAP] is set to record the debug exception.

13.2.5 Branch Taken Debug Event

A branch taken debug event (BRT) occurs if branch taken debug events are enabled (DBCR0[BRT] = 1), execution is attempted of a branch instruction that will be taken (either an unconditional branch or a conditional branch whose branch condition is true), and MSR[DE] = 1 or DBCR0[EDM] = 1. Branch taken debug events are not recognized if MSR[DE] = 0 and DBCR0[EDM] = 0 at the time of execution of the branch instruction and thus DBSR[IDE] can not be set by a branch taken debug event. When a branch taken debug event is recognized, DBSR[BRT] is set to record the debug exception, and the address of the branch instruction is recorded in DSRR0.

13.2.6 Instruction Complete Debug Event

An instruction complete debug event (ICMP) occurs if instruction complete debug events are enabled (DBCR0[ICMP] = 1), execution of any instruction is completed, and MSR[DE] = 1 or DBCR0[EDM] = 1. If execution of an instruction is suppressed due to the instruction causing some other exception which is enabled to generate an interrupt, then the attempted execution of that instruction does not cause an instruction complete debug event. The **sc** instruction does not fall into the category of an instruction whose execution is suppressed, since the instruction actually executes and then generates a system call interrupt. In this case, the instruction complete debug exception is also set. When an instruction complete debug event is recognized, DBSR[ICMP] is set to record the debug exception, and the address of the next instruction to be executed will be recorded in DSRR0.

Instruction complete debug events are not recognized if MSR[DE] = 0 and DBCR0[EDM] = 0 at the time of execution of the instruction. DBSR[IDE] is not generally set by an ICMP debug event.

When an EFPU round exception occurs, the DBSR[ICMP] and DBSR[IDE] are set. Because the instruction is by definition completed (SRR0 points to the following instruction), this interrupt takes higher priority than the debug interrupt so as not to be lost. DBSR[IDE] is set to indicate the imprecise recognition of a debug interrupt. In this case, the debug interrupt is taken with SRR0 pointing to the instruction following the instruction that generated the EFPU round exception, and DSRR0 points to the round exception handler. In addition to occurring when DBCR0[IDM] = 1, this circumstance can also occur when DBCR0[EDM] = 1.

NOTE

> Instruction complete debug events are not generated by the execution of an instruction that sets MSR[DE] 1 while DBCR0[ICMP] = 1, nor by the execution of an instruction which sets DBCR0[ICMP] while MSR[DE] = 1 or DBCR0[EDM] = 1.

13.2.7 Interrupt Taken Debug Event

An interrupt taken debug event (IRPT) occurs if interrupt taken debug events are enabled (DBCR0[IRPT] = 1) and a noncritical interrupt occurs. Only noncritical class interrupts cause an interrupt taken debug event. This event can occur and be recorded in DBSR regardless of the setting of MSR[DE]. When an interrupt taken debug event occurs, DBSR[IRPT] is set to record the debug exception. The value saved in DSRR0 is the address of the noncritical interrupt handler.

13.2.8 Critical Interrupt Taken Debug Event

A critical interrupt taken debug event (CIRPT) occurs if critical interrupt taken debug events are enabled (DBCR0[CIRPT] = 1) and a critical interrupt (other than a debug interrupt when the debug unit is disabled) occurs. Only critical class interrupts cause a critical interrupt taken debug event. This event can occur and be recorded in DBSR regardless of the setting of MSR[DE]. When a critical interrupt taken debug event occurs, DBSR[CIRPT] is set to record the debug exception. The value saved in DSRR0 is the address of the critical interrupt handler. Note that this debug event should not normally be enabled unless the debug unit is also enabled to avoid corruption of CSRR0/1.

13.2.9 Return Debug Event

A return debug event (RET) occurs if return debug events are enabled (DBCR0[RET] = 1) and an attempt is made to execute an **rfi** or **se_rfi** instruction. This event can occur and be recorded in DBSR regardless of the setting of MSR[DE]. When a return debug event occurs, DBSR[RET] is set to record the debug exception.

If MSR[DE] = 0 and DBCR0[EDM] = 0 at the time of the execution of the **rfi** or **se_rfi** (i.e. before the MSR is updated by the **rfi** or **se_rfi**), DBSR[IDE] is also set to record the imprecise debug event.

If MSR[DE] = 1 at the time of the execution of the **rfi** or **se_rfi**, a debug interrupt occurs provided that no higher priority exception is enabled to cause an interrupt. Debug save/restore register 0 is set to the address of the **rfi** or **se_rfi** instruction.

13.2.10 Critical Return Debug Event

A critical return debug event (CRET) occurs if critical return debug events are enabled (DBCR0[CRET] = 1) and an attempt is made to execute an **rfci** or **se_rfci** instruction. This event can occur and be recorded in DBSR regardless of the setting of MSR[DE]. When a critical return debug event occurs, DBSR[CRET] is set to record the debug exception.

If MSR[DE] = 0 and DBCR0[EDM] = 0 at the time of the execution of the **rfci** or **se_rfci** (i.e. before the MSR is updated by the **rfci** or **se_rfci**), DBSR[IDE] is also set to record the imprecise debug event.

If MSR[DE] = 1 at the time of the execution of the **rfci** or **se_rfci**, a debug interrupt will occur provided there exists no higher priority exception which is enabled to cause an interrupt. Debug save/restore register 0 is set to the address of the **rfci** or **se_rfci** instruction. Note that this debug event should not normally be enabled unless the debug unit is also enabled to avoid corruption of CSRR0/1.

13.2.11 Debug Counter Debug Event

A debug counter debug event (DCNT1, DCNT2) occurs if debug counter debug events are enabled (DBCR0[DCNT1] = 1 or DBCR0[DCNT2] = 1), a debug counter is enabled, and a counter decrements to zero. This event can occur and be recorded in DBSR regardless of the setting of MSR[DE]. When a debug counter debug event occurs, DBSR[DCNT{1,2}] is set to record the debug exception.

13.2.12 External Debug Event

An external debug event (DEVT1, DEVT2) occurs if External debug events are enabled (DBCR0[DEVT1] = 1 or DBCR0[DEVT2] = 1), and the respective **p_devt1** or **p_devt2** input signal transitions to the asserted state. This event can occur and be recorded in DBSR regardless of the setting of MSR[DE]. When an external debug event occurs, DBSR[DEVT{1,2}] is set to record the debug exception.

13.2.13 Unconditional Debug Event

An unconditional debug event (UDE) occurs when the unconditional debug event (**p_ude**) input transitions to the asserted state, and either DBCR0[IDM] = 1 or DBCR0[EDM] = 1. The unconditional debug event is the only debug event that does not have a corresponding enable bit for the event in DBCR0. This event can occur and be recorded in DBSR regardless of the setting of MSR[DE]. When an unconditional debug event occurs, DBSR[UDE] is set to record the debug exception.

13.3 Debug Registers

This section describes debug-related registers that are software accessible. These registers are intended for use by special debug tools and debug software, not by general application code.

Access to these registers (other than DBSR) by software is conditioned by the external debug mode control bit (DBCR0[EDM]/EDBCR0[EDM]) and the settings of debug control register DBERC0, which can be set by the hardware debug port. If DBCR0[EDM] is set and if the bit in DBERC0 corresponding to the resource is cleared, software is prevented from modifying debug register values other than in DBSR, since the resource is not owned by software. Software always has ownership of DBSR. Execution of an **mtspr** instruction targeting a debug register or register field not owned by software does not cause modifications to occur, and no exception is signaled. In addition, since the external debugger hardware may be manipulating debug register values, the state of these registers or register fields not owned by software is not guaranteed to be consistent if accessed (read) by software with a **mfspr** instruction, except for DBCR0[EDM] itself and the DBERC0 register.

Hardware always has full access to all registers and all register fields through the OnCE register access mechanism. The debug firmware must properly use read-modify-write operations to modify these registers to implement any control sharing with software. The debug firmware should consider settings in DBERC0 in order to preserve software settings of control registers as appropriate when hardware modifications to the debug registers is performed.

13.3.1 Debug Address and Value Registers

Instruction address compare registers, IAC1–8, are used to hold instruction addresses for address comparison purposes. In addition, IAC2 and IAC4 hold mask information for IAC1 and IAC3 respectively and IAC6 and IAC8 hold mask information for IAC5 and IAC7 respectively, when address bit match compare modes are selected. Note that when performing instruction address compares, the low order two address bits of the instruction address and the corresponding IAC register are ignored for Power ISA instruction pages, and the low order bit of the instruction address and the corresponding IAC register is ignored for VLE instruction pages.

Data address compare registers, DAC1 and DAC2, are used to hold data access addresses for address comparison purposes. In addition, DAC2 holds mask information for DAC1 when address bit match compare mode is selected.

Data value compare registers, DVC1 and DVC2, are used to hold data values for data comparison purposes. DVC1 and DVC2 are 64-bit registers. Data value comparisons are used to qualify data address compare debug events. DVC1 is associated with DAC1, and DVC2 is associated with DAC2. The most significant byte of the DVC1(2) register (labeled B0 in Figure 13-2) corresponds to the byte data value transferred to/from memory byte offset 0, 8, ..., and the least significant byte of the register (labeled B7 in Figure 13-2) corresponds to byte offset 7, F, ... When enabled for performing data value comparisons, each enabled byte in DVC1(2) is compared with the memory value transferred on the corresponding active byte lane of the data memory interface to determine if a match occurs. Inactive byte lanes do not participate in the comparison, they are implicitly masked. Table 11-11 shows active byte lanes for data transfers. Software must also program the DVC1(2) register byte positions based on the endian mode and alignment of the access. Misaligned accesses are not fully supported, since the data address and data value comparisons are only performed on the initial access in the case of a misaligned access; thus, accesses which cross a 64-bit boundary cannot be fully matched. For address and size combinations which involve two transfers, only the initial transfer is used for data address and value matching. DVC1 and DVC2 may be read or written using **mtspr** and **mfspr** instructions. All 64-bits of the GPR will be accessed, regardless of the value of MSR[SPE].

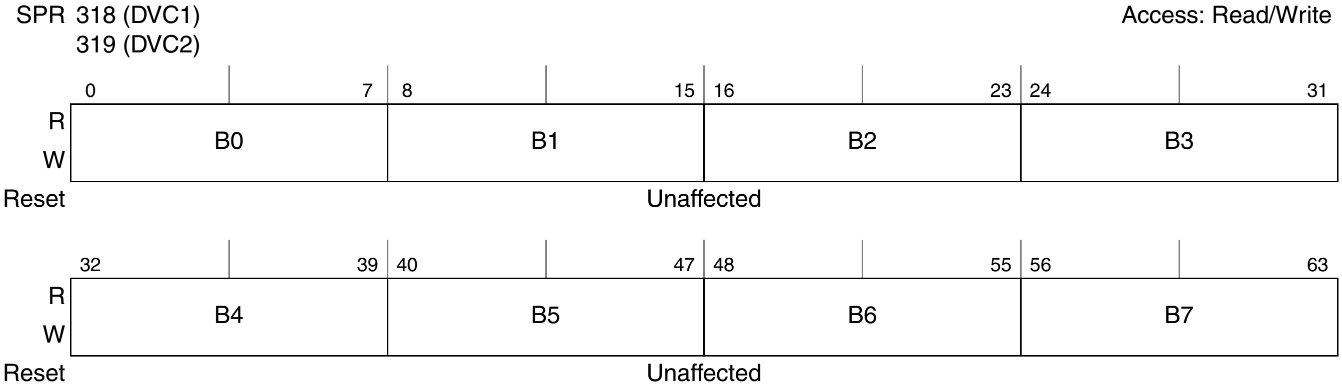


Figure 13-2. DVC1, DVC2 Registers

13.3.2 Debug Counter Register (DBCNT)

The debug counter register (DBCNT) contains two 16-bit counters (CNT1 and CNT2), which can be configured to operate independently or concatenated into a single 32-bit counter. Each counter can be configured to count down (decrement) when one or more count-enabled events occur. The counters operate regardless of whether counters are enabled to generate debug exceptions. When a count value reaches zero, a debug count event is signaled, and a debug event can be generated (if enabled). Upon reaching zero, the counter(s) are frozen. A debug counter signals an event on the transition from a value of one to a final value of zero. Loading a value of zero into the counter prevents the counter from counting. The debug counter is configured by the contents of debug control register 3. Figure 13-3 shows the DBCNT register.

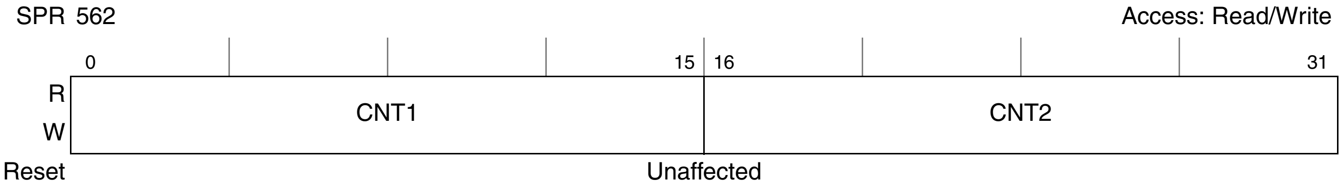


Figure 13-3. DBCNT Register

Refer to Section 13.3.3.4, "Debug Control Register 3 (DBCR3)," for more information about updates to the DBCNT register. Certain caveats exist on how the DBCNT and DBCR3 register are modified when one or more counters are enabled.

13.3.3 Debug Control and Status Registers

The debug control registers (DBCR0–6 and DBERC0) are used to enable debug events, reset the processor, control timer operation during debug events, and set the debug mode of the processor. The debug status register (DBSR) records debug exceptions while internal or external debug mode is enabled.

The e200z7requires that a context synchronizing instruction follow a **mtspr** DBCR0–6 or DBSR to ensure that any alterations enabling/disabling debug events are effective. The context synchronizing instruction may or may not be affected by the alteration. Typically, an **isync** instruction is used to create a synchronization boundary beyond which it can be guaranteed that the newly written control values are in effect.

For watchpoint generation and counter operation, configuration settings contained in DBCR1–5 are used, even though the corresponding event(s) may be disabled (via DBCR0) from setting DBSR flags.

13.3.3.1 Debug Control Register 0 (DBCR0)

Debug control register 0 enables debug modes and controls which debug events are allowed to set DBSR or EDBSR0 flags. The e200z7 adds some implementation specific bits to this register, as seen in Figure 13-4.

SPR 308 Access: Read/Write

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R W	EDM	IDM	RST		ICMP	BRT	IRPT	TRAP	IAC1	IAC2	IAC3	IAC4	DAC1		DAC2	
Reset	All zeros[1]															

	16	17	18	19	20	21	22	23	24	25	26	27			30	31
R W	RET	IAC5	IAC6	IAC7	IAC8	DEVT 1	DEVT 2	DCNT 1	DCNT 2	CIRPT	CRET	—				FT
Reset	All zeros															

Figure 13-4. DBCR0 Register

[1] DBCR0[EDM] is affected by **j_trst_b** or **m_por** assertion and remains reset while in the Test_Logic_Reset state, but it is not affected by **p_reset_b**. All other bits are reset by processor reset **p_reset_b** if DBCR0[EDM] = 0, as well as unconditionally by **m_por**. If DBCR0[EDM]=1, DBERC0 masks off hardware-owned resources (other than RST) from reset by **p_reset_b**, and only software-owned resources indicated by DBERC0 and the DBCR0[RST] field will be reset by **p_reset_b**. DBCR0[RST] is always reset by **p_reset_b** regardless of the value of DBCR0[EDM].

Table 13-6 provides bit definitions for debug control register 0.

Table 13-6. DBCR0 Bit Definitions

Bit(s)	Name	Description
0	EDM	External Debug Mode. This bit is read-only by software. 0 External debug mode disabled. Internal debug events not mapped into external debug events. 1 External debug mode enabled. Events will not cause the CPU to vector to interrupt code. Software is not permitted to write to debug registers {DBCR0, DBCNT, IAC1, DAC1–2} unless permitted by settings in EDBCR0. Hardware-owned events set status bits in EDBSR0. When external debug mode is enabled, hardware-owned resources in debug registers are not affected by processor reset **p_reset_b**. This allows the debugger to set up hardware debug events which remain active across a processor reset.
		Programming Notes: • It is recommended that debug status bits in the debug status registers be cleared before disabling external debug mode to avoid any internal imprecise debug interrupts. • Software may use this bit to determine if external debug has control over the debug registers. • The hardware debugger must set the EDM bit before other bits in this register (and other debug registers) may be altered. On the initial setting. all other bits are unchanged. This bit is only writable through the OnCE port.
1	IDM	Internal Debug Mode 0 Debug exceptions are disabled. Debug events do not affect DBSR. 1 Debug exceptions are enabled. Enabled debug events owned by software update the DBSR. If MSR[DE] = 1, the occurrence of a debug event,or the recording of an earlier debug event in the debug status register when MSR[DE] was cleared causes a debug interrupt.
2–3	RST	Reset Control 00 No function 01 **p_dbrstc[1]** pin asserted by debug reset control. Allows external device to initiate processor or system reset 10 **p_dbrstc[0]** pin asserted by debug reset control. Allows external device to initiate processor or system reset. 11 Reserved
4	ICMP	Instruction Complete Debug Event Enable 0 ICMP debug events are disabled 1 ICMP debug events are enabled
5	BRT	Branch Taken Debug Event Enable 0 BRT debug events are disabled 1 BRT debug events are enabled
6	IRPT	Interrupt Taken Debug Event Enable 0 IRPT debug events are disabled 1 IRPT debug events are enabled
7	TRAP	Trap Taken Debug Event Enable 0 TRAP debug events are disabled 1 TRAP debug events are enabled
8	IAC1	Instruction Address Compare 1 Debug Event Enable 0 IAC1 debug events are disabled 1 IAC1 debug events are enabled
9	IAC2	Instruction Address Compare 2 Debug Event Enable 0 IAC2 debug events are disabled 1 IAC2 debug events are enabled

Table 13-6. DBCR0 Bit Definitions (continued)

Bit(s)	Name	Description
10	IAC3	Instruction Address Compare 3 Debug Event Enable 0 IAC3 debug events are disabled 1 IAC3 debug events are enabled
11	IAC4	Instruction Address Compare 4 Debug Event Enable 0 IAC4 debug events are disabled 1 IAC4 debug events are enabled
12–13	DAC1	Data Address Compare 1 Debug Event Enable 00 DAC1 debug events are disabled 01 DAC1 debug events are enabled only for store-type data storage accesses 10 DAC1 debug events are enabled only for load-type data storage accesses 11 DAC1 debug events are enabled for load-type or store-type data storage accesses
14–15	DAC2	Data Address Compare 2 Debug Event Enable 00 DAC2 debug events are disabled 01 DAC2 debug events are enabled only for store-type data storage accesses 10 DAC2 debug events are enabled only for load-type data storage accesses 11 DAC2 debug events are enabled for load-type or store-type data storage accesses
16	RET	Return Debug Event Enable 0 RET debug events are disabled 1 RET debug events are enabled
17	IAC5	Instruction Address Compare 5 Debug Event Enable 0 IAC5 debug events are disabled 1 IAC5 debug events are enabled
18	IAC6	Instruction Address Compare 6 Debug Event Enable 0 IAC6 debug events are disabled 1 IAC6 debug events are enabled
19	IAC7	Instruction Address Compare 7 Debug Event Enable 0 IAC7 debug events are disabled 1 IAC7 debug events are enabled
20	IAC8	Instruction Address Compare 8 Debug Event Enable 0 IAC8 debug events are disabled 1 IAC8 debug events are enabled
21	DEVT1	External Debug Event 1 Enable 0 DEVT1 debug events are disabled 1 DEVT1 debug events are enabled
22	DEVT2	External Debug Event 2 Enable 0 DEVT2 debug events are disabled 1 DEVT2 debug events are enabled
23	DCNT1	Debug Counter 1 Debug Event Enable 0 Counter 1 debug events are disabled 1 Counter 1 debug events are enabled
24	DCNT2	Debug Counter 2 Debug Event Enable 0 Counter 2 debug events are disabled 1 Counter 2 debug events are enabled

Table 13-6. DBCR0 Bit Definitions (continued)

Bit(s)	Name	Description
25	CIRPT	Critical Interrupt Taken Debug Event Enable 0 CIRPT debug events are disabled 1 CIRPT debug events are enabled
26	CRET	Critical Return Debug Event Enable 0 CRET debug events are disabled 1 CRET debug events are enabled
27–30	—	Reserved
31	FT	Freeze Timers on Debug Event 0 Time base timers are unaffected by set DBSR/EDBSR0 bits 1 Disable clocking of time base timers if any DBSR bit is set (any EDBSR0 bit set if DBCR0[FT] owned by hardware) except MRR or CNT1TRG

13.3.3.2 Debug Control Register 1 (DBCR1)

Debug control register 1 is used to configure instruction address compare operation. The DBCR1 register is shown in Figure 13-5.

SPR 309 Access: Read/Write

	0 1	2 3	4 5	6 7	8 9	10 15	16 17	18 19	20 21	22 23	24 25	26 31
R W	IAC1US	IAC1ER	IAC2US	IAC2ER	IAC12M	—	IAC3US	IAC3ER	IAC4US	IAC4ER	IAC34M	—
Reset	All zeros[1]											

Figure 13-5. DBCR1 Register

[1] Reset by processor reset **p_reset_b** if DBCR0[EDM] = 0, as well as unconditionally by **m_por**. If DBCR0[EDM] = 1, DBERC0 masks off hardware-owned resources from reset by **p_reset_b,** and only software-owned resources indicated by DBERC0 are reset by **p_reset_b.**

Table 13-7 provides bit definitions for debug control register 1.

Table 13-7. DBCR1 Bit Definitions

Bit(s)	Name	Description
0–1	IAC1US	Instruction Address Compare 1 User/Supervisor Mode 00 IAC1 debug events are not affected by MSR[PR] 01 Reserved 10 IAC1 debug events can only occur if MSR[PR] = 0 (supervisor mode). 11 IAC1 debug events can only occur if MSR[PR] = 1 (user mode).
2–3	IAC1ER	Instruction Address Compare 1 Effective/Real Mode 00 IAC1 debug events are based on effective address. 01 Unimplemented in e200 (the Power ISA embedded category real address compare), no match can occur. 10 IAC1 debug events are based on effective address and can only occur if MSR[IS] = 0. 11 IAC1 debug events are based on effective address and can only occur if MSR[IS] = 1.

Table 13-7. DBCR1 Bit Definitions (continued)

Bit(s)	Name	Description
4–5	IAC2US	Instruction Address Compare 2 User/Supervisor Mode 00 IAC2 debug events are not affected by MSR[PR]. 01 Reserved 10 IAC2 debug events can only occur if MSR[PR] = 0 (supervisor mode). 11 IAC2 debug events can only occur if MSR[PR] = 1 (user mode).
6–7	IAC2ER	Instruction Address Compare 2 Effective/Real Mode 00 IAC2 debug events are based on effective address. 01 Unimplemented in e200 (the Power ISA embedded category real address compare), no match can occur 10 IAC2 debug events are based on effective address and can only occur if MSR[IS] = 0. 11 IAC2 debug events are based on effective address and can only occur if MSR[IS] = 1.
8–9	IAC12M	Instruction Address Compare 1/2 Mode 00 Exact address compare. IAC1 debug events can only occur if the address of the instruction fetch is equal to the value specified in IAC1. IAC2 debug events can only occur if the address of the instruction fetch is equal to the value specified in IAC2. 01 Address bit match. IAC1 debug events can occur only if the address of the instruction fetch, ANDed with the contents of IAC2 are equal to the contents of IAC1, also ANDed with the contents of IAC2. IAC2 debug events do not occur. IAC1US and IAC1ER settings are used. 10 Inclusive address range compare. IAC1 debug events can occur only if the address of the instruction fetch is greater than or equal to the value specified in IAC1 and less than the value specified in IAC2. IAC2 debug events do not occur. IAC1US and IAC1ER settings are used. 11 Exclusive address range compare. IAC1 debug events can occur only if the address of the instruction fetch is less than the value specified in IAC1 or is greater than or equal to the value specified in IAC2. IAC2 debug events do not occur. IAC1US and IAC1ER settings are used.
10–15	—	Reserved
16–17	IAC3US	Instruction Address Compare 3 User/Supervisor Mode 00 IAC3 debug events not affected by MSR[PR] 01 Reserved 10 IAC3 debug events can only occur if MSR[PR] = 0 (supervisor mode). 11 IAC3 debug events can only occur if MSR[PR] = 1 (user mode).
18–19	IAC3ER	Instruction Address Compare 3 Effective/Real Mode 00 IAC3 debug events are based on effective address. 01 Unimplemented in e200 (the Power ISA embedded category real address compare), no match can occur 10 IAC3 debug events are based on effective address and can only occur if MSR[IS] = 0 11 IAC3 debug events are based on effective address and can only occur if MSR[IS] = 1
20–21	IAC4US	Instruction Address Compare 4 User/Supervisor Mode 00 IAC4 debug events are not affected by MSR[PR]. 01 Reserved 10 IAC4 debug events can only occur if MSR[PR] = 0 (supervisor mode). 11 IAC4 debug events can only occur if MSR[PR] = 1 (user mode).
22–23	IAC4ER	Instruction Address Compare 4 Effective/Real Mode 00 IAC4 debug events are based on effective address 01 Unimplemented in e200 (the Power ISA embedded category real address compare), no match can occur 10 IAC4 debug events are based on effective address and can only occur if MSR[IS] = 0 11 IAC4 debug events are based on effective address and can only occur if MSR[IS] = 1

Table 13-7. DBCR1 Bit Definitions (continued)

Bit(s)	Name	Description
24–25	IAC34M	Instruction Address Compare 3/4 Mode 00 Exact address compare. IAC3 debug events can only occur if the address of the instruction fetch is equal to the value specified in IAC3. IAC4 debug events can only occur if the address of the instruction fetch is equal to the value specified in IAC4. 01 Address bit match. IAC3 debug events can occur only if the address of the instruction fetch, ANDed with the contents of IAC4 are equal to the contents of IAC3, also ANDed with the contents of IAC4. IAC4 debug events do not occur. IAC3US and IAC3ER settings are used. 10 Inclusive address range compare. IAC3 debug events can occur only if the address of the instruction fetch is greater than or equal to the value specified in IAC3 and less than the value specified in IAC4. IAC4 debug events do not occur. IAC3US and IAC3ER settings are used. 11 Exclusive address range compare. IAC3 debug events can occur only if the address of the instruction fetch is less than the value specified in IAC3 or is greater than or equal to the value specified in IAC4. IAC4 debug events do not occur. IAC3US and IAC3ER settings are used.
26–31	—	Reserved

13.3.3.3 Debug Control Register 2 (DBCR2)

Debug control register 2 is used to configure data address compare and data value compare operation. Figure 13-6 shows the DBCR2 register.

SPR 310 Access: Read/Write

	0 1	2 3	4 5	6 7	8 9	10	11	12 13	14 15	16 23	24 31
R W	DAC1 US	DAC1 ER	DAC2 US	DAC2 ER	DAC1 2M	DAC1 LNK	DAC2 LNK	DVC1 M	DVC2 M	DVC1BE	DVC2BE
Reset	All zeros[1]										

Figure 13-6. DBCR2 Register

[1] Reset by processor reset **p_reset_b** if DBCR0[EDM] = 0, as well as unconditionally by **m_por**. If DBCR0[EDM] = 1, DBERC0 masks off hardware-owned resources from reset by **p_reset_b,** and only software-owned resources indicated by DBERC0 are reset by **p_reset_b**.

Table 13-8 provides bit definitions for debug control register 2.

Table 13-8. DBCR2 Bit Definitions

Bit(s)	Name	Description
0–1	DAC1US	Data Address Compare 1 User/Supervisor Mode 00 DAC1 debug events are not affected by MSR[PR]. 01 Reserved 10 DAC1 debug events can only occur if MSR[PR] = 0 (supervisor mode). 11 DAC1 debug events can only occur if MSR[PR] = 1 (user mode).
2–3	DAC1ER	Data Address Compare 1 Effective/Real Mode 00 DAC1 debug events are based on effective address. 01 Unimplemented in Zen (Power ISA real address compare), no match can occur 10 DAC1 debug events are based on effective address and can only occur if MSR[DS] = 0. 11 DAC1 debug events are based on effective address and can only occur if MSR[DS] = 1.

Table 13-8. DBCR2 Bit Definitions (continued)

Bit(s)	Name	Description
4–5	DAC2US	Data Address Compare 2 User/Supervisor Mode. 00 DAC2 debug events are not affected by MSR[PR]. 01 Reserved 10 DAC2 debug events can only occur if MSR[PR] = 0 (supervisor mode). 11 DAC2 debug events can only occur if MSR[PR] = 1. (user mode).
6–7	DAC2ER	Data Address Compare 2 Effective/Real Mode 00 DAC2 debug events are based on effective address. 01 Unimplemented in Zen (Power ISA real address compare), no match can occur 10 DAC2 debug events are based on effective address and can only occur if MSR[DS] = 0 11 DAC2 debug events are based on effective address and can only occur if MSR[DS] = 1
8–9	DAC12M	Data Address Compare 1/2 Mode 00 Exact address compare. DAC1 debug events can only occur if the address of the data access is equal to the value specified in DAC1. DAC2 debug events can only occur if the address of the data access is equal to the value specified in DAC2. 01 Address bit match. DAC1 debug events can occur only if the address of the data access ANDed with the contents of DAC2, are equal to the contents of DAC1 also ANDed with the contents of DAC2. DAC2 debug events do not occur. DAC1US and DAC1ER settings are used. 10 Inclusive address range compare. DAC1 debug events can occur only if the address of the data access is greater than or equal to the value specified in DAC1 and less than the value specified in DAC2. DAC2 debug events do not occur. DAC1US and DAC1ER settings are used. 11 Exclusive address range compare. DAC1 debug events can occur only if the address of the data access is less than the value specified in DAC1 or is greater than or equal to the value specified in DAC2. DAC2 debug events do not occur. DAC1US and DAC1ER settings are used.
10	DAC1LNK	Data Address Compare 1 Linked 0 No effect 1 DAC1 debug events are linked to IAC1 debug events. IAC1 debug events do not affect DBSR. When linked to IAC1, DAC1 debug events are conditioned based on whether the instruction also generated an IAC1 debug event
11	DAC2LNK	Data Address Compare 2 Linked 0 No effect 1 DAC 2 debug events are linked to IAC3 debug events. IAC3 debug events do not affect DBSR When linked to IAC3, DAC2 debug events are conditioned based on whether the instruction also generated an IAC3 debug event. DAC2 can only be linked if DAC12M specifies Exact Address Compare because DAC2 debug events are not generated in the other compare modes.

Table 13-8. DBCR2 Bit Definitions (continued)

Bit(s)	Name	Description
12–13	DVC1M	Data Value Compare 1 Mode When DBCR4[DVC1C] = 0, it has the following settings: 00 DAC1 debug events not affected by data value compares. 01 DAC1 debug events can only occur when all bytes specified in the DVC1BE field match the corresponding data byte values for active byte lanes of the memory access. 10 DAC1 debug events can only occur when any byte specified in the DVC1BE field matches the corresponding data byte value for active byte lanes of the memory access. 11 DAC1 debug events can only occur when all bytes specified in the DVC1BE field within at least one of the half words of the data value of the memory access matches the corresponding DVC1 value. **Note:** Inactive byte lanes of the memory access are automatically masked. When DBCR4[DVC1C] = 1, it has the following settings: 00 Reserved 01 DAC1 debug events can only occur when any byte specified in the DVC1BE field does not match the corresponding data byte value for active byte lanes of the memory access. If all active bytes match, then no event will be generated. 10 DAC1 debug events can only occur when all bytes specified in the DVC1BE field do not match the corresponding data byte values for active byte lanes of the memory access. If any active byte match occurs, no event will be generated. 11 Reserved **Note:** Inactive byte lanes of the memory access are automatically masked.
14–15	DVC2M	Data Value Compare 2 Mode When DBCR4[DVC2C] = 0, it has the following settings: 00 DAC2 debug events not affected by data value compares. 01 DAC2 debug events can only occur when all bytes specified in the DVC2BE field match the corresponding data byte values for active byte lanes of the memory access. 10 DAC2 debug events can only occur when any byte specified in the DVC2BE field matches the corresponding data byte value for active byte lanes of the memory access. 11 DAC2 debug events can only occur when all bytes specified in the DVC2BE field within at least one of the half words of the data value of the memory access matches the corresponding DVC2 value. **Note:** Inactive byte lanes of the memory access are automatically masked. When DBCR4[DVC2C] = 1, it has the following settings: 00 Reserved 01 DAC2 debug events can only occur when any byte specified in the DVC2BE field does not match the corresponding data byte value for active byte lanes of the memory access. If all active bytes match, then no event will be generated. 10 DAC2 debug events can only occur when all bytes specified in the DVC2BE field do not match the corresponding data byte values for active byte lanes of the memory access. If any active byte match occurs, no event will be generated. 11 Reserved **Note:** Inactive byte lanes of the memory access are automatically masked.

Table 13-8. DBCR2 Bit Definitions (continued)

Bit(s)	Name	Description
16–23	DVC1BE	Data Value Compare 1 Byte Enables Specifies which bytes in the aligned double word value associated with the memory access are compared to the corresponding bytes in DVC1. Inactive byte lanes of a memory access smaller than 64-bits are automatically masked by hardware. If all bits in the DVC1BE field are clear, then a match will occur regardless of the data. Misaligned accesses which cross a double-word boundary are not fully supported. 1xxxxxxxByte lane 0 is enabled for comparison with the value in bits 0–7 of DVC1. x1xxxxxxByte lane 1 is enabled for comparison with the value in bits 8–15 of DVC1. xx1xxxxxByte lane 2 is enabled for comparison with the value in bits 16–23 of DVC1. xxx1xxxxByte lane 3 is enabled for comparison with the value in bits 24–31 of DVC1. xxxx1xxxByte lane 4 is enabled for comparison with the value in bits 32–39 of DVC1. xxxxx1xxByte lane 5 is enabled for comparison with the value in bits 40–47 of DVC1. xxxxxx1xByte lane 6 is enabled for comparison with the value in bits 48–55 of DVC1. xxxxxxx1Byte lane 7 is enabled for comparison with the value in bits 56–63 of DVC1.
24–31	DVC2BE	Data Value Compare2 Byte Enables Specifies which bytes in the aligned double word value associated with the memory access are compared to the corresponding bytes in DVC2. Inactive byte lanes of a memory access smaller than 64-bits are automatically masked by hardware. If all bits in the DVC1BE field are clear, then a match will occur regardless of the data. Misaligned accesses which cross a double-word boundary are not fully supported. 1xxxxxxxByte lane 0 is enabled for comparison with the value in bits 0–7 of DVC2. x1xxxxxxByte lane 1 is enabled for comparison with the value in bits 8–15 of DVC2. xx1xxxxxByte lane 2 is enabled for comparison with the value in bits 16–23 of DVC2. xxx1xxxxByte lane 3 is enabled for comparison with the value in bits 24–31 of DVC2. xxxx1xxxByte lane 4 is enabled for comparison with the value in bits 32–39 of DVC2. xxxxx1xxByte lane 5 is enabled for comparison with the value in bits 40–47 of DVC2. xxxxxx1xByte lane 6 is enabled for comparison with the value in bits 48–55 of DVC2. xxxxxxx1Byte lane 7 is enabled for comparison with the value in bits 56–63 of DVC2.

13.3.3.4 Debug Control Register 3 (DBCR3)

Debug control register 3 is used to enable and configure the debug counter and debug counter events. For counter operation, the specific debug events which cause counters to decrement are specified in DBCR3.

NOTE

The corresponding events do not need to be (and probably should not be) enabled in DBCR0.

The IAC1–IAC4 and DAC1–DAC2 control fields in DBCR0 are ignored for counter operations, and the control fields in DBCR3 determine when counting is enabled. DBCR1 and DBCR2 control fields are also used to determine the configuration of IAC1–4 and DAC1–2 operation for counting, even though corresponding events may be disabled via DBCR0. Multiple count-enabled events that occur during execution of an instruction typically cause only a single decrement of a counter. As an example, if more than one IAC or DAC register hits and is enabled for counting, only a single count occurs per counter. During **lmw** and **stmw** instructions, multiple DACx hits can occur. If the instruction is not interrupted prior to completion, a single decrement of a counter occurs. Note that if the counters are operating independently, both may count for the same instruction.

The debug counter register (DBCNT) is configured by DBCR3[CONFIG] to operate either as separate 16-bit counter 1 and counter 2 or as a combined 32-bit counter (using control bits in DBCR3 for counter 1). Counters are enabled whenever any of their respective count enable event control bits are set and either DBCR0[IDM] or DBCR0[EDM] is set. Counters are frozen during a hardware debug session (see Section 13.4.2, "OnCE Introduction"). Counter 1 may be configured to count down on a number of different debug events. Counter 2 is also configurable to count down on instruction complete, instruction or data address compare events, and external events.

Special capability is provided for counter 1 to be triggered to begin counting down by a subset of events (IAC1, IAC3, DAC1R, DAC1W, DEVT1, DEVT2, and counter 2). When one or more of the counter 1 trigger bits are set (IAC1T1, IAC3T1, DAC1RT1, DAC1WT1, DEVT1T1, DEVT2T1, CNT2T1), counter 1 is frozen until at least one of the triggering events occurs. It is then enabled to begin operation. Depending on the trigger source, if it is enabled for counting, the trigger event may be counted. Triggering status for counter 1 is provided in the debug status register or external debug status register 0. Triggering mode is enabled by a **mtspr** DBCR3, which sets one or more of the trigger enable bits and also enables counter 1. Once set, the trigger can be re-armed by clearing the DBSR[CNT1TRG] or EDBSR0[CNTITRG] status bit.

Most combinations of enables do not make sense and should be avoided. As an example, if DBCR3[ICMP] is set for counter 1, no other count enable should be set for counter 1. Conversely, multiple instruction address compare count enables are allowed to be set and may be useful.

Due to instruction pipelining issues and other constraints, most combinations of events are not supported for event counting. Only the following combinations are intended to be used:

- Any combination of IAC[1–4]
- Any combination of DAC[1–2] including linking
- Any combination of DEVT[1–2]
- Any combination of IRPT, RET

Limited support is provided for any combination of IAC[1–4] with DAC[1–2] (linked or unlinked). Note that these combinations may be reported in an imprecise fashion, with DBSR[IDE] set in such cases.

All other combinations are not supported.

Due to pipelining and detection of IAC events early in the pipeline and DAC events late in the pipeline, no guarantee is made on the exact instruction boundary that a debug exception will be generated when IAC and DAC events are combined for counting. This also applies to the case where counter 1 is being triggered by counter 2, and a combination of IAC and DAC events are being enabled for the counters, even if only one of these types is enabled for a particular counter. In general, when an IAC event logically follows closely behind a DAC event (within several instructions), it cannot be recognized immediately since the DAC event has not necessarily been generated in the pipeline at the time the IAC is seen, and thus the counter may not decrement to zero for the IAC event until after the instruction with the IAC (and perhaps several additional instructions) has proceeded down the execution pipeline. The instruction boundary where the debug exception is actually generated in this case will typically follow the IAC by up to several instructions.

Note that the counters will operate regardless of whether counters are enabled to generate debug exceptions.

If counter 2 is being used to trigger counter 1, counter 2 events should not normally be enabled in DBCR0, and will not be blocked.

NOTE

> Multiple IAC or DAC events will not be counted during a load multiple word or store multiple word type instruction, and no count will occur if either is interrupted by a critical input or external input interrupt prior to completion.

DBCR3, shown in Figure 13-7, is an e200z7-implementation-specific register.

SPR 561 Access: Read/Write

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R W	DEVT1C1	DEVT2C1	ICMPC1	IAC1C1	IAC2C1	IAC3C1	IAC4C1	DAC1RC1	DAC1WC1	DAC2RC1	DAC2WC1	IRPTC1	RETC1	DEVT1C2	DEVT2C2	ICMPC2
Reset	All zeros[1]															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R W	IAC1C2	IAC2C2	IAC3C2	IAC4C2	DAC1RC2	DAC1WC2	DAC2RC2	DAC2WC2	DEVT1T1	DEVT2T1	IAC1T1	IAC3T1	DAC1RT1	DAC1WT1	CNT2T1	CONFIG
Reset	All zeros[1]															

Figure 13-7. DBCR3 Register

[1] Reset by processor reset **p_reset_b** if DBCR0[EDM] = 0, as well as unconditionally by m_por. If DBCR0[EDM] = 1, DBERC0 masks off hardware-owned resources from reset by **p_reset_b** and only software-owned resources indicated by DBERC0 will be reset by **p_reset_b.**

Table 13-9 provides bit definitions for debug control register 3.

Table 13-9. DBCR3 Bit Definitions

Bit(s)	Name	Description
0	DEVT1C1	External Debug Event 1 Count 1 Enable 0 Counting DEVT1 debug events by Counter 1 is disabled 1 Counting DEVT1 debug events by Counter 1 is enabled
1	DEVT2C1	External Debug Event 2 Count 1 Enable 0 Counting DEVT2 debug events by Counter 1 is disabled 1 Counting DEVT2 debug events by Counter 1 is enabled
2	ICMPC1	Instruction Complete Debug Event Count 1 Enable 0 Counting ICMP debug events by Counter 1 is disabled 1 Counting ICMP debug events by Counter 1 is enabled **Note:** ICMP events are masked by MSR[DE] = 0 when operating in internal debug mode
3	IAC1C1	Instruction Address Compare 1 Debug Event Count 1 Enable 0 Counting IAC1 debug events by Counter 1 is disabled 1 Counting IAC1 debug events by Counter 1 is enabled
4	IAC2C1	Instruction Address Compare2 Debug Event Count 1 Enable 0 Counting IAC2 debug events by Counter 1 is disabled 1 Counting IAC2 debug events by Counter 1 is enabled

Table 13-9. DBCR3 Bit Definitions (continued)

Bit(s)	Name	Description
5	IAC3C1	Instruction Address Compare 3 Debug Event Count 1 Enable 0 Counting IAC3 debug events by Counter 1 is disabled 1 Counting IAC3 debug events by Counter 1 is enabled
6	IAC4C1	Instruction Address Compare 4 Debug Event Count 1 Enable 0 Counting IAC4 debug events by Counter 1 is disabled 1 Counting IAC4 debug events by Counter 1 is enabled
7	DAC1RC1	Data Address Compare 1 Read Debug Event Count 1 Enable[1] 0 Counting DAC1R debug events by Counter 1 is disabled 1 Counting DAC1R debug events by Counter 1 is enabled
8	DAC1WC1	Data Address Compare 1 Write Debug Event Count 1 Enable[1] 0 Counting DAC1W debug events by Counter 1 is disabled 1 Counting DAC1W debug events by Counter 1 is enabled
9	DAC2RC1	Data Address Compare 2 Read Debug Event Count 1 Enable[1] 0 Counting DAC2R debug events by Counter 1 is disabled 1 Counting DAC2R debug events by Counter 1 is enabled
10	DAC2WC1	Data Address Compare 2 Write Debug Event Count 1 Enable[1] 0 Counting DAC2W debug events by Counter 1 is disabled 1 Counting DAC2W debug events by Counter 1 is enabled
11	IRPTC1	Interrupt Taken Debug Event Count 1 Enable 0 Counting IRPT debug events by Counter 1 is disabled 1 Counting IRPT debug events by Counter 1 is enabled
12	RETC1	Return Debug Event Count 1 Enable 0 Counting RET debug events by Counter 1 is disabled 1 Counting RET debug events by Counter 1 is enabled
13	DEVT1C2	External Debug Event 1 Count 2 Enable 0 Counting DEVT1 debug events by Counter 2 is disabled 1 Counting DEVT1 debug events by Counter 2 is enabled
14	DEVT2C2	External Debug Event 2 Count 2 Enable 0 Counting DEVT2 debug events by Counter 2 is disabled 1 Counting DEVT2 debug events by Counter 2 is enabled
15	ICMPC2	Instruction Complete Debug Event Count 2 Enable 0 Counting ICMP debug events by Counter 2 is disabled 1 Counting ICMP debug events by Counter 2 is enabled **Note:** ICMP events are masked by MSR[DE] = 0 when operating in Internal Debug Mode
16	IAC1C2	Instruction Address Compare 1 Debug Event Count 2 Enable 0 Counting IAC1 debug events by Counter 2 is disabled 1 Counting IAC1 debug events by Counter 2 is enabled
17	IAC2C2	Instruction Address Compare2 Debug Event Count 2 Enable 0 Counting IAC2 debug events by Counter 2 is disabled 1 Counting IAC2 debug events by Counter 2 is enabled
18	IAC3C2	Instruction Address Compare 3 Debug Event Count 2 Enable 0 Counting IAC3 debug events by Counter 2 is disabled 1 Counting IAC3 debug events by Counter 2 is enabled

Table 13-9. DBCR3 Bit Definitions (continued)

Bit(s)	Name	Description
19	IAC4C2	Instruction Address Compare 4 Debug Event Count 2 Enable 0 Counting IAC4 debug events by Counter 2 is disabled 1 Counting IAC4 debug events by Counter 2 is enabled
20	DAC1RC2	Data Address Compare 1 Read Debug Event Count 2 Enable[1] 0 Counting DAC1R debug events by Counter 2 is disabled 1 Counting DAC1R debug events by Counter 2 is enabled
21	DAC1WC2	Data Address Compare 1 Write Debug Event Count 2 Enable[1] 0 Counting DAC1W debug events by Counter 2 is disabled 1 Counting DAC1W debug events by Counter 2 is enabled
22	DAC2RC2	Data Address Compare 2 Read Debug Event Count 2 Enable[1] 0 Counting DAC2R debug events by Counter 2 is disabled 1 Counting DAC2R debug events by Counter 2 is enabled
23	DAC2WC2	Data Address Compare 2 Write Debug Event Count 2 Enable[1] 0 Counting DAC2W debug events by Counter 2 is disabled 1 Counting DAC2W debug events by Counter 2 is enabled
24	DEVT1T1	External Debug Event 1 Trigger Counter 1 Enable 0 No effect 1 A DEVT1 debug event will trigger Counter 1 operation
25	DEVT2T1	External Debug Event 2 Trigger Counter 1 Enable 0 No effect 1 A DEVT2 debug event will trigger Counter 1 operation
26	IAC1T1	Instruction Address Compare 1 Trigger Counter 1 Enable 0 No effect 1 An IAC1 debug event will trigger Counter 1 operation
27	IAC3T1	Instruction Address Compare 3 Trigger Counter 1 Enable 0 No effect 1 An IAC3 debug event will trigger Counter 1 operation
28	DAC1RT1	Data Address Compare 1 Read Trigger Counter 1 Enable 0 No effect 1 A DAC1R debug event will trigger Counter 1 operation
29	DAC1WT1	Data Address Compare 1 Write Trigger Counter 1 Enable 0 No effect 1 A DAC1W debug event will trigger Counter 1 operation
30	CNT2T1	Debug Counter 2 Trigger Counter 1 Enable 0 No effect 1 Counter 2 decrementing to a value of 0 will trigger Counter 1 operation
31	CONFIG	Debug Counter Configuration 0 Counter 1 and Counter 2 are independent counters 1 Counter 1 and Counter 2 are concatenated into a single 32-bit counter. The event count control bits for Counter 1 are used and the event count control bits for Counter 2 are ignored.

[1] If the DACx field in DBCR0 is set to restrict events to only reads or only writes, only those events will be counted if enabled in DBCR3. In general, DAC events should be disabled in DBCR0.

NOTE

Updates to the DBCR0, DBSR, DBCR3, and DBCNT registers should be performed carefully if the counters are currently enabled for counting events. It is possible for the instruction that updates the counters or control over the counters to cause one or more counter events to occur (DCNT1, DCNT2, CNT1TRG), even if the result of the instruction is to modify the counter value or control value to a state where counter events would not be expected to occur. This is due to the pipelined nature of the counter and control operation.

For example, if a counter was enabled to count ICMP events, MSR[DE] = 1, and the value of the counter is 1 prior to execution of the **mtspr** instruction that loads the counter with a different value, a counter event is generated after completion of the **mtspr**, even though the counter ends up being loaded with a new value. At the end of the **mtspr** instruction, a debug event is posted, but the counter value is that of the newly written count value. In addition, no decrement of the new counter value is performed at the completion of an **mtspr** instruction which modifies a counter, regardless of whether a debug event is generated based on the old counter value.

To avoid this, it is recommended that the DBCNT and DBCR3 values be modified only when no possibility of a counter related debug event on the **mtspr** instruction is possible. Modifying DBCR0 to affect counter event enabling/disabling may have similar issues, as may modifying the DBSR[CNT1TRG].

As another example, if a counter was enabled to count ICMP events, MSR[DE] = 1, and the value of the counter is 1 prior to execution of the **mtspr** instruction that loads DBCR3 with a different value, a counter event may be generated following completion of the **mtspr**, even though DBCR3 ends up being loaded with a new value which is disabling the particular event from being counted. At the end of the **mtspr** instruction, a debug event is posted, but the DBCR3 value reflects the newly established control, which may indicate that the particular event is not to cause a counter update. Modifying DBCR0 to affect counter event enabling/disabling may have similar issues, as may modifying DBSR[CNT1TRG].

13.3.3.5 Debug Control Register 4 (DBCR4)

Debug control register 4 is used to extend data address and value compare matching functionality. DBCR4 is shown in Figure 13-8.

SPR 563 Access: Read/Write

	0	1	2	3	4–15	16–19	20–23	24–31
R W	—	DVC1C	—	DVC2C	—	DAC1XM	DAC2XM	—
Reset	All zeros[1]							

Figure 13-8. DBCR4 Register

[1] DBCR4 is reset by processor reset **p_reset_b** if DBCR0[EDM] = 0, as well as unconditionally by **m_por**. If DBCR0[EDM] = 1, DBERC0 masks off hardware-owned resources from reset by **p_reset_b.** Only software-owned resources indicated by DBERC0 are reset by **p_reset_b**.

Table 13-10 provides bit definitions for debug control register 4.

Table 13-10. DBCR4 Bit Definitions

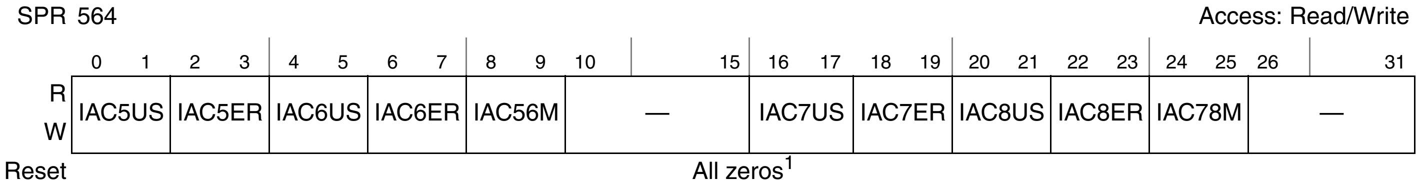
Bit(s)	Name	Description
0	—	Reserved
1	DVC1C	Data Value Compare 1 Control 0 Normal DVC1 operation. 1 Inverted polarity DVC1 operation DVC1C controls whether DVC1 data value comparisons utilize the normal Power ISA operation, or an alternate “inverted compare” operation. In inverted polarity mode, data value compares perform a not-equal comparison. See details in the DBCR2 register definition
2	—	Reserved
3	DVC2C	Data Value Compare 2 Control 0 Normal DVC2 operation. 1 Inverted polarity DVC2 operation DVC2C controls whether DVC2 data value comparisons utilize the normal Power ISA operation, or an alternate “inverted compare” operation. In inverted polarity mode, data value compares perform a not-equal comparison. See details in the DBCR2 register definition
4–15	—	Reserved
16–19	DAC1XM	Data Address Compare 1 Extended Mask Control 0000 No additional masking when DBCR2[DAC12M] = 00 0001–1100 Exact Match Bit Mask. Number of low order bits masked in DAC1 when comparing the storage address with the value in DAC1 for exact address compare (DBCR2[DAC12M] = 00). Ranges up to 4KB are supported. 1101–1111 Reserved DAC1XM allows for binary power of 2 address range compares for DAC1 without requiring the use of DAC2.

Table 13-10. DBCR4 Bit Definitions (continued)

Bit(s)	Name	Description
20–23	DAC2XM	Data Address Compare 2 Extended Mask Control 0000—No additional masking when DBCR2[DAC12M] = 00 0001–1100 Exact Match Bit Mask. Number of low order bits masked in DAC2 when comparing the storage address with the value in DAC2 for exact address compare (DBCR2[DAC12M] = 00). Ranges up to 4 KB are supported. 1101–1111 Reserved DAC2XM allows for binary power of 2 address range compares for DAC2 without requiring the use of DAC1.
24–31	—	Reserved

13.3.3.6 Debug Control Register 5 (DBCR5)

Debug control register 5 is used to configure instruction address compare operation for IAC5–8. The DBCR5 register is shown in Figure 13-9.



SPR 564 Access: Read/Write

	0–1	2–3	4–5	6–7	8–9	10–15	16–17	18–19	20–21	22–23	24–25	26–31
R W	IAC5US	IAC5ER	IAC6US	IAC6ER	IAC56M	—	IAC7US	IAC7ER	IAC8US	IAC8ER	IAC78M	—
Reset	All zeros[1]											

Figure 13-9. DBCR5 Register

[1] Reset by processor reset **p_reset_b** if DBCR0[EDM] = 0, as well as unconditionally by **m_por**. If DBCR0[EDM] = 1, DBERC0 masks off hardware-owned resources from reset by **p_reset_b** and only software-owned resources indicated by DBERC0 will be reset by **p_reset_b**.

Table 13-11 provides bit definitions for debug control register 5.

Table 13-11. DBCR5 Bit Definitions

Bit(s)	Name	Description
0–1	IAC5US	Instruction Address Compare 5 User/Supervisor Mode 00 IAC5 debug events are not affected by MSR[PR]. 01 Reserved 10 IAC5 debug events can only occur if MSR[PR] = 0 (supervisor mode). 11 IAC5 debug events can only occur if MSR[PR] = 1 (user mode).
2–3	IAC5ER	Instruction Address Compare 5 Effective/Real Mode 00 IAC5 debug events are based on effective address. 01 Unimplemented in the e200 (the Power ISA embedded category real address compare), no match can occur 10 IAC5 debug events are based on effective address and can only occur if MSR[IS] = 0. 11 IAC5 debug events are based on effective address and can only occur if MSR[IS] = 1.
4–5	IAC6US	Instruction Address Compare 6 User/Supervisor Mode 00 IAC6 debug events are not affected by MSR[PR]. 01 Reserved 10 IAC6 debug events can only occur if MSR[PR] = 0 (supervisor mode). 11 IAC6 debug events can only occur if MSR[PR] = 1 (user mode).

Table 13-11. DBCR5 Bit Definitions (continued)

Bit(s)	Name	Description
6–7	IAC6ER	Instruction Address Compare 6 Effective/Real Mode 00 IAC6 debug events are based on effective address. 01 Unimplemented in the e200 (the Power ISA embedded category real address compare), no match can occur 10 IAC6 debug events are based on effective address and can only occur if MSR[IS] = 0. 11 IAC6 debug events are based on effective address and can only occur if MSR[IS] = 1.
8–9	IAC56M	Instruction Address Compare 5/6 Mode 00 Exact address compare. IAC5 debug events can only occur if the address of the instruction fetch is equal to the value specified in IAC5. IAC6 debug events can only occur if the address of the instruction fetch is equal to the value specified in IAC6. 01 Address bit match. IAC5 debug events can occur only if the address of the instruction fetch, ANDed with the contents of IAC6 are equal to the contents of IAC5, also ANDed with the contents of IAC6. IAC6 debug events do not occur. IAC5US and IAC5ER settings are used. 10 Reserved 11 Reserved
10–15	—	Reserved
16–17	IAC7US	Instruction Address Compare 7 User/Supervisor Mode 00 IAC7 debug events are not affected by MSR[PR]. 01 Reserved 10 IAC7 debug events can only occur if MSR[PR] = 0 (supervisor mode). 11 IAC7 debug events can only occur if MSR[PR] = 1 (user mode).
18–19	IAC7ER	Instruction Address Compare 7 Effective/Real Mode 00 IAC7 debug events are based on effective address. 01 Unimplemented in e200 (the Power ISA embedded category real address compare), no match can occur 10 IAC7 debug events are based on effective address and can only occur if MSR[IS] = 0. 11 IAC7 debug events are based on effective address and can only occur if MSR[IS] = 1.
20–21	IAC8US	Instruction Address Compare 8 User/Supervisor Mode 00 IAC8 debug events are not affected by MSR[PR]. 01 Reserved 10 IAC8 debug events can only occur if MSR[PR] = 0 (supervisor mode). 11 IAC8 debug events can only occur if MSR[PR] = 1 (user mode).
22–23	IAC8ER	Instruction Address Compare 8 Effective/Real Mode 00 IAC8 debug events are based on effective address. 01 Unimplemented in e200 (the Power ISA embedded category real address compare), no match can occur 10 IAC8 debug events are based on effective address and can only occur if MSR[IS] = 0. 11 IAC8 debug events are based on effective address and can only occur if MSR[IS] = 1.
24–25	IAC78M	Instruction Address Compare 7/8 Mode 00 Exact address compare. IAC7 debug events can only occur if the address of the instruction fetch is equal to the value specified in IAC7. IAC8 debug events can only occur if the address of the instruction fetch is equal to the value specified in IAC8. 01 Address bit match. IAC7 debug events can occur only if the address of the instruction fetch, ANDed with the contents of IAC8 are equal to the contents of IAC7, also ANDed with the contents of IAC8. IAC8 debug events do not occur. IAC7US and IAC7ER settings are used. 10 Reserved 11 Reserved
26–31	—	Reserved

13.3.3.7 Debug Control Register 6 (DBCR6)

Debug control register 6 extends the instruction address compare matching functionality. Figure 13-10 shows DBCR6.

SPR 603

Access: Read/Write

	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
R W	IAC1XM	IAC2XM	IAC3XM	IAC4XM	IAC5XM	IAC6XM	IAC7XM	IAC8XM
Reset	All zeros[1]							

Figure 13-10. DBCR6 Register

[1] DBCR6 is reset by processor reset **p_reset_b** if DBCR0[EDM] = 0, as well as unconditionally by **m_por**. If DBCR0[EDM] = 1, DBERC0 masks off hardware-owned resources from reset by **p_reset_b** and only software-owned resources indicated by DBERC0 will be reset by **p_reset_b**.

Table 13-12 provides bit definitions for debug control register 6.

Table 13-12. DBCR6 Bit Definitions

Bit(s)	Name	Description
0–3	IAC1XM	Instruction Address Compare 1 Extended Mask Control 0000 No additional masking when DBCR1[IAC12M] = 00. 0001–1100 Exact Match Bit Mask. Number of low order bits masked in IAC1 when comparing the storage address with the value in IAC1 for exact address compare (DBCR1[IAC12M] = 00). Ranges up to 4 KB are supported. 1101–1111 Reserved IAC1XM allows for binary power of 2 address range compares for IAC1 without requiring the use of IAC2.
4–7	IAC2XM	Instruction Address Compare 2 Extended Mask Control 0000 No additional masking when DBCR1[IAC12M] = 00. 0001–1100 Exact Match Bit Mask. Number of low order bits masked in IAC2 when comparing the storage address with the value in IAC2 for exact address compare (DBCR1[IAC12M] = 00). Ranges up to 4 KB are supported. 1101–1111 Reserved IAC2XM allows for binary power of 2 address range compares for IAC2 without requiring the use of IAC1.
8–11	IAC3XM	Instruction Address Compare 3 Extended Mask Control 0000 No additional masking when DBCR1[IAC34M] = 00. 0001–1100 Exact Match Bit Mask. Number of low order bits masked in IAC3 when comparing the storage address with the value in IAC3 for exact address compare (DBCR1[IAC34M] = 00). Ranges up to 4 KB are supported. 1101–1111 Reserved IAC3XM allows for binary power of 2 address range compares for IAC1 without requiring the use of IAC2.
12–15	IAC4XM	Instruction Address Compare 4 Extended Mask Control 0000 No additional masking when DBCR1[IAC34M] = 00. 0001–1100 Exact Match Bit Mask. Number of low order bits masked in IAC4 when comparing the storage address with the value in IAC4 for exact address compare (DBCR1[IAC34M] = 00). Ranges up to 4 KB are supported. 1101–1111 Reserved IAC4XM allows for binary power of 2 address range compares for IAC4 without requiring the use of IAC3.

Table 13-12. DBCR6 Bit Definitions (continued)

Bit(s)	Name	Description
16–19	IAC5XM	Instruction Address Compare 5 Extended Mask Control 0000 No additional masking when DBCR5[IAC56M] = 00. 0001–1100 Exact Match Bit Mask. Number of low order bits masked in IAC5 when comparing the storage address with the value in IAC5 for exact address compare (DBCR5[IAC56M] = 00). Ranges up to 4 KB are supported. 1101–1111 Reserved IAC5XM allows for binary power of 2 address range compares for IAC5 without requiring the use of IAC6.
20–23	IAC6XM	Instruction Address Compare 6 Extended Mask Control 0000 No additional masking when DBCR5[IAC56M] = 00. 0001–1100 Exact Match Bit Mask. Number of low order bits masked in IAC6 when comparing the storage address with the value in IAC6 for exact address compare (DBCR5[IAC56M] = 00). Ranges up to 4 KB are supported. 1101–1111 Reserved IAC6XM allows for binary power of 2 address range compares for IAC6 without requiring the use of IAC5.
24–27	IAC7XM	Instruction Address Compare 7 Extended Mask Control 0000 No additional masking when DBCR5[IAC78M] = 00. 0001–1100 Exact Match Bit Mask. Number of low order bits masked in IAC7 when comparing the storage address with the value in IAC7 for exact address compare (DBCR5[IAC78M] = 00). Ranges up to 4 KB are supported. 1101–1111 Reserved IAC7XM allows for binary power of 2 address range compares for IAC7 without requiring the use of IAC8.
28–31	IAC8XM	Instruction Address Compare 8 Extended Mask Control 0000 No additional masking when DBCR5[IAC78M] = 00. 0001–1100 Exact Match Bit Mask. Number of low order bits masked in IAC8 when comparing the storage address with the value in IAC8 for exact address compare (DBCR5[IAC78M] = 00). Ranges up to 4 KB are supported. 1101–1111 Reserved IAC8XM allows for binary power of 2 address range compares for IAC8 without requiring the use of IAC7.

13.3.3.8 Debug Status Register (DBSR)

The debug status register (DBSR) contains status on debug events and the most recent processor reset. Hardware sets the debug status register, and software reads and clears it. Bits in the debug status register can be cleared using **mtspr** *DBSR,RS*. Clearing is done by writing to the debug status register with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The write data to the debug status register is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect. Debug status bits are set by debug events only while internal debug mode is enabled (DBCR0[IDM] = 1).

When debug interrupts are enabled (MSR[DE] = 1, DBCR0[IDM] = 1, and DBCR0[EDM] = 0, or MSR[DE] = 1, DBCR0[IDM] = 1 and DBCR0[EDM] = 1 and software is allocated resource(s) via DBERC0), a set bit in DBSR that is not MRR, VLES, or CNT1TRG causes a debug interrupt to be generated. The debug interrupt handler is responsible for clearing DBSR bits prior to returning to normal execution. The Power ISA VLE unit adds the DBSR[VLES] status bit to indicate debug events occurring due to a Power ISA VLE instruction. When resource sharing is enabled, (DBCR0[EDM] = 1 and DBERC0[IDM] = 1), only software-owned resources may be modified by software, and status bits associated with hardware-owned resources will not be set by hardware in DBSR.

Figure 13-11 shows the debug status register.

SPR 304 Access: Read/Write

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R W	IDE	UDE	MRR		ICMP	BRT	IRPT	TRAP	IAC1	IAC2	IAC3	IAC4-8	DAC1R	DAC1W	DAC2R	DAC2W
Reset[1]	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0

	16	17			20	21	22	23	24	25	26	27	28	29	30	31
R W	RET	—				DEVT1	DEVT2	DCNT1	DCNT2	CIRPT	CRET	VLES	DAC_OFST			CNT1TRG
Reset[1]	All zeros															

Figure 13-11. DBSR Register

[1] Reset by processor reset **p_reset_b** if DBCR0[EDM] = 0, as well as unconditionally by **m_por**. However, DBSR[MRR] is always updated by **p_reset_b**. If DBCR0[EDM] = 1, DBERC0 masks off hardware-owned resources from reset by **p_reset_b**, and **p_reset_b** only resets the software-owned resources indicated by DBERC0**.** However, **p_reset_b** always updates DBSR[MRR].

Table 13-13 provides bit definitions for the debug status register.

Table 13-13. DBSR Bit Definitions

Bit(s)	Name	Description
0	IDE	Imprecise Debug Event Set if MSR[DE] = 0 and DBCR0[IDM] = 1 and a debug event causes its respective debug status register bit to be set. It may also be set if an imprecise debug event occurs due to a DAC event on a load or store which is terminated with error or if an ICMP event occurs in conjunction with a EFPU FP round exception.
1	UDE	Unconditional Debug Event Set if an unconditional debug event occurred.
2–3	MRR	Most Recent Reset. 00 No reset occurred since these bits were last cleared by software. 01 A hard reset occurred since these bits were last cleared by software. 10 Reserved 11 Reserved
4	ICMP	Instruction Complete Debug Event Set if an instruction complete debug event occurred.
5	BRT	Branch Taken Debug Event Set if an branch taken debug event occurred.
6	IRPT	Interrupt Taken Debug Event Set if an interrupt taken debug event occurred.
7	TRAP	Trap Taken Debug Event Set if a trap taken debug event occurred.
8	IAC1	Instruction Address Compare 1 Debug Event Set if an IAC1 debug event occurred.
9	IAC2	Instruction Address Compare 2 Debug Event Set if an IAC2 debug event occurred.

Table 13-13. DBSR Bit Definitions (continued)

Bit(s)	Name	Description
10	IAC3	Instruction Address Compare 3 Debug Event Set if an IAC3 debug event occurred.
11	IAC4–8	Instruction Address Compare 4-8 Debug Event Set if an IAC4, IAC5, IAC6, IAC7, or IAC8 debug event occurred.
12	DAC1R	Data Address Compare 1 Read Debug Event Set if a read-type DAC1 debug event occurred while DBCR0[DAC1] = 0b10 or DBCR0[DAC1] = 0b11
13	DAC1W	Data Address Compare 1 Write Debug Event Set if a write-type DAC1 debug event occurred while DBCR0[DAC1] = 0b01 or DBCR0[DAC1] = 0b11
14	DAC2R	Data Address Compare 2 Read Debug Event Set if a read-type DAC2 debug event occurred while DBCR0[DAC2] = 0b10 or DBCR0[DAC2] = 0b11
15	DAC2W	Data Address Compare 2 Write Debug Event Set if a write-type DAC2 debug event occurred while DBCR0[DAC2] = 0b01 or DBCR0[DAC2] = 0b11
16	RET	Return Debug Event Set if a return debug event occurred.
17–20	—	Reserved
21	DEVT1	External Debug Event 1 Debug Event Set if a DEVT1 debug event occurred.
22	DEVT2	External Debug Event 2 Debug Event Set if a DEVT2 debug event occurred.
23	DCNT1	Debug Counter 1 Debug Event Set if a DCNT1 debug event occurred.
24	DCNT2	Debug Counter 2 Debug Event Set if a DCNT2 debug event occurred.
25	CIRPT	Critical Interrupt Taken Debug Event Set if a critical interrupt taken debug event occurred.
26	CRET	Critical Return Debug Event Set if a critical return debug event occurred.
27	VLES	VLE Status Set if an ICMP, BRT, TRAP, RET, CRET, IAC, or DAC debug event occurred on a Power ISA VLE Instruction. Undefined for IRPT, CIRPT, DEVT[1,2], DCNT[1,2], and UDE events
28–30	DAC_OFST	Data Address Compare Offset Indicates offset-1 of saved DSRR0 value from the address of the load or store instruction which took a DAC Debug exception, unless a simultaneous DTLB or DSI error occurs, in which case this field is set to 0b000 and DBSR[IDE] is set. Normally set to 0b000 by a non-DVC DAC. A DVC DAC may set this field to any value.
31	CNT1TRG	Counter 1 Triggered Set if debug counter 1 is triggered by a trigger event.

13.3.4 Debug External Resource Control Register (DBERC0)

The debug external resource control register (DBERC0) controls resource allocation when DBCR0[EDM] is set. DBERC0 provides a mechanism for the hardware debugger to share certain debug resources with software. Individual resources are allocated based on the settings of DBERC0 when DBCR0[EDM] = 1. DBERC0 settings are ignored when DBCR0[EDM] = 0.

Hardware-owned resources that generate debug events update EDBSR0 instead of DBSR and cause entry into debug mode if the event is not masked in EDBSRMSK0. Software-owned resources that generate debug events if DBCR0[IDM] = 1 update DBSR, causing debug interrupts to occur if MSR[DE] = 1. DBERC0 is controlled via the OnCE port hardware and is read-only to software.

The DBSR status register is always owned by software. Debug status bits in DBSR are set by software-owned debug events only while internal debug mode is enabled. When debug interrupts are enabled (MSR[DE] = 1, DBCR0[IDM] = 1, and DBCR0[EDM] = 0, or MSR[DE] = 1, DBCR0[IDM] = 1 and DBCR0[EDM] = 1 and software is allocated resource(s) via DBERC0), a set bit in DBSR by an event that is software-owned (other than MRR, DAC_OFST, CNT1TRG, or VLES) causes a debug interrupt to be generated.

Debug status bits in EDBSR0 are set by hardware-owned debug events only while external debug mode is enabled (DBCR0[EDM] = 1). When DBCR0[EDM] = 1, a set bit in EDBSR0 by an event that is hardware-owned (other than IDE, DAC_OFST, CNT1TRG, or VLES) causes entry into debug mode.

If DBCR0[EDM] = 1, DBSR status bits corresponding to hardware-owned debug events are masked from being set by hardware.

Software-owned resources may be modified by software, but only the corresponding control bits in DBCR0–6 are affected by execution of a **mtspr**. Only a portion of these registers may be affected, depending on the allocation settings in DBERC0. The debug interrupt handler is still responsible for clearing DBSR bits for software-owned resources prior to returning to normal execution. Hardware always has full access to all registers and register fields through the OnCE register access mechanism, and it is up to the debug firmware to properly implement modifications to these registers with read-modify-write operations to implement any control sharing with software. Settings in DBERC0 should be considered by the debug firmware in order to preserve software settings of control and status registers as appropriate when hardware modifications to the debug registers is performed.

Figure 13-12 shows the DBERC0 register.

SPR 569

Access: Read only

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	—	IDM	RST	UDE	ICMP	BRT	IRPT	TRAP	IAC1	IAC2	IAC3	IAC4	DAC1	—	DAC2	—
W																
Reset	Unaffected[1]															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	RET	IAC5	IAC6	IAC7	IAC8	DEVT 1	DEVT 2	DCNT 1	DCNT 2	CIRPT	CRET	BKPT	DQM	—		FT
W																
Reset	Unaffected[1]															

[1] Unaffected by **p_reset_b**; cleared by **m_por** or while in the test-logic-reset OnCE controller state

Figure 13-12. DBERC0 Register

Table 13-14 provides bit definitions for the debug external resource control register. Note that DBERC0 controls are disabled when DBCR0[EDM] = 0.

Table 13-14. DBERC0 Bit Definitions

Bit(s)	Name	Description
0	—	Reserved
1	IDM	Internal Debug Mode control 0 Internal debug mode may not be enabled by software. DBCR0[IDM] is owned exclusively by hardware. **mtspr** DBCR0–6 or DBCNT is always ignored. No resource sharing occurs, regardless of the settings of other fields in DBERC0. Hardware exclusively owns all resources. 1 Internal debug mode may be enabled by software. DBCR0[IDM] is owned by software. DBCR0[IDM] is software readable/writable. When DBERC0[IDM] = 1, software writes to hardware-owned bits in DBCR0–6 and DBCNT via **mtspr** are ignored.
2	RST	Reset Field Control 0 DBCR0[RST] owned exclusively by hardware debug. No **mtspr** access by software to DBCR0[RST] field. 1 DBCR0[RST] accessible by software debug. DBCR0[RST] is software readable/writable.
3	UDE	Unconditional Debug Event 0 Event owned by hardware debug. 1 Event owned by software debug.
4	ICMP	Instruction Complete Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to DBCR0[ICMP]. 1 Event owned by software debug. DBCR0[ICMP] is software readable/writable.
5	BRT	Branch Taken Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to DBCR0[BRT]. 1 Event owned by software debug. DBCR0[BRT] is software readable/writable.
6	IRPT	Interrupt Taken Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to DBCR0[IRPT]. 1 Event owned by software debug. DBCR0[IRPT] is software readable/writable.
7	TRAP	Trap Taken Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to DBCR0[TRAP]. 1 Event owned by software debug. DBCR0[TRAP] is software readable/writable.
8	IAC1	Instruction Address Compare 1 Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to IAC1 control and status fields. 1 Event owned by software debug. IAC1 control fields are software readable/writable.
9	IAC2	Instruction Address Compare 2 Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to IAC2 control and status fields. 1 Event owned by software debug. IAC2 control fields are software readable/writable.
10	IAC3	Instruction Address Compare 3 Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to IAC3 control and status fields. 1 Event owned by software debug. IAC3 control fields are software readable/writable.
11	IAC4	Instruction Address Compare 4 Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to IAC4 control and status fields. 1 Event owned by software debug. IAC4 control fields are software readable/writable.

Table 13-14. DBERC0 Bit Definitions (continued)

Bit(s)	Name	Description
12	DAC1	Data Address Compare 1 Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to DAC1 control and status fields. 1 Event owned by software debug. DAC1 control fields are software readable/writable.
13	—	Reserved
14	DAC2	Data Address Compare 2 Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to DAC2 control and status fields. 1 Event owned by software debug. DAC2 control fields are software readable/writable.
15	—	Reserved
16	RET	Return Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to DBCR0[RET]. 1 Event owned by software debug. DBCR0[RET] is software readable/writable.
17	IAC5	Instruction Address Compare 5 Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to IAC5 control and status fields. 1 Event owned by software debug. IAC5 control fields are software readable/writable.
18	IAC6	Instruction Address Compare 6 Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to IAC6 control and status fields. 1 Event owned by software debug. IAC6 control fields are software readable/writable.
19	IAC7	Instruction Address Compare 7 Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to IAC7 control and status fields. 1 Event owned by software debug. IAC7 control fields are software readable/writable.
20	IAC8	Instruction Address Compare 8 Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to IAC8 control and status fields. 1 Event owned by software debug. IAC8 control are software readable/writable.
21	DEVT1	External Debug Event Input 1 Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to DBCR0[DEVT1]. 1 Event owned by software debug. DBCR0[DEVT1] is software readable/writable.
22	DEVT2	External Debug Event Input 2 Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to DBCR0[DEVT2]. 1 Event owned by software debug. DBCR0[DEVT2] is software readable/writable.
23	DCNT1	Debug Counter 1 Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to Counter1 control and status fields. 1 Event owned by software debug. Counter1 control and status fields are software readable/writable.
24	DCNT2	Debug Counter 2 Debug Event 0 Event owned by hardware debug.No **mtspr** access by software to Counter2 control and status fields. 1 Event owned by software debug. Counter2 control and status fields are software readable/writable.
25	CIRPT	Critical Interrupt Taken Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to DBCR0[CIRPT]. 1 Event owned by software debug. DBCR0[CIRPT] is software readable/writable.
26	CRET	Critical Return Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to DBCR0[CRET]. 1 Event owned by software debug. DBCR0[CRET] is software readable/writable.

Table 13-14. DBERC0 Bit Definitions (continued)

Bit(s)	Name	Description
27	BKPT	Breakpoint Instruction Debug Control 0 Breakpoint owned by hardware debug. Execution of a bkpt instruction (all zeros opcode) results in entry into debug mode. 1 Breakpoint owned by software debug. Execution of a bkpt instruction (all zeros opcode) results in illegal instruction exception.
28	DQM	Data Acquisition Messaging Registers 0 DEVENT[DQTAG] and DDAM register are exclusively owned by hardware debug. No **mtspr** access by software to DEVENT[DQTAG] or DDAM register. Attempted access by software is ignored. 1 DEVENT[DQTAG] and DDAM register are owned by software. Software has read/write access to DEVENT[DQTAG] and DDAM register.
29–30	—	Reserved
31	FT	Freeze Timer Debug Control 0 DBCR0[FT] owned by hardware debug. No access by software. 1 DBCR0[FT] owned by software debug. DBCR0[FT] is software readable/writable.

Table 13-15 shows which resources are controlled by DBERC0 settings.

Table 13-15. DBERC0 Resource Control

DBCR0[EDM]	DBERC0[IDM]	DBERC0[RST]	DBERC0[UDE]	DBERC0[ICMP]	DBERC0[BRT]	DBERC0[IRPT]	DBERC0[TRAP]	DBERC0[IAC1]	DBERC0[IAC2]	DBERC0[IAC3]	DBERC0[IAC4]	DBERC0[IAC5]	DBERC0[IAC6]	DBERC0[IAC7]	DBERC0[IAC8]	DBERC0[DAC1]	DBERC0[DAC2]	DBERC0[RET]	DBERC0[DEVT1]	DBERC0[DEVT2]	DBERC0[DCNT1]	DBERC0[DCNT2]	DBERC0[CIRPT]	DBERC0[CRET]	DBERC0[BKPT]	DBERC0[DQM]	DBERC0[FT]	Software Accessible via mtspr, affected by p_reset_b
0	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	All debug registers
1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	DBCR0[IDM]
1	1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	DBCR0[RST]
1	1	—	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	DBCR0[UDE]
1	1	—	—	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	DBCR0[ICMP]
1	1	—	—	—	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	DBCR0[BRT]
1	1	—	—	—	—	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	DBCR0[IRPT]
1	1	—	—	—	—	—	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	DBCR0[TRAP]
1	1	—	—	—	—	—	—	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	IAC1 DBCR0[IAC1] DBCR1[IAC1US, IAC1ER] DBCR6[IAC1XM]
1	1	—	—	—	—	—	—	—	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	IAC2 DBCR0[IAC2] DBCR1[IAC2US, IAC2ER] DBCR6[IAC2XM]

Table 13-15. DBERC0 Resource Control (continued)

DBCR0[EDM]	DBERC0[IDM]	DBERC0[RST]	DBERC0[UDE]	DBERC0[ICMP]	DBERC0[BRT]	DBERC0[IRPT]	DBERC0[TRAP]	DBERC0[IAC1]	DBERC0[IAC2]	DBERC0[IAC3]	DBERC0[IAC4]	DBERC0[IAC5]	DBERC0[IAC6]	DBERC0[IAC7]	DBERC0[IAC8]	DBERC0[DAC1]	DBERC0[DAC2]	DBERC0[RET]	DBERC0[DEVT1]	DBERC0[DEVT2]	DBERC0[DCNT1]	DBERC0[DCNT2]	DBERC0[CIRPT]	DBERC0[CRET]	DBERC0[BKPT]	DBERC0[DQM]	DBERC0[FT]	Software Accessible via mtspr, affected by p_reset_b
1	1	—	—	—	—	—	—	1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	DBCR1[IAC12M]
1	1	—	—	—	—	—	—	—	—	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	IAC3 DBCR0[IAC3] DBCR1[IAC3US, IAC3ER] DBCR6[IAC3XM]
1	1	—	—	—	—	—	—	—	—	—	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	IAC4 DBCR0[IAC4] DBCR1[IAC4US, IAC4ER] DBCR6[IAC4XM]
1	1	—	—	—	—	—	—	—	—	1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	DBCR1[IAC34M]
1	1	—	—	—	—	—	—	—	—	—	—	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	IAC5 DBCR0[IAC5] DBCR5[IAC5US, IAC5ER] DBCR6[IAC5XM]
1	1	—	—	—	—	—	—	—	—	—	—	—	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	IAC6 DBCR0[IAC6] DBCR5[IAC6US, IAC6ER] DBCR6[IAC6XM]
1	1	—	—	—	—	—	—	—	—	—	—	1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	DBCR5[IAC56M]
1	1	—	—	—	—	—	—	—	—	—	—	—	—	1	—	—	—	—	—	—	—	—	—	—	—	—	—	IAC7 DBCR0[IAC7] DBCR5[IAC7US IAC7ER] DBCR6[IAC7XM]
1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	1	—	—	—	—	—	—	—	—	—	—	—	—	IAC8 DBCR0[IAC8] DBCR5[IAC8US IAC8ER] DBCR6[IAC8XM]
1	1	—	—	—	—	—	—	—	—	—	—	—	—	1	1	—	—	—	—	—	—	—	—	—	—	—	—	DBCR5[IAC78M]
1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	1	—	—	—	—	—	—	—	—	—	—	—	DAC1, DVC1 DBCR0[DAC1] DBCR2[DAC1US DAC1ER] DBCR2[DVC1M DVC1BE] DBCR[4DVC1C DAC1XM]

Table 13-15. DBERC0 Resource Control (continued)

DBCR0[EDM]	DBERC0[IDM]	DBERC0[RST]	DBERC0[UDE]	DBERC0[ICMP]	DBERC0[BRT]	DBERC0[IRPT]	DBERC0[TRAP]	DBERC0[IAC1]	DBERC0[IAC2]	DBERC0[IAC3]	DBERC0[IAC4]	DBERC0[IAC5]	DBERC0[IAC6]	DBERC0[IAC7]	DBERC0[IAC8]	DBERC0[DAC1]	DBERC0[DAC2]	DBERC0[RET]	DBERC0[DEVT1]	DBERC0[DEVT2]	DBERC0[DCNT1]	DBERC0[DCNT2]	DBERC0[CIRPT]	DBERC0[CRET]	DBERC0[BKPT]	DBERC0[DQM]	DBERC0[FT]	Software Accessible via mtspr, affected by p_reset_b
1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	1	—	—	—	—	—	—	—	—	—	—	DAC2, DVC2 DBCR0[DAC2] DBCR2[DAC2US DAC2ER] DBCR2[DVC2M DVC2BE] DBCR4[DVC2C DAC2XM]
1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	1	1	—	—	—	—	—	—	—	—	—	—	DBCR2[DAC12M]
1	1	—	—	—	—	—	—	1	—	—	—	—	—	—	—	1	—	—	—	—	—	—	—	—	—	—	—	DBCR2[DAC1LNK]
1	1	—	—	—	—	—	—	—	—	1	—	—	—	—	—	—	1	—	—	—	—	—	—	—	—	—	—	DBCR2[DAC2LNK]
1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	1	—	—	—	—	—	—	—	—	—	DBCR0[RET]
1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	1	—	—	—	—	—	—	—	—	DBCR0[DEVT1]
1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	1	—	—	—	—	—	—	—	DBCR0[DEVT2]
1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	1	—	—	—	—	—	—	DBCR0D[CNT1] DBCR3[DEVT1C1, DEVT2C1, ICMPC1, IAC1C1, IAC2C1, IAC3C1, IAC4C1, DAC1RC1, DAC1WC1, DAC2RC1, DAC2WC1, IRPTC1, RETC1, DEVT1T1, DEVT2T1, IAC1T1, IAC3T1, DAC1RT1, DAC1WT1, CNT2T1][1] DBCNT[CNT1]
1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	1	—	—	—	—	—	DBCR0[DCNT2] DBCR3[DEVT1C2, DEVT2C2, ICMPC2, IAC1C2, IAC2C2, IAC3C2, IAC4C2, DAC1RC2, DAC1WC2, DAC2RC2, DAC2WC2][2] DBCNT[CNT2]
1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	1	1	—	—	—	—	—	DBCR3[CONFIG]
1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	1	—	—	—	—	DBCR0[CIRPT]
1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	1	—	—	—	DBCR0[CRET]
1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	1	—	—	No software-visible resource

Table 13-15. DBERC0 Resource Control (continued)

DBCR0[EDM]	DBERC0[IDM]	DBERC0[RST]	DBERC0[UDE]	DBERC0[ICMP]	DBERC0[BRT]	DBERC0[IRPT]	DBERC0[TRAP]	DBERC0[IAC1]	DBERC0[IAC2]	DBERC0[IAC3]	DBERC0[IAC4]	DBERC0[IAC5]	DBERC0[IAC6]	DBERC0[IAC7]	DBERC0[IAC8]	DBERC0[DAC1]	DBERC0[DAC2]	DBERC0[RET]	DBERC0[DEVT1]	DBERC0[DEVT2]	DBERC0[DCNT1]	DBERC0[DCNT2]	DBERC0[CIRPT]	DBERC0[CRET]	DBERC0[BKPT]	DBERC0[DQM]	DBERC0[FT]	Software Accessible via mtspr, affected by p_reset_b
1	—[3]	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	1	—	DEVENT[DQTAG] DDAM
1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	1	DBCR0[FT]

[1] Note that software is given write access to all counter 1 control events and triggers regardless of whether software owns these events. It is considered a programming error to enable counter or trigger events in DBCR3 which are not owned by software, and operational results of the counter(s) are undefined if programmed.

[2] Note that software is given write access to all counter 2 control events regardless of whether software owns these events. It is considered a programming error to enable counter events in DBCR3 which are not owned by software, and operational results of the counter(s) are undefined if programmed.

[3] Note: IDM not required to be set to enable software access.

DBERC0 also controls which bits or fields in DBCR0–6 are reset by assertion of **p_reset_b** when DBCR0[EDM] = 1. Only software-owned bits or fields as shown in Table 13-15 are affected in this case, except that DBCR0[RST] and DBSR[MRR] are updated by assertion of **p_reset_b** regardless of the value of DBCR0[EDM] or DBERC0.

13.3.5 Debug Event Select Register (DEVENT)

The debug event select register allows instrumented software to internally generate signals when a **mtspr** instruction is executed and this register is accessed. The values written to this register determine which of the **p_devnt_out[0:7]** processor output signals are asserted upon access. Writing a 1 to any of these bit positions causes a one clock pulse to be generated on the corresponding output. For **p_devnt_out[0:3]**, a corresponding **jd_watchpt[x]** output is asserted as well to indicate a watchpoint has occurred. These signals may be used for internal core debug resources as well as for SoC-level cross-triggering. See the reference manual for your specific device for information about SoC use cases.

DEVENT[DEVNT] is undefined on a read; it may or may not remain set to the last value written. Because it is unconditionally shared by hardware debug and software, software should not rely on any value remaining.

The upper 8-bits of the DEVENT register also provide the DQTAG used to identify channels within Data Acquisition Messages. See Section 14.13.1, “Data Acquisition ID Tag Field,” for more detail on the DQTAG.

Figure 13-13 shows the DEVENT register.

SPR 975 Access: Special

	0–7	8–23	24–31
R W	DQTAG	—	DEVNT
Reset	All zeros[1]		

[1] Reset by processor reset **p_reset_b** if DBCR0[EDM] = 0, as well as unconditionally by **m_por**. If DBCR0[EDM] = 1, DBERC0 masks off hardware-owned resources from reset by **p_reset_b**, and **p_reset_b** only resets software-owned resources indicated by DBERC0**.** Note that DEVNT field is shared by hardware and software but is always reset by **p_reset_b**.

Figure 13-13. DEVENT Register

Table 13-16 provides bit definitions for the debug event register.

Table 13-16. DEVENT Bit Definitions

Bit(s)	Name	Description
0–7	DQTAG	Data Acquisition Message IDTAG channel identifier (supplied to Nexus 3)
8–23	—	Reserved, should be cleared.
24–31	DEVNT	Debug Event Signals 00000000—No signal is asserted xxxxxxx1—**p_devnt_out[0]** and **jd_watchpt[12]** are asserted for one clock xxxxxx1x—**p_devnt_out[1]** and **jd_watchpt[13]** are asserted for one clock xxxxx1xx—**p_devnt_out[2]** and **jd_watchpt[20]** are asserted for one clock xxxx1xxx—**p_devnt_out[3]** and **jd_watchpt[21]** are asserted for one clock xxx1xxxx—**p_devnt_out[4]** is asserted for one clock xx1xxxxx—**p_devnt_out[5]** is asserted for one clock x1xxxxxx—**p_devnt_out[6]** is asserted for one clock 1xxxxxxx—**p_devnt_out[7]** is asserted for one clock

13.3.6 Debug Data Acquisition Message Register (DDAM)

The debug data acquisition message register allows instrumented software to generate real-time data acquisition messages (as defined by Nexus 3) via a **mtspr** instruction to this register. See Section 14.13, “Data Acquisition Messaging,” for details.

Figure 13-14 shows the DDAM register.

SPR 576 Access: Special

	0–31
R W	DDAM
Reset	All zeros[1]

[1] Reset by processor reset **p_reset_b** if DBCR0[EDM] = 0, as well as unconditionally by **m_por**. If DBCR0[EDM] = 1, DBERC0 masks off hardware-owned resources from reset by **p_reset_b,** and **p_reset_b** only resets software-owned resources indicated by DBERC0.

Figure 13-14. DDAM Register

Table 13-17 provides bit definitions for the debug data acquisition message register.

Table 13-17. DDAM Bit Definitions

Bit(s)	Name	Description
0–31	DDAM	Value to be transmitted in a data acquisition message (DQM) (supplied to Nexus 3 with strobe)

13.4 External Debug Support

External debug support is supplied through the OnCE controller serial interface which allows access to internal CPU registers and other system state while the CPU is halted in debug mode. All debug resources including DBCR0–6, DBSR, IAC1–8, DAC1–2, DVC1–2, and DBCNT are accessible through the serial OnCE interface in external debug mode. Setting EDBCR0[EDM]/DBCR0[EDM] to 1 through the OnCE interface enables external debug mode, and unless otherwise permitted by the settings in DBERC0, disables software updates to the debug control registers. When [E]DBCR0[EDM] is set, debug events enabled to set respective status bits also cause the CPU to enter debug mode if the event is not masked in EDBSRMSK0, as opposed to generating debug interrupts, unless the specific events are allocated to software via the settings in DBERC0. In debug mode, the CPU is halted at a recoverable boundary, and an external debug control module may control CPU operation through the on-chip emulation logic (OnCE). [EDM]

Note that the descriptions of events in the subsections of Section 13.2, “Software Debug Events and Exceptions,” refer to setting DBSR status bits. However, when resources are owned by hardware, the events for those resources set the respective status bits in EDBSR0 instead of DBSR.

NOTE

On the initial setting of EDBCR0[EDM]/DBCR0[EDM], other bits in DBCR0 remain unchanged. After EDBCR0[EDM]/DBCR0[EDM] has been set, all debug register resources may be subsequently controlled through the OnCE interface. The CPU should be placed into debug mode via the OCR[DR] control bit prior to writing EDM to 1. This gives the debugger the opportunity to cleanly write to the DBCRx registers and the DBSR to clear out any residual state/control information that can cause unintended operation.

It is intended for the CPU to remain in external debug mode (DBCR0[EDM] = 1) in order to single step or perform other debug mode entry/reentry via the OCR[DR] by performing go+noexit commands or by assertion of the **jd_de_b** signal.

DBCR0[EDM] operation is blocked if the OnCE operation is disabled (**jd_en_once** negated), regardless of whether it is set or cleared. This means that if DBCR0[EDM] was previously set, and then **jd_en_once** was negated (this should not occur). Entry into debug mode is blocked, all events are blocked, and watchpoints are blocked.

Due to clock domain design, the CPU clock (**m_clk**) must be active to perform writes to debug registers other than the OnCE command register (OCMD), the OnCE control register (OCR), external debug control register 0 (EDBCR0), external debug status register 0 (EDBSR0), external debug status register mask 0 (EDBSRMSK0), or DBCR0[EDM]. Register read data is synchronized back to the **j_tclk** clock domain. The OnCE control register allows signaling the system level clock controller that the CPU clock should be activated if not already active.

Updates to the DBCRx, DBSR, and DBCNT registers via the OnCE interface should be performed with the CPU in debug mode to guarantee proper operation. Due to the various points in the CPU pipeline where control is sampled and event handshaking is performed, modifications to these registers while the CPU is running may result in early or late entry into debug mode and may have incorrect status posted in the DBSR register.

If resource sharing is enabled via DBERC0, updates to the DBERC0, DBCRx, DBCNT, and DBSR registers must be performed with the CPU in debug mode because otherwise, simultaneous updates of register portions can be attempted. These updates are not guaranteed to occur properly. The results of such an attempt are undefined.

13.4.1 External Debug Registers

The external debug registers are used for controlling several debug aspects of the core and reporting status while the e200z7 is in external debug mode.

13.4.1.1 External Debug Control Register 0 (EDBCR0)

EDBCR0 is a control register accessible to an external debugger through the OnCE/JTAG port. An external development tool can write to this register in order to enable external debug mode or to enable Debugger Notify Halt instructions (**dnh**, **se_dnh**).

EDBCR0 is not accessible by software, However, the state of EDBCR0[EDM] is reflected as a read-only bit in DBCR0[EDM] to software. There is only one physical EDM bit implemented. It is reflected in both the DBCR0 and EDBCR0 registers and may be written and read using either register by the hardware debugger. For future compatibility, EDBCR0 updates are preferred.

Figure 13-15 shows EDBCR0.

Access: Special

	0	1	2 3	4 … 31
R	EDM	DNH_EN	DFT	—
W				
Reset	All zeros[1]			

[1] EDBCR0 is affected (reset) by **j_trst_b** or **m_por** assertion and remains reset while in the Test_Logic_Reset state. It is not affected by **p_reset_b**.

Figure 13-15. EDBCR0 Register

Table 13-18 provides bit definitions for external debug control register 0.

Table 13-18. EDBCR0 Bit Definitions

Bit(s)	Name	Description
0	EDM	External Debug Mode. This bit is also reflected in DBCR0. 0 External debug mode disabled. Internal debug events not mapped into external debug events. 1 External debug mode enabled. Hardware-owned events will not cause the CPU to vector to interrupt code. Software is not permitted to write to debug registers {DBCRx, DBCNT, IAC1–8, DAC1–2, DVC1–2} unless permitted by settings in DBERC0. When external debug mode is enabled, hardware-owned resources in debug registers are not affected by processor reset **p_reset_b**. This allows the debugger to set up hardware debug events which remain active across a processor reset.
1	DNH_EN	**dnh** Instruction Enable 0 Execution of **dnh** and **se_dnh** instructions cause illegal instruction exceptions to occur. 1 Execution of **dnh** and **se_dnh** instructions cause entry into debug mode and a debug halt occurs, regardless of the value of EDM.
2–3	DFT	Debug Freeze Timers Control 00 Timebase, watchdog timer, and decrementer are not clocked during a debug session 01 Timebase and watchdog timer are not clocked during a debug session. Decrementer is unaffected 10 Decrementer is not clocked during a debug session. Timebase and watchdog timers are unaffected 11 No timer freeze during a debug session
4–31	---	Reserved

13.4.1.2 External Debug Status Register 0 (EDBSR0)

The external debug status register 0 (EDBSR0) contains status on debug events owned by hardware. Hardware is used to set the external debug status register 0 and the debugger reads and clears it by writing to it by means of the OnCE port. A 1 is in any bit position that is to be cleared and a 0 is in all other bit positions. The write data to EDBSR0 is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect.

Figure 13-16 shows the EDBSR0 register.

Access: Read/Write

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R W	IDE	UDE	DNH	0	ICMP	BRT	IRPT	TRAP	IAC1	IAC2	IAC3	IAC4–8	DAC1R	DAC1W	DAC2R	DAC2W
Reset	All zeros															

	16	17–20	21	22	23	24	25	26	27	28–30	31
R W	RET	—	DEVT1	DEVT2	DCNT1	DCNT2	CIRPT	CRET	VLES	DAC_OFST	CNT1TRG
Reset	All zeros										

[1] Reset by **j_trst_b** or **m_por** assertion and remains reset while in the Test_Logic_Reset state or while EDBCR0[EDM] = 0.

Figure 13-16. EDBSR0 Register

Table 13-19 provides bit definitions for external debug status register 0.

Table 13-19. EDBSR0 Bit Definitions

Bit(s)	Name	Description
0	IDE	Imprecise Debug Event Set if DBCR0[EDM] = 1 and an imprecise debug event occurs for a hardware-owned DAC event due to a load or store which is terminated with error or if a hardware-owned ICMP event occurs in conjunction with a EFPU round exception. This bit will not be set for imprecise debug events which are masked via settings in EDBSRMSK0.
1	UDE	Unconditional Debug Event Set if a hardware-owned unconditional debug event occurred.
2	DNH	Debugger Notify Halt Event Set if a debugger notify halt instruction was executed and caused a debug halt.
3	—	Reserved
4	ICMP	Instruction Complete Debug Event Set if a hardware-owned Instruction complete debug event occurred.
5	BRT	Branch Taken Debug Event Set if a hardware-owned branch taken debug event occurred.
6	IRPT	Interrupt Taken Debug Event Set if a hardware-owned Interrupt taken debug event occurred.
7	TRAP	Trap Taken Debug Event Set if a hardware-owned trap taken debug event occurred.
8	IAC1	Instruction Address Compare 1 Debug Event Set if a hardware-owned IAC1 debug event occurred.
9	IAC2	Instruction Address Compare 2 Debug Event Set if a hardware-owned IAC2 debug event occurred.
10	IAC3	Instruction Address Compare 3 Debug Event Set if a hardware-owned IAC3 debug event occurred.
11	IAC4-8	Instruction Address Compare 4-8 Debug Event Set if a hardware-owned IAC4, IAC5, IAC6, IAC7, or IAC8 debug event occurred.
12	DAC1R	Data Address Compare 1 Read Debug Event Set if a hardware-owned read-type DAC1 debug event occurred while DBCR0[DAC1] = 0b10 or DBCR0[DAC1] = 0b11
13	DAC1W	Data Address Compare 1 Write Debug Event Set if a hardware-owned write-type DAC1 debug event occurred while DBCR0[DAC1] = 0b01 or DBCR0[DAC1] = 0b11
14	DAC2R	Data Address Compare 2 Read Debug Event Set if a hardware-owned read-type DAC2 debug event occurred while DBCR0[DAC2] = 0b10 or DBCR0[DAC2] = 0b11
15	DAC2W	Data Address Compare 2 Write Debug Event Set if a hardware-owned write-type DAC2 debug event occurred while DBCR0[DAC2] = 0b01 or DBCR0[DAC2] = 0b11
16	RET	Return Debug Event Set if a hardware-owned return debug event occurred
17:20	—	Reserved

Table 13-19. EDBSR0 Bit Definitions (continued)

Bit(s)	Name	Description
21	DEVT1	External Debug Event 1 Debug Event Set if a hardware-owned DEVT1 debug event occurred
22	DEVT2	External Debug Event 2 Debug Event Set if a hardware-owned DEVT2 debug event occurred
23	DCNT1	Debug Counter 1 Debug Event Set if a hardware-owned DCNT1 debug event occurred
24	DCNT2	Debug Counter 2 Debug Event Set if a hardware-owned DCNT2 debug event occurred
25	CIRPT	Critical Interrupt Taken Debug Event Set if a hardware-owned critical interrupt taken debug event occurred.
26	CRET	Critical Return Debug Event Set if a hardware-owned critical return debug event occurred
27	VLES	VLE Status Set if a hardware-owned ICMP, BRT, TRAP, RET, CRET, IAC, or DAC debug event occurred on a PowerPC VLE Instruction. Also set for execution of an e_dnh or se_dnh instruction when enabled by EDBCR0[DNH_EN]. Undefined for IRPT, CIRPT, DEVT[1,2], DCNT[1,2], and UDE events
28:30	DAC_OFST	Data Address Compare Offset Indicates offset-1 of saved DSRR0 value from the address of the load or store instruction which took a hardware-owned DAC Debug exception, unless a simultaneous DTLB or DSI error occurs, in which case this field is set to 0b000 and EDBSR0[IDE] is set. Normally set to 0b000 by a non-DVC DAC. A DVC DAC may set this field to any value.
31	CNT1TRG	Counter 1 Triggered Set if hardware-owned debug counter 1 is triggered by a trigger event.

13.4.1.3 External Debug Status Register Mask 0 (EDBSRMSK0)

The external debug status register mask 0 (EDBSRMSK0) is used to mask debug events set in EDBSR0 from causing entry into debug halted mode. A 1 stored in any mask bit prevents debug mode entry caused by the corresponding bit being set in EDBSR0. The mask has no effect on DBSR actions or on the setting of EDBSR0 status bits by hardware-owned events, except that the IDE bit will not be set by imprecise hardware-owned debug events which are masked. EDBSRMSK0 may be used to allow debug events owned by hardware to be configured for watchpoint generation purposes without causing debug mode entry when the watchpoint occurs. EDBSRMSK0 is read and written via OnCE access by the debugger. No software access is provided.

NOTE

Not all implementations of the e200z760n3 use this register. Consult your processor-specific device manual for whether it is used.

Figure 13-17 shows the EDBSRMSK0 register.

Access: Read/Write

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R W	—	UDE	DNH	—	ICMP	BRT	IRPT	TRAP	IAC1	IAC2	IAC3	IAC4–8	DAC1R	DAC1W	DAC2R	DAC2W
Reset	All zeros[1]															

	16	17–20	21	22	23	24	25	26	27–31
R W	RET	—	DEVT1	DEVT2	DCNT1	DCNT2	CIRPT	CRET	—
Reset	All zeros[1]								

[1] Reset by **j_trst_b** or **m_por** assertion and remains reset while in the Test_Logic_Reset state or while EDBCR0[EDM] = 0.

Figure 13-17. EDBSRMSK0 Register

Table 13-20 provides bit definitions for external debug status register mask 0.

Table 13-20. EDBSRMSK0 Bit Definitions

Bit(s)	Name	Description
0	—	Reserved
1	UDE	Unconditional Debug Event Set to mask debug mode entry by EDBSR0[UDE]
2	DNH	Debugger Notify Halt Event Set to mask debug mode entry by EDBSR0[DNH]
3	—	Reserved
4	ICMP	Instruction Complete Debug Event Set to mask debug mode entry by EDBSR0[ICMP]
5	BRT	Branch Taken Debug Event Set to mask debug mode entry by EDBSR0[BRT]
6	IRPT	Interrupt Taken Debug Event Set to mask debug mode entry by EDBSR0[IRPT]

Table 13-20. EDBSRMSK0 Bit Definitions (continued)

Bit(s)	Name	Description
7	TRAP	Trap Taken Debug Event Set to mask debug mode entry by EDBSR0[TRAP]
8	IAC1	Instruction Address Compare 1 Debug Event Set to mask debug mode entry by EDBSR0[IAC1]
9	IAC2	Instruction Address Compare 2 Debug Event Set to mask debug mode entry by EDBSR0[IAC2]
10	IAC3	Instruction Address Compare 3 Debug Event Set to mask debug mode entry by EDBSR0[IAC3]
11	IAC4–8	Instruction Address Compare 4-8 Debug Event Set to mask debug mode entry by EDBSR0[IAC4–8]
12	DAC1R	Data Address Compare 1 Read Debug Event Set to mask debug mode entry by EDBSR0[DAC1R]
13	DAC1W	Data Address Compare 1 Write Debug Event Set to mask debug mode entry by EDBSR0[DAC1W]
14	DAC2R	Data Address Compare 2 Read Debug Event Set to mask debug mode entry by EDBSR0[DAC2R]
15	DAC2W	Data Address Compare 2 Write Debug Event Set to mask debug mode entry by EDBSR0[DAC2W]
16	RET	Return Debug Event Set to mask debug mode entry by EDBSR0[RET]
17–20	—	Reserved
21	DEVT1	External Debug Event 1 Debug Event Set to mask debug mode entry by EDBSR0[DEVT1]
22	DEVT2	External Debug Event 2 Debug Event Set to mask debug mode entry by EDBSR0[DEVT2]
23	DCNT1	Debug Counter 1 Debug Event Set to mask debug mode entry by EDBSR0[DCNT1]
24	DCNT2	Debug Counter 2 Debug Event Set to mask debug mode entry by EDBSR0[DCNT1]
25	CIRPT	Critical Interrupt Taken Debug Event Set to mask debug mode entry by EDBSR0[CIRPT]
26	CRET	Critical Return Debug Event Set to mask debug mode entry by EDBSR0[CRET]
22–31	—	Reserved

13.4.2 OnCE Introduction

The e200z7 on-chip emulation circuitry (OnCE™/Nexus Class 1 interface) provides a means of interacting with the e200z7 core and integrated system so that a user may examine registers, memory, or on-chip peripherals facilitating hardware/software development. OnCE operation is controlled via an industry standard IEEE 1149.1 TAP controller. By using public instructions, the external hardware debugger can freeze or halt the CPU, read and write internal state, and resume normal execution. The core does not contain IEEE 1149.1 standard boundary cells on its interface, as it is a building block for further integration. It does not support the JTAG related boundary scan instruction functionality, although JTAG public instructions may be decoded and signaled to external logic.

The OnCE logic provides for Nexus Class 1 static debug capability (utilizing the same set of resources available to software while in internal debug mode), and is present in all e200-based designs. The OnCE module also provides support for directly integrating a Nexus class 2 or class 3 real-time debug unit with the e200 core for development of real-time systems where traditional static debug is insufficient. The partitioning between a OnCE module and a connected Nexus module to provide real-time debug allows for capability and cost trade-offs to be made.

The e200z7 core is designed to be a fully integratable module. The OnCE TAP controller and associated enabling logic are designed to allow concatenation with an existing JTAG controller if present in the system. Thus, the e200z7 can be easily integrated with existing JTAG designs or as a stand-alone controller.

In order to enable full OnCE operation, the **jd_enable_once** input signal must be asserted. In some system integrations, this is automatic, since the input will be tied asserted. Other integrations may require the execution of the Enable OnCE command via the TAP and appropriate entry of serial data. Exact requirements will be documented by the integrated product specification. The **jd_enable_once** input signal should not change state during a debug session, or undefined activity may occur.

The following figures show the TAP controller state model and the TAP registers implemented by the OnCE logic.

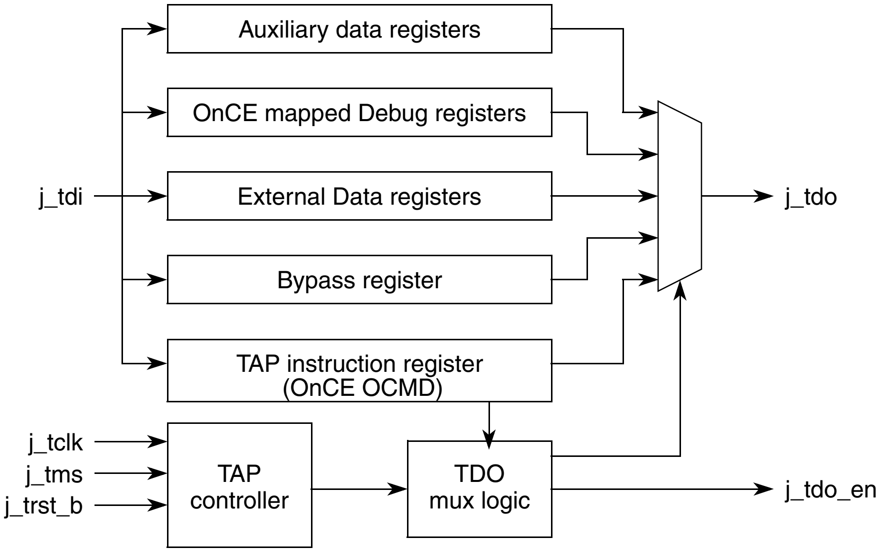
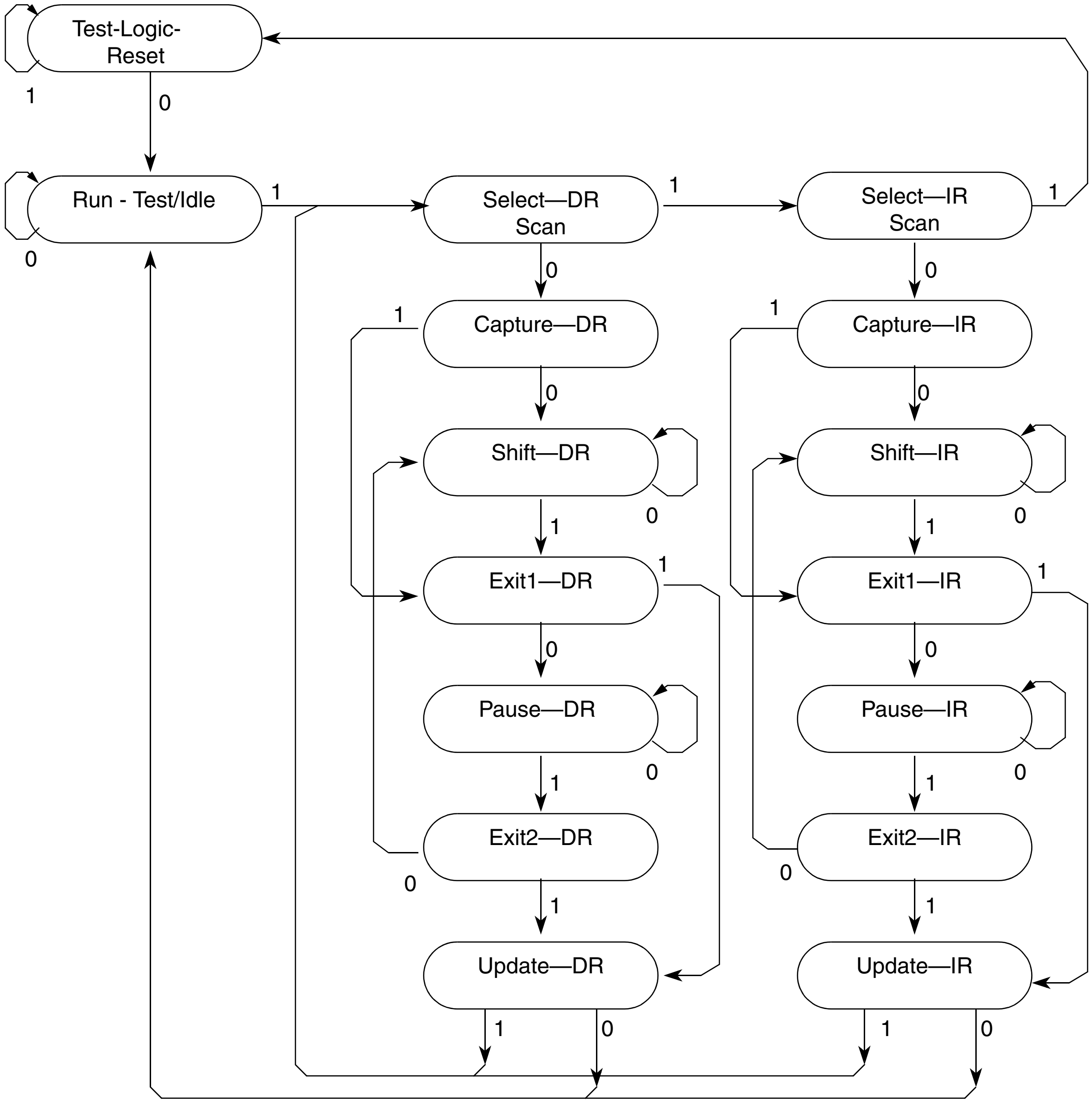


Figure 13-18. OnCE TAP Controller and Registers

The OnCE controller is implemented as a 16-state FSM, with a one-to-one correspondence to the states defined for the JTAG TAP controller.



Access to processor registers and the contents of memory locations are performed by enabling external debug mode (setting DBCR0[EDM]), placing the processor into debug mode, followed by scanning instructions and data into and out of the CPU scan chain (CPUSCR). Execution of scanned instructions by the CPU is used as the method to access required data. Memory locations may be read by scanning a load instruction into the CPU core, which references the desired memory location, executes the load instruction, and scans out the result of the load. Other resources are accessed in a similar manner.

The initial entry by the CPU into the debug state (or mode) from normal, waiting, stopped, or halted states (all indicated via the OnCE status register (OSR), Section 13.4.6.1, "e200 OnCE Status Register)" by assertion of one or more debug requests, begins a debug session. The **jd_debug_b** output signal indicates that a debug session is in progress, and the OSR indicates the CPU is in the debug state. Instructions may the be single-stepped by scanning new values into the CPUSCR, and performing a OnCE go + noexit command (See Section 13.4.6.2, “e200 OnCE Command Register (OCMD)”). The CPU then temporarily exits the debug state, but not the debug session, to execute the instruction. It then returns to the debug state (again indicated via the OnCE Status Register (OSR)). The debug session remains in force until the final OnCE go + exit command is executed, at which time the CPU returns to its previous state (unless a new debug request is pending).

Note that a scan into the CPUSCR is required prior to executing each go + exit or go + noexit OnCE command.

13.4.3 JTAG/OnCE Pins

The JTAG/OnCE pin interface is used to transfer OnCE instructions and data to the OnCE control block. Depending on the particular resource being accessed, the CPU may need to be placed in debug mode. For resources outside of the CPU block and contained in the OnCE block, the processor is not disturbed and may continue execution. If a processor resource is required, an internal debug request (**dbg_dbgrq**) may be asserted to the CPU by the OnCE controller and cause the CPU to finish the current instruction being executed, save the instruction pipeline information, enter Debug Mode, and wait for further commands. Asserting **dbg_dbgrq** causes the chip to exit the low power mode enabled by the setting of MSR[WE], as well as temporarily exiting the waiting, stopped or halted power management states.

Table 13-21 details the primary JTAG/OnCE interface signals.

Table 13-21. JTAG/OnCE Primary Interface Signals

Signal Name	Type	Description
j_trst_b	I	JTAG test reset
j_tclk	I	JTAG test clock
j_tms	I	JTAG test mode select
j_tdi	I	JTAG test data input
j_tdo	O	Test data out to master controller or pad
j_tdo_en[1]	O	Enables TDO output buffer

[1] j_tdo_en is asserted when the TAP controller is in the shift_DR or shift_IR state.

A full description of JTAG pins is provided in Section 11.2.25, “JTAG Support Signals.”

13.4.4 OnCE Internal Interface Signals

The following paragraphs describe the OnCE interface signals to other internal blocks associated with the OnCE controller.

13.4.4.1 CPU Debug Request (dbg_dbgrq)

The **dbg_dbgrq** signal is asserted by the OnCE control logic to request the CPU to enter the debug state. It may be asserted for a number of different conditions and causes the CPU to finish the current instruction being executed, save the instruction pipeline information, enter the debug mode, and wait for further commands.

13.4.4.2 CPU Debug Acknowledge (cpu_dbgack)

The **cpu_dbgack** signal is asserted by the CPU upon entering the debug state. This signal is used as part of the handshake mechanism between the OnCE control logic and the rest of the CPU. The CPU core may enter debug mode either through a software or hardware event.

13.4.4.3 CPU Address, Attributes

The CPU address and attribute information are used by a Nexus class 2-4 debug unit with information for real-time address trace information.

13.4.4.4 CPU Data

The CPU data buses are used to supply a Nexus class 2-4 debug unit with information for real-time data trace capability.

13.4.5 OnCE Interface Signals

The following paragraphs describe additional OnCE interface signals to other external blocks, such as a Nexus controller and external blocks that may need information pertaining to debug operation.

13.4.5.1 OnCE Enable (jd_en_once)

The OnCE enable signal **jd_en_once** is used to enable the OnCE controller to allow certain instructions and operations to be executed. Assertion of this signal enables the full OnCE command set, as well as operation of control signals and OnCE control register functions. When this signal is disabled, only the Bypass, ID and Enable_OnCE commands are executed by the OnCE unit, and all other commands default to a bypass command. The OnCE status register (OSR) is not visible when OnCE operation is disabled. In addition, OnCE control register (OCR) functions are disabled, as is the operation of the **jd_de_b** input. Secure systems may choose to leave the **jd_en_once** signal negated until a security check has been performed. Other systems should tie this signal asserted to enable full OnCE operation. The **j_en_once_regsel** output signal is provided to assist external logic performing security checks. Refer to Section 11.2.25.8, "Enable Once Register Select (j_en_once_regsel)," for a description of the **j_en_once_regsel** output signal.

The **jd_en_once** input must only change state during the Test-Logic-Reset, Run-Test/Idle, or Update_DR TAP states. A new value will take affect after one additional **j_tclk** cycle of synchronization. In addition, **jd_enable_once** input signal must not change state during a debug session, or undefined activity may occur.

13.4.5.2 OnCE Debug Request/Event (jd_de_b, jd_de_en)

If implemented at the SoC level, a system level bidirectional open drain debug event pin **DE_b** (not part of the e200 interface) provides a fast means of entering the debug mode of operation from an external command controller (when input) as well as a fast means of acknowledging the entering of the debug mode of operation to an external command controller (when output). The assertion of this pin by a command controller causes the CPU core to finish the current instruction being executed, save the instruction pipeline information, enter debug mode, and wait for commands to be entered. If **DE_b** was used to enter the debug mode, **DE_b** must be negated after the OnCE controller responds with an acknowledge and before sending the first OnCE command. The assertion of this pin by the CPU core acknowledges that it has entered the debug mode and is waiting for commands to be entered.

To support operation of this system pin, the OnCE logic supplies the **jd_de_en** output and samples the **jd_de_b** input when OnCE is enabled (**jd_en_once** asserted). Assertion of **jd_de_b** causes the OnCE logic to place the CPU into debug mode. Once debug mode has been entered, the **jd_de_en** output will be asserted for three **j_tclk** periods to signal an acknowledge. **jd_de_en** can be used to enable the open-drain pulldown of the system level **DE_b** pin.

For systems that do not implement a system level bidirectional open drain debug event pin **DE_b**, the **jd_de_en** and **jd_de_b** signals may still be used to handshake debug entry.

13.4.5.3 e200 OnCE Debug Output (jd_debug_b)

The e200 OnCE debug output **jd_debug_b** is used to indicate to on-chip resources that a debug session is in progress. Peripherals and other units may use this signal to modify normal operation for the duration of a debug session, which may involve the CPU executing a sequence of instructions solely for the purpose of visibility/system control that are not part of the normal instruction stream the CPU would have executed had it not been placed in debug mode. This signal is asserted the first time the CPU enters the debug state and remains asserted until the CPU is released by a write to the e200 OnCE command register with the GO and EX bits set, and a register specified as either “No Register Selected” or the CPUSCR. This signal remains asserted even though the CPU may enter and exit the debug state for each instruction executed under control of the e200 OnCE controller. See Section 13.4.6.2, “e200 OnCE Command Register (OCMD),” for more information on the function of the GO and EX bits. This signal is not normally used by the CPU.

13.4.5.4 e200 CPU Clock On Input (jd_mclk_on)

The e200 CPU Clock On input **jd_mclk_on** is used to indicate that the CPU’s **m_clk** input is active. This input signal is expected to be driven by system logic external to the e200 core, is synchronized to the **j_tclk** (scan clock) clock domain, and is presented as a status flag on the **j_tdo** output during the Shift_IR state. External firmware may use this signal to ensure proper scan sequences occur to access debug resources in the **m_clk** clock domain.

13.4.5.5 Watchpoint Events (jd_watchpt[0:29])

The **jd_watchpt[0:29]** signals may be asserted by the e200 OnCE control logic to signal that a watchpoint condition has occurred. Watchpoints do not cause the CPU to be affected. They are provided to allow external visibility only. Watchpoint events are conditioned by the settings in the DBCR0, DBCR1, and DBCR2 registers, as well as by the DEVENT register, and the Performance Monitor control register settings.

13.4.6 e200 OnCE Controller and Serial Interface

The OnCE controller contains the OnCE command register, the OnCE decoder, and the status/control register. Figure 13-19 is a block diagram of the e200 OnCE controller. In operation, the OnCE command register acts as the IR for the TAP controller, and all other OnCE resources are treated as data registers (DR) by the TAP controller. The command register is loaded by serially shifting in commands during the TAP controller Shift-IR state and is loaded during the Update-IR state. The command register selects a resource to be accessed as a data register (DR) during the TAP controller Capture-DR, Shift-DR, and Update-DR states.

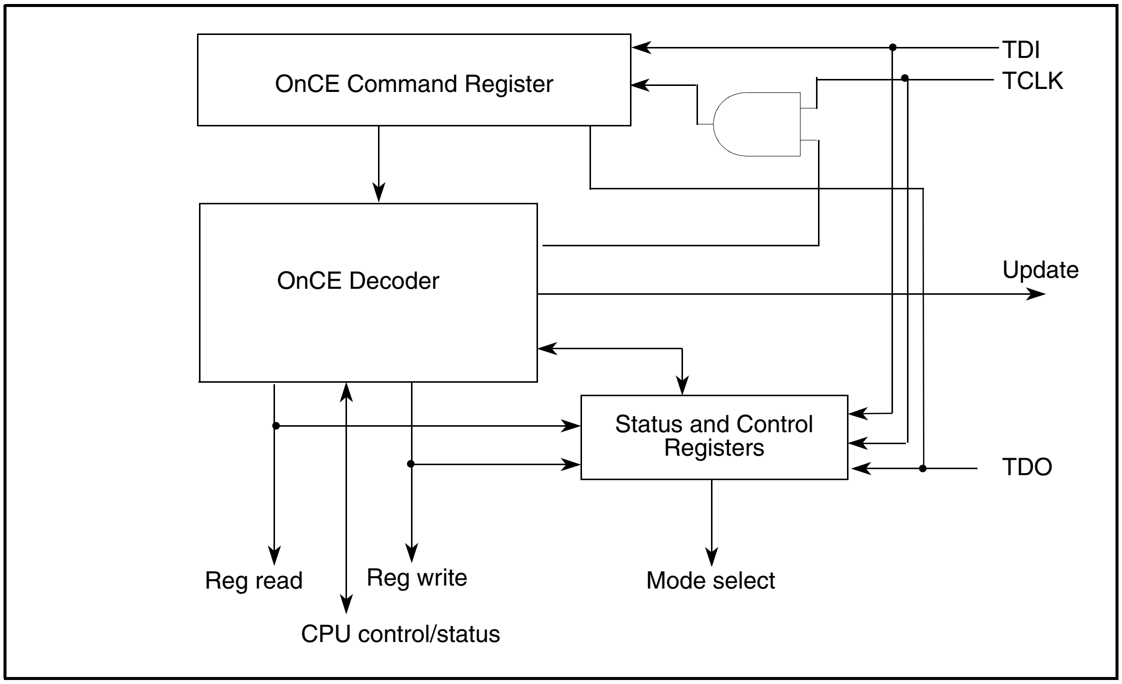


Figure 13-19. e200 OnCE Controller and Serial Interface

13.4.6.1 e200 OnCE Status Register

Status information regarding the state of the CPU is latched into the OnCE Status register, shown in Figure 13-20, when the OnCE controller state machine enters the Capture-IR state. When OnCE operation is enabled, this information is provided on the **j_tdo** output in serial fashion when the Shift_IR state is entered following a Capture-IR. Information is shifted out least significant bit first.

	0	1	2	3	4	5	6	7	8	9
R	MCLK	ERR	0	RESET	HALT	STOP	DEBUG	WAIT	0	1
W										

Figure 13-20. e200 OnCE Status Register

Table 13-22 provides bit definitions for the OnCE status register.

Table 13-22. OnCE Status Register Bit Definitions

Bit(s)	Name	Description
0	MCLK	MCLK **m_clk** Status Bit 0 Inactive state 1 Active state This status bit reflects the logic level on the **jd_mclk_on** input signal after capture by **j_tclk**.
1	ERR	ERROR This bit is used to indicate that an error condition occurred during attempted execution of the last single-stepped instruction (GO+NoExit with CPUSCR or No Register Selected in OCMD), and that the instruction may not have been properly executed. This could occur if an Interrupt (all classes including External, Critical, machine check, Storage, Alignment, Program, TLB, etc.) occurred while attempting to perform the instruction single step. In this case, the CPUSCR will contain information related to the first instruction of the Interrupt handler, and no portion of the handler will have been executed.
2	—	Reserved, set to zero
3	RESET	RESET Mode This bit reflects the inverted logic level on the CPU **p_reset_b** input after capture by **j_tclk**.
4	HALT	HALT Mode This bit reflects the logic level on the CPU **p_halted** output after capture by **j_tclk**.
5	STOP	STOP Mode This bit reflects the logic level on the CPU **p_stopped** output after capture by **j_tclk**.
6	DEBUG	Debug Mode This bit is asserted once the CPU is in debug mode. It is negated once the CPU exits debug mode (even during a debug session)
7	WAIT	Waiting Mode This bit reflects the logic level on the CPU **p_waiting** output after capture by **j_tclk**.
8	0	Reserved, set to 0 to conform to IEEE Std. 1149.1 standard
9	1	Reserved, set to conform to IEEE Std. 1149.1 standard

13.4.6.2 e200 OnCE Command Register (OCMD)

The OnCE command register (OCMD) is a 10-bit shift register that receives its serial data from the TDI pin and serves as the instruction register (IR). It holds the 10-bit commands to be used as input for the e200 OnCE decoder. The command register is shown in Figure 13-21. The OCMD is updated when the TAP controller enters the Update-IR state. It contains fields for controlling access to a resource, as well as controlling single-step operation and exit from OnCE mode.

Although the OCMD is updated during the Update-IR TAP controller state, the corresponding resource is accessed in the DR scan sequence of the TAP controller, and as such, the Update-DR state must be transitioned through in order for an access to occur. In addition, the Update-DR state must also be transitioned through in order for the single-step and/or exit functionality to be performed, even though the command appears to have no data resource requirement associated with it.

Access: Read/Write

	0	1	2	3	4	5	6	7	8	9
R W	R/W	GO	EX	RS[0–6]						
Reset[1]	1	0	0	0	0	0	0	0	1	0

[1] On assertion of **j_trst_b** or **m_por**, or while in the Test_Logic_Reset state

Figure 13-21. OnCE Command Register

Table 13-23 provides bit definitions for the OnCe command register.

Table 13-23. OnCE Command Register Bit Definitions

Bit(s)	Name	Description
0	R/W	Read/Write Command Bit The R/W bit specifies the direction of data transfer. The table below describes the options defined by the R/W bit. 0 Write the data associated with the command into the register specified by RS[0–6] 1 Read the data contained in the register specified by RS[0–6] **Note:** The R/W bit generally ignored for read-only or write-only registers, although the PC FIFO pointer is only guaranteed to be update when R/W = 1. In addition, it is ignored for all bypass operations. When performing writes, most registers are sampled in the Capture-DR state into a 32-bit shift register, and subsequently shifted out on **j_tdo** during the first 32 clocks of Shift-DR.
1	GO	Go Go Command Bit 0 Inactive (no action taken) 1 Execute instruction in IR If the GO bit is set, the chip will execute the instruction which resides in the IR register in the CPUSCR. To execute the instruction, the processor leaves the debug mode, executes the instruction, and if the EX bit is cleared, returns to the debug mode immediately after executing the instruction. The processor goes on to normal operation if the EX bit is set, and no other debug request source is asserted. The GO command is executed only if the operation is a read/write to CPUSCR or a read/write to “No Register Selected”. Otherwise the GO bit is ignored.The processor will leave the debug mode after the TAP controller Update-DR state is entered. On a GO+NoExit operation, returning to debug mode is treated as a debug event, thus exceptions such as machine checks and interrupts may take priority and prevent execution of the intended instruction. Debug firmware should mask these exceptions as appropriate. OSR[ERR] indicates such an occurrence. **Note:** Asynchronous interrupts are blocked on a GO+Exit operation until the first instruction to be executed begins execution. See Section 13.4.9.6, “Exiting Debug Mode and Interrupt Blocking.”

Table 13-23. OnCE Command Register Bit Definitions (continued)

Bit(s)	Name	Description
2	EX	Exit Command Bit 0 Remain in debug mode 1 Leave debug mode If the EX bit is set, the processor will leave the debug mode and resume normal operation until another debug request is generated. The Exit command is executed only if the Go command is issued, and the operation is a read/write to CPUSCR or a read/write to “No Register Selected”. Otherwise the EX bit is ignored. The processor will leave the debug mode after the TAP controller Update-DR state is entered. **Note:** If the DR bit in the OnCE control register is set or remains set, or if a bit in the EDBSR0 is set and EDBCR0[EDM] = 1 (external debug mode is enabled), or if another debug request source is asserted, then the processor may return to the debug mode without execution of an instruction, even though the EX bit was set. **Note:** Asynchronous interrupts are blocked on a GO+Exit operation until the first instruction to be executed begins execution. See Section 13.4.9.6, “Exiting Debug Mode and Interrupt Blocking.”
3–9	RS	Register Select The register select bits define which register is source (destination) for the read (write) operation. Table 13-24 shows the e200 OnCE register addresses. Attempted writes to read-only registers are ignored.

Table 13-24 shows the e200 OnCE register addresses.

Table 13-24. e200 OnCE Register Addressing

RS[0–6]	Register Selected
000 0000	Reserved
000 0001	Reserved
000 0010	JTAG ID (read-only)
000 0011–000 1111	Reserved
001 0000	CPU Scan Register (CPUSCR)
001 0001	No Register Selected (Bypass)
001 0010	OnCE Control Register (OCR)
001 0011	Reserved
001 0100– 001 1111	Reserved
010 0000	Instruction Address Compare 1 (IAC1)
010 0001	Instruction Address Compare 2 (IAC2)
010 0010	Instruction Address Compare 3 (IAC3)
010 0011	Instruction Address Compare 4 (IAC4)
010 0100	Data Address Compare 1 (DAC1)
010 0101	Data Address Compare 2 (DAC2)
010 0110	Data Value Compare 1 (DVC1)

Table 13-24. e200 OnCE Register Addressing

RS[0–6]	Register Selected
010 0111	Data Value Compare 2 (DVC2)
010 1000	Instruction Address Compare 5 (IAC5)
010 1001	Instruction Address Compare 6 (IAC6)
010 1010	Instruction Address Compare 7 (IAC7)
010 1011	Instruction Address Compare 8 (IAC8)
010 1100	Debug Counter Register (DBCNT)
010 1101	Debug PCFIFO (PCFIFO)
010 1110	External Debug Control Register 0 (EDBCR0)
010 1111	External Debug Status Register 0 (EDBSR0)
011 0000	Debug Status Register (DBSR)
011 0001	Debug Control Register 0 (DBCR0)
011 0010	Debug Control Register 1 (DBCR1)
011 0011	Debug Control Register 2 (DBCR2)
011 0100	Debug Control Register 3 (DBCR3)
011 0101	Debug Control Register 4 (DBCR4)
011 0110	Debug Control Register 5 (DBCR5)
011 0111	Debug Control Register 6 (DBCR6)
011 1000– 011 1011	Reserved (do not access)
011 1100	External Debug Status Register MASK 0 (EDBSRMSK0)
011 1101	Debug Data Acquisition Message Register (DDAM)
011 1110	Debug Event Control (DEVENT)
011 1111	Debug External Resource Control (DBERC0)
100 0000– 110 1110	Reserved (do not access)
110 1111	Reserved for Shared Nexus Control Register Select
111 0000– 111 1001	General Purpose register selects [0–9]
111 1010	Cache Debug Access Control Register (CDACNTL) (see Section 9.19, “Cache Memory Access For Debug/Error Handling)
111 1011	Cache Debug Access Data Register (CDADATA) (see Section 9.19, “Cache Memory Access For Debug/Error Handling)
111 1100	Nexus3-Access (see Chapter 14, “Nexus 3 Module)
111 1101	LSRL Select (see Test Specification)

Table 13-24. e200 OnCE Register Addressing

RS[0–6]	Register Selected
111 1110	Enable_OnCE[1]
111 1111	Bypass

[1] Causes assertion of the j_en_once_regsel output. Refer to Section 11.2.25.8, “Enable Once Register Select (j_en_once_regsel)

The OnCE decoder receives as input the 10-bit command from the OCMD, and status signals from the processor, and generates all the strobes required for reading and writing the selected OnCE registers.

Single stepping of instructions is performed by placing the CPU in debug mode, scanning in appropriate information into the CPUSCR, and setting the Go bit (with the EX bit cleared) with the RS field indicating either the CPUSCR or No Register Selected. After executing a single instruction, the CPU re-enters debug mode and awaits further commands. During single-stepping, exception conditions may occur if not properly masked by debug firmware (interrupts, machine checks, bus error conditions, etc.) and may prevent the desired instruction from being successfully executed. The OSR[ERR] is set to indicate this condition. In these cases, values in the CPUSCR will correspond to the first instruction of the exception handler.

Additionally, EDBCR0[EDM]/DBCR0[EDM] is forced to 1 internally while single-stepping to prevent debug events from generating debug interrupts. Also, during a debug session, the DBSR and the DBCNT registers are frozen from updates due to debug events regardless of EDBCR0[EDM]/DBCR0[EDM]. They may still be modified during a debug session via a single-stepped **mtspr** instruction or via OnCE access if EDBCR0[EDM]/DBCR0[EDM] is set.

13.4.6.3 e200 OnCE Control Register (OCR)

The e200 OnCE Control Register is a 32-bit register used to force the e200 core into debug mode and to enable/disable sections of the e200 OnCE control logic. It also provides control over the MMU during a debug session (see Section 13.6, “MMU and Cache Operation During Debug”). The control bits are read/write. These bits are only effective while OnCE is enabled (**jd_en_once** asserted). The OCR is shown in Figure 13-22.

Access: Read/Write

	0–7	8	9–10	11	12	13	14	15
R W	—	I_ DMDIS	—	I_DVLE	I_DI	I_DM	—	I_DE
Reset	All zeros[1]							

	16	17–18	19	20	21	22	23	24–28	29	30	31
R W	D_ DMDIS	—	D_DW	D_DI	D_DM	D_DG	D_DE	—	WKUP	FDB	DR
Reset	All zeros[1]										

[1] All zeros on **m_por**, **j_trst_b**, or entering Test_logic_Reset state

Figure 13-22. OnCE Control Register

Table 13-25 provides bit definitions for the OnCE control register.

Table 13-25. OnCE Control Register Bit Definitions

Bit(s)	Name	Description
0–7	—	Reserved
8	I_DMDIS	Instruction Side Debug MMU Disable Control Bit (I_DMDIS) 0 MMU not disabled for debug sessions 1 MMU disabled for debug sessions This bit may be used to control whether the MMU is enabled normally, or whether the MMU is disabled during a debug session for Instruction Accesses. When enabled, the MMU functions normally. When disabled, for Instruction Accesses, no address translation is performed (1:1 address mapping), and the TLB VLE, I,M, and E bits are taken from the OCR bits I_VLE, I_DI, I_DM, and I_DE bits. The W and G bits are assumed 0. The SX and UX access permission control bits are set to1 to allow full access. When disabled, no TLB miss or TLB exceptions are generated for Instruction accesses. External access errors can still occur.
9–10	—	Reserved
11	I_DVLE	Instruction Side Debug TLB 'VLE' Attribute Bit (I_DVLE) This bit is used to provide the 'VLE' attribute bit to be used when the MMU is disabled during a debug session.
12	I_DI	Instruction Side Debug TLB 'I' Attribute Bit (I_DI) This bit is used to provide the 'I' attribute bit to be used for Instruction accesses when the MMU is disabled for Instruction accesses during a debug session.
13	I_DM	Instruction Side Debug TLB 'M' Attribute Bit (I_DM) This bit is used to provide the 'M' attribute bit to be used for Instruction accesses when the MMU is disabled for Instruction accesses during a debug session.
14	—	Reserved
15	I_DE	Instruction Side Debug TLB 'E' Attribute Bit (I_DE) This bit is used to provide the 'E' attribute bit to be used for Instruction accesses when the MMU is disabled for Instruction accesses during a debug session.
16	D_DMDIS	Data Side Debug MMU Disable Control Bit (D_DMDIS) 0 MMU not disabled for debug sessions 1 MMU disabled for debug sessions This bit may be used to control whether the MMU is enabled normally, or whether the MMU is disabled during a debug session for Data Accesses. When enabled, the MMU functions normally. When disabled, for Data Accesses, no address translation is performed (1:1 address mapping), and the TLB WIMGE bits are taken from the OCR bits D_DW, D_DI, D_DM, D_DG, and D_DE bits. The SR, SW, UR, and UW access permission control bits are set to1 to allow full access. When disabled, no TLB miss or TLB exceptions are generated for Data accesses. External access errors can still occur.
17–18	—	Reserved
19	D_DW	Data Side Debug TLB 'W' Attribute Bit (D_DW) This bit is used to provide the 'W' attribute bit to be used for Data accesses when the MMU is disabled for Data accesses during a debug session.
20	D_DI	Data Side Debug TLB 'I' Attribute Bit (D_DI) This bit is used to provide the 'I' attribute bit to be used for Data accesses when the MMU is disabled for Data accesses during a debug session.
21	D_DM	Data Side Debug TLB 'M' Attribute Bit (D_DM) This bit is used to provide the 'M' attribute bit to be used for Data accesses when the MMU is disabled for Data accesses during a debug session.

Table 13-25. OnCE Control Register Bit Definitions (continued)

Bit(s)	Name	Description
22	D_DG	Data Side Debug TLB 'G' Attribute Bit (D_DG) This bit is used to provide the 'G' attribute bit to be used for Data accesses when the MMU is disabled for Data accesses during a debug session.
23	D_DE	Data Side Debug TLB 'E' Attribute Bit (D_DE) This bit is used to provide the 'E' attribute bit to be used for Data accesses when the MMU is disabled for Data accesses during a debug session.
24–28	—	Reserved
29	WKUP	Wakeup Request Bit (WKUP) This control bit may be used to force the e200 **p_wakeup** output signal to be asserted. This control function may be used by debug firmware to request that the chip-level clock controller restore the **m_clk** input to normal operation regardless of whether the CPU is in a low power state to ensure that debug resources may be properly accessed by external hardware through scan sequences.
30	FDB	Force Breakpoint Debug Mode Bit (FDB) • This control bit is used to determine whether the processor is operating in breakpoint debug enable mode or not. The processor may be placed in breakpoint debug enable mode by setting this bit. In breakpoint debug enable mode, execution of the '**bkpt**' pseudo- instruction will cause the processor to enter debug mode, as if the **jd_de_b** input had been asserted. • This bit is qualified with DBCR0[EDM], which must be set for FDB to take effect. **Note:** This bit has no effect on **dnh** or **se_dnh** instruction operation.
31	DR	CPU Debug Request Control Bit This control bit is used to unconditionally request the CPU to enter the debug mode. The CPU indicates that debug mode has been entered via the data scanned out in the shift-IR state. 0 No Debug Mode request 1 Unconditional Debug Mode request When the DR bit is set, the processor enters debug mode at the next instruction boundary.

13.4.7 Access to Debug Resources

Resources contained in the e200 OnCE module that do not require the e200 processor core to be halted for access may be accessed while the e200 core is running. They do not interfere with processor execution. Accesses to other resources such as the CPUSCR require the e200 core to be placed in debug mode to avoid synchronization hazards. Debug firmware may ensure that it is safe to access these resources by determining the state of the e200 core prior to access. Note that a scan operation to update the CPUSCR is required prior to exiting debug mode if debug mode has been entered.

Some cases of write accesses other than accesses to the OnCE Command and Control registers, or the EDM bit of DBCR0 require the e200 **m_clk** to be running for proper operation. The OnCE control register provides a means of signaling this need to a system level clock control module.

In addition, since the CPU may cause multiple bits of certain registers to change state, reads of certain registers while the CPU is running (DBSR, DBCNT, etc.) may not have consistent bit settings unless read twice with the same value indicated. In order to guarantee that the contents are consistent, the CPU should be placed into debug mode or multiple reads should be performed until consistent values have been obtained on consecutive reads.

Table 13-26 provides a list of access requirements for OnCE registers.

Table 13-26. OnCE Register Access Requirements

Register Name	Access Requirements					Notes
	Requires jd_en_once to be asserted	Requires DBCR0 [EDM] = 1	Requires m_clk active for Write Access	Requires CPU to be halted for Read Access	Requires CPU to be halted for Write Access	
Enable_OnCE	N	N	N	N	—	—
Bypass	N	N	N	N	N	—
CPUSCR	Y	Y	Y	Y	Y	—
DAC1	Y	Y	Y	N	*[1]	—
DAC2	Y	Y	Y	N	*[1]	—
DBCNT	Y	Y	Y	N[2]	*[1]	—
DBCR0	Y	Y	Y	N	*[1]	DBCR0[EDM] access only requires **jd_en_once** asserted
DBCR1–6	Y	Y	Y	N	*[1]	—
DEVENT	Y	Y	Y	N	*[1]	—
DBERC0	Y	N	Y	N	*[1]	—
DBSR	Y	Y	Y	N[2]	*[1]	—
EDBCR0	Y	N	N	N	N	—
EDBSR0	Y	N	N	N	N	—
EDBSRMSK0	Y	N	N	N	N	—
IAC1–8	Y	Y	Y	N	*[1]	—
JTAG ID	N	N	—	N	—	Read-only
OCR	Y	N	N	N	N	—
OSR	Y	N	—	N	—	Read-only, accessed by scanning out IR while **jd_en_once** is asserted
PC FIFO	Y	N	Y	N	N	Updates frozen while OCMD holds PCFIFO register encoding. **Note:** No updates occur to the PCFIFO while the OnCE state machine is in the Test_Logic_Reset state
Cache Debug Access Control (CDACNTL)	Y	N	Y	Y	Y	CPU must be in debug mode with clocks running
Cache Debug Access Data (CDADATA)	Y	N	Y	Y	Y	CPU must be in debug mode with clocks running
Nexus3-Access	Y	N	N	N	N	—

Table 13-26. OnCE Register Access Requirements (continued)

Register Name	Access Requirements					Notes
	Requires jd_en_once to be asserted	Requires DBCR0 [EDM] = 1	Requires m_clk active for Write Access	Requires CPU to be halted for Read Access	Requires CPU to be halted for Write Access	
External GPRs	Y	N	N	N	N	—
LSRL Select	Y	N	?	?	?	System Test logic implementation determines LSRL functionality

[1] Writes to these registers while the CPU is running may have unpredictable results due to the pipelined nature of operation, and the fact that updates are not synchronized to a particular clock, instruction, or bus cycle boundary, therefore it is strongly recommended to ensure the processor is first placed into debug mode before updates to these registers are performed.

[2] Reads of these registers while the CPU is running may not give data that is self-consistent due to synchronization across clock domains.

13.4.8 Methods of Entering Debug Mode

The OnCE status register indicates that the CPU has entered the debug mode via the DEBUG status bit. The following sections describe how the e200 debug mode is entered, assuming the OnCE circuitry has been enabled. e200 OnCE operation is enabled by the assertion of the **jd_en_once** input (see Section 13.4.5.1, “OnCE Enable (jd_en_once)”).

13.4.8.1 External Debug Request During RESET

Holding the **jd_de_b** signal asserted during the assertion of **p_reset_b** and continuing to hold it asserted following the negation of **p_reset_b** causes the e200 core to enter debug mode. After receiving an acknowledge via the OnCE status register DEBUG bit, the external command controller should negate the **jd_de_b** signal before sending the first command. Note that in this case the e200 core does not execute an instruction before entering debug mode, although the first instruction to be executed may be fetched prior to entering debug mode.

In this case, all values in the debug scan chain are undefined, and the external debug control module is responsible for proper initialization of the chain before debug mode is exited. In particular, the exception processing associated with reset may not be performed when the debug mode is exited. The debug controller must initialize the PC, MSR, and IR to the image that the processor would have obtained in performing reset exception processing or must cause the appropriate reset to be re-asserted.

13.4.8.2 Debug Request During RESET

Asserting a debug request by setting the DR bit in the OCR during the assertion of **p_reset_b** causes the chip to enter debug mode. In this case the chip may fetch the first instruction of the reset exception handler, but does not execute an instruction before entering debug mode. In this case, all values in the debug scan chain are undefined, and the external debug control module is responsible for proper initialization of the chain before debug mode is exited. In particular, the exception processing associated with reset may not be performed when the debug mode is exited, thus, the debug controller must initialize the PC, MSR, and IR to the image that the processor would have obtained in performing reset exception processing or must cause the appropriate reset to be re-asserted.

13.4.8.3 Debug Request During Normal Activity

Asserting a debug request by setting the DR bit in the OCR during normal chip activity causes the chip to finish the execution of the current instruction and then enter the debug mode. Note that in this case the chip completes the execution of the current instruction and stops after the newly fetched instruction enters the CPU instruction register. This process is the same for any newly fetched instruction including instructions fetched by the interrupt processing or those that will be aborted by the interrupt processing.

13.4.8.4 Debug Request During Waiting, Halted, or Stopped State

Asserting a debug request by setting the DR bit in the OCR when the chip is in the waiting state (**p_waiting** asserted), halted state (**p_halted** asserted) or stopped state (**p_stopped** asserted) causes the CPU to exit the state and enter the debug mode once the CPU clock **m_clk** has been restored. Note that in this case, the CPU negates the **p_waiting, p_halted** and **p_stopped** outputs. Once the debug session has ended, the CPU returns to the state it was in prior to entering debug mode.

To signal the chip-level clock generator to re-enable **m_clk**, the **p_wakeup** output is asserted whenever the debug block asserts a debug request to the CPU due to OCR[DR] being set or **jd_de_b** assertion. It remains set from then until the debug session ends (**jd_debug_b** goes from asserted to negated). In addition, the status of the **jd_mclk_on** input (after synchronization to the **j_tclk** clock domain) may be sampled along with other status bits from the **j_tdo** output during the Shift_IR TAP controller state. This status may be used if necessary by external debug firmware to ensure proper scan sequences occur to registers in the **m_clk** clock domain.

13.4.8.5 Software Request During Normal Activity

Upon executing a '**bkpt**' pseudo-instruction (for e200, defined to be an all 0's instruction opcode) when the OCR register's (FDB) bit is set (debug mode enable control bit is true), and DBCR0[EDM] = 1, the CPU enters the debug mode after the instruction following the '**bkpt**' pseudo-instruction has entered the instruction register.

13.4.8.6 Debug Notify Halt Instructions

The **dnh, e_dnh,** and **se_dnh** instructions allow software to transition the core from a running state to a debug halted state if enabled by EDBCR0[DNH_EN]. They also provide the external debugger with bits reserved in the instruction itself to pass additional information. Entry into debug mode is not conditioned on EDBCR0[EDM], allowing for debug of software debug handlers as well as other software.

When the CPU enters a debug halted state due to a **dnh**, **e_dnh**, or **se_dnh** instruction, the instruction is stored in the CPUSCR[IR] portion, and the CPUSCR[PC] value points to the instruction. The external debugger should update the CPUSCR prior to exiting the debug halted state to point past the **dnh**, **e_dnh**, or **se_dnh** instruction.

13.4.9 CPU Status and Control Scan Chain Register (CPUSCR)

A number of on-chip registers store the CPU pipeline status and are configured in a single scan chain for access by the e200 OnCE controller. The CPUSCR register contains these processor resources, which are used to restore the pipeline and resume normal chip activity upon return from the debug mode, as well as a mechanism for the emulator software to access processor and memory contents.

Figure 13-23 shows the block diagram of the pipeline information registers contained in the CPUSCR. Once debug mode has been entered, it is required to scan in and update this register prior to exiting debug mode.

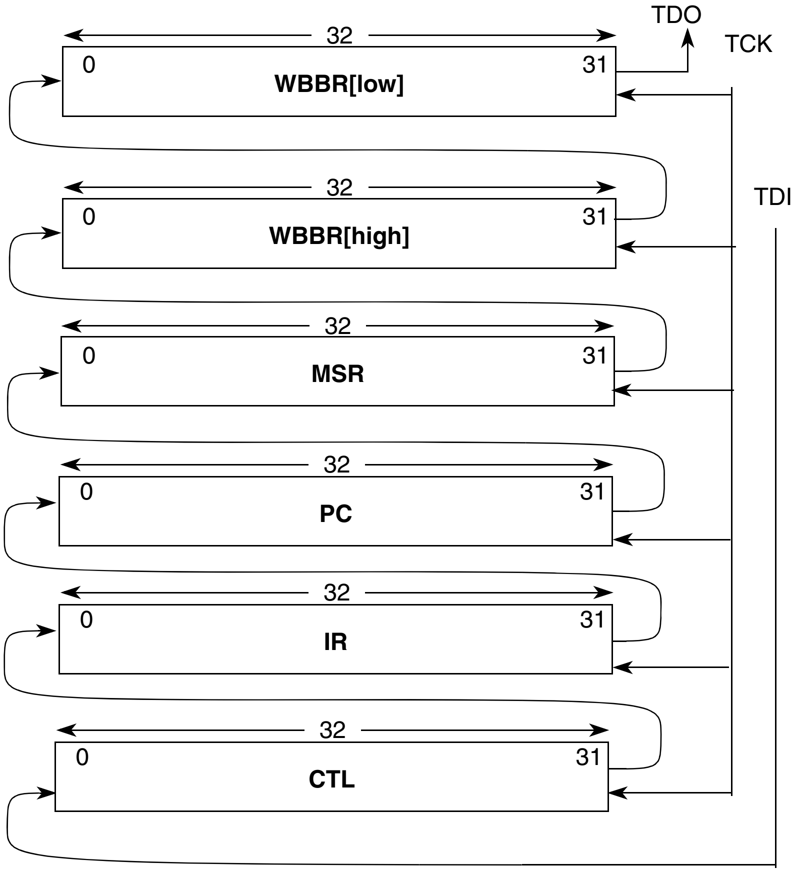


Figure 13-23. CPU Scan Chain Register (CPUSCR)

13.4.9.1 Instruction Register (IR)

The instruction register (IR) provides a mechanism for controlling the debug session by serving as a means for forcing in selected instructions and then causing them to be executed in a controlled manner by the debug control block. The opcode of the next instruction to be executed when entering debug mode is contained in this register when the scan-out of this chain begins. This value should be saved for later restoration if continuation of the normal instruction stream is desired.

On scan-in, in preparation for exiting debug mode, this register is filled with an instruction opcode selected by debug control software. By selecting appropriate instructions and controlling the execution of those instructions, the results of execution may be used to examine or change memory locations and processor registers. The debug control module external to the processor core controls execution by providing a single-step capability. Once the debug session is complete and normal processing is to be resumed, this register may be loaded with the value originally scanned out.

13.4.9.2 Control State Register (CTL)

The control state register (CTL), shown in Figure 13-24, is a 32-bit register that stores the value of certain internal CPU state variables before the debug mode is entered. This register is affected by the operations performed during the debug session and should normally be restored by the external command controller when returning to normal mode. In addition to saved internal state variables, two of the bits are used by emulation firmware to control the debug process. In certain circumstances, emulation firmware must modify the content of this register as well as the PC and IR values in the CPUSCR before exiting debug mode. These cases are described below.

Access: Read/Write

	0–10	11	12	13	14	15
R W	*	IRSTAT 13	IRSTAT 12	IRSTAT 11	IRSTAT 10	WAITING

	16–19	20	21	22	23	24	25	26	27	28	29	30	31
R W	PCOFST	PCINV	FFRA	IRSTAT 0	IRSTAT 1	IRSTAT 2	IRSTAT 3	IRSTAT 4	IRSTAT 5	IRSTAT 6	IRSTAT 7	IRSTAT 8	IRSTAT 9

Figure 13-24. Control State Register (CTL)

Table 13-27 describes the control state register fields.

Table 13-27. Control State Register Field Descriptions

Bit	Name	Description
0–10	*	Internal State Bits These control bits represent internal processor state and should be restored to their original value after a debug session is completed, such as when a e200 OnCE command is issued with the GO and EX bits set and not ignored. When performing instruction execution during a debug session (see Section 13.4.5.3, “e200 OnCE Debug Output (jd_debug_b)”) that is not part of the normal program execution flow, these bits should be set to a 0.
11	IRStat13	IR Status Bit 13 This control bit indicates an Instruction Address Compare 8 event status for the IR. 0 No Instruction Address Compare 8 event occurred on the fetch of this instruction. 1 An Instruction Address Compare 8 event occurred on the fetch of this instruction.
12	IRStat12	IR Status Bit 12 This control bit indicates an Instruction Address Compare 7 event status for the IR. 0 No Instruction Address Compare 7 event occurred on the fetch of this instruction. 1 An Instruction Address Compare 7 event occurred on the fetch of this instruction.

Table 13-27. Control State Register Field Descriptions (continued)

Bit	Name	Description
13	IRStat11	IR Status Bit 11 This control bit indicates an Instruction Address Compare 6 event status for the IR. 0 No Instruction Address Compare 6 event occurred on the fetch of this instruction. 1 An Instruction Address Compare 6 event occurred on the fetch of this instruction.
14	IRStat10	IR Status Bit 10 This control bit indicates an Instruction Address Compare 5 event status for the IR. 0 No Instruction Address Compare 5 event occurred on the fetch of this instruction. 1 An Instruction Address Compare 5 event occurred on the fetch of this instruction.
15	Waiting	WAITING State Status This bit indicates whether the CPU was in the waiting state prior to entering debug mode. If set, the CPU was in the waiting state. Upon exiting a debug session, the value of this bit in the restored CPUSCR will determine whether the CPU re-enters the waiting state on a go+exit. 0 CPU was not in the waiting state when debug mode was entered 1 CPU was in the waiting state when debug mode was entered
16–19	PCOFST	PC Offset Field This field indicates whether the value in the PC portion of the CPUSCR must be adjusted prior to exiting debug mode. Due to the pipelined nature of the CPU, the PC value must be backed-up by emulation software in certain circumstances. The PCOFST field specifies the value to be subtracted from the original value of the PC. This adjusted PC value should be restored into the PC portion of the CPUSCR just prior to exiting debug mode with a go+exit. In the event the PCOFST is non-zero, the IR should be loaded with a no-op instruction instead of the original IR value, otherwise the original value of IR should be restored. (But see PCINV which overrides this field) 0000 No correction required. 0001 Subtract 0x04 from PC. 0010 Subtract 0x08 from PC. 0011 Subtract 0x0C from PC. 0100 Subtract 0x10 from PC. 0101 Subtract 0x14 from PC. All other encodings are reserved
20	PCINV	PC and IR Invalid Status Bit This status bit indicates that the values in the IR and PC portions of the CPUSCR are invalid. Exiting debug mode with the saved values in the PC and IR will have unpredictable results. Debug firmware should initialize the PC and IR values in the CPUSCR with desired values prior to exiting debug mode if this bit was set when debug mode was initially entered. 0 No error condition exists. 1 Error condition exists. PC and IR are corrupted.
21	FFRA	Feed Forward RA Operand Bit This control bit causes the content of the WBBR to be used as the RA operand value (RS for logical, mtspr, mtdcr, cntlzw, and shift operations, RX for VLE se_ instructions, RT for e_{logical_op}2i type instructions, RB for **evaddiw**, **evsubifw**, and the value to use as the PC for calculating the LR update value for branch with link type instructions) of the first instruction to be executed following an update of the CPUSCR. This allows the debug firmware to update processor register—initialize the WBBR with the desired value, set the FFRA bit, and execute a ori Rx,Rx,0 instruction to the desired register. 0 No action. 1 Content of WBBR used as operand value

Table 13-27. Control State Register Field Descriptions (continued)

Bit	Name	Description
22	IRStat0	IR Status Bit 0 This control bit indicates a TEA status for the IR. 0 No TEA occurred on the fetch of this instruction. 1 TEA occurred on the fetch of this instruction.
23	IRStat1	IR Status Bit 1 This control bit indicates a TLB Miss status for the IR. 0 No TLB Miss occurred on the fetch of this instruction. 1 TLB Miss occurred on the fetch of this instruction.
24	IRStat2	IR Status Bit 2 This control bit indicates an Instruction Address Compare 1 event status for the IR. 0 No Instruction Address Compare 1 event occurred on the fetch of this instruction. 1 An Instruction Address Compare 1 event occurred on the fetch of this instruction.
25	IRStat3	IR Status Bit 3 This control bit indicates an Instruction Address Compare 2 event status for the IR. 0 No Instruction Address Compare 2 event occurred on the fetch of this instruction. 1 An Instruction Address Compare 2 event occurred on the fetch of this instruction.
26	IRStat4	IR Status Bit 4 This control bit indicates an Instruction Address Compare 3 event status for the IR. 0 No Instruction Address Compare 3 event occurred on the fetch of this instruction. 1 An Instruction Address Compare 3 event occurred on the fetch of this instruction.
27	IRStat5	IR Status Bit 5 This control bit indicates an Instruction Address Compare 4 event status for the IR. 0 No Instruction Address Compare 4 event occurred on the fetch of this instruction. 1 An Instruction Address Compare 4 event occurred on the fetch of this instruction.
28	IRStat6	IR Status Bit 6 This control bit indicates a Parity Error status for the IR. 0 No Parity Error occurred on the fetch of this instruction. 1 Parity Error occurred on the fetch of this instruction.
29	IRStat7	IR Status Bit 7 This control bit indicates a Precise External Termination Error status for the IR, or a 2nd half TLB Miss for the instruction in the IR. 0 No Precise External Termination Error occurred on the fetch of this instruction. 1 If IRStat1 = 0, a Precise External Termination Error occurred on the fetch of this instruction. If IRStat1 = 1, a TLB Miss occurred on the 2nd half of this instruction.
30	IRStat8	IR Status Bit 8 This control bit indicates the Power ISA VLE status for the IR. 0 IR contains a Power ISA instruction. 1 IR contains a Power ISA VLE instruction, aligned in the most significant portion of IR if 16-bit.
31	IRStat9	IR Status Bit 9 This control bit indicates the Power ISA VLE Byte-ordering Error status for the IR, or a Power ISA misaligned instruction fetch, depending on the state of IRStat8. 0 IR contains an instruction without a byte-ordering error and no misaligned instruction fetch exception has occurred (no MIF). 1 If IRStat8 = 0, A Power ISA misaligned instruction fetch exception has occurred while filling the IR. If IRStat8 = 1, IR contains an instruction with a byte-ordering error due to mismatched VLE page attributes, or due to E indicating little-endian for a VLE page.

Emulation firmware should modify the content of the CTL, PC, and IR values in the CPUSCR during execution of debug related instructions as well as just prior to exiting debug with a go + exit command. During the debug session, the CTL register should be written with the FFRA bit set as appropriate, all other bits set to 0, and the IR set to the value of the desired instruction to be executed. IRStat8 is used to determine the type of instruction present in the IR.

Just prior to exiting debug mode with a go+exit, the PCINV status bit that was originally present when debug mode was first entered should be tested. If set, the PC and IR should be initialized for performing whatever recovery sequence is appropriate for a faulted exception vector fetch. If the PCINV bit is cleared, the PCOFST bits should be examined to determine whether the PC value must be adjusted. Due to the pipelined nature of the CPU, the PC value must be backed-up by emulation software in certain circumstances. The PCOFST field specifies the value to be subtracted from the original value of the PC. This adjusted PC value should be restored in to the PC portion of the CPUSCR just prior to exiting debug mode with a go+exit. In the event the PCOFST is non-zero, the IR should be loaded with a no-op instruction (such as **ori r0,r0,0**) instead of the original IR value, otherwise the original value of IR should be restored. Note that when a correction is made to the PC value, it generally points to the last completed instruction, although that instruction will not be re-executed. The no-op instruction is executed instead, and instruction fetch and execution resume at location PC+4. IRStat8 is used to determine the type of instruction present in the IR, thus should be cleared in this case. Note that debug events that may occur on the no-op (ICMP) will be generated (and optionally counted) if enabled.

For the CTL register, the internal state bits should be restored to their original value. The IR status bits should be set to 0s if the PC was adjusted. If no PC adjustment was performed, emulation firmware should determine whether IRStat2–5 should be set to 0 to avoid re-entry into debug mode for an instruction breakpoint request. Upon exiting debug mode with go+exit, if one of these bits is set, debug mode will be re-entered prior to any further instruction execution.

13.4.9.3 Program Counter Register (PC)

The PC is a 32-bit register that stores the value of the program counter which was present when the chip entered the debug mode. It is affected by the operations performed during the debug mode and must be restored by the external command controller when the CPU returns to normal mode. The PC normally points to the instruction contained in the IR portion of CPUSCR. If debug firmware wishes to redirect program flow to an arbitrary location, the PC and IR should be initialized to correspond to the first instruction to be executed upon resumption of normal processing. Alternatively, the IR may be set to a no-op and the PC set to point to the location prior to the location at which it is desired to redirect flow to. On exiting debug mode, the no-op is executed, and instruction fetch and execution resume at PC + 4.

13.4.9.4 Write-Back Bus Register (WBBR[low], WBBR[high])

WBBR is used as a means of passing operand information between the CPU and the external command controller. Whenever the external command controller needs to read the contents of a register or memory location, it forces the chip to execute an instruction that brings that information to WBBR. WBBR[low] holds the 32-bit result of most instructions including load data returned for a load or load with update instruction. For SPE/EFPU instructions that generate 64-bit results, WBBR[low] holds the low-order 32 bits of the result. WBBR[high] holds the updated effective address calculated by a load with update instruction. For SPE/EFPU instructions which generate 64-bit results, WBBR[high] holds the high-order 32 bits of the result. It is undefined for other instructions.

As an example, to read the lower 32 bits of processor register **r1**, an **ori r1,r1,0** instruction is executed, and the result value of the instruction is latched into WBBR[low]. The contents of WBBR[low] can be delivered serially to the external command controller. To update a processor resource, this register is initialized with a data value to be written, and an **ori** instruction is executed which uses this value as a substitute data value. The control state register FFRA bit forces the value of the WBBR[low] to be substituted for the normal RS source value of the **ori** instruction, thus allowing updates to processor registers to be performed (refer to Section 13.4.9.2, "Control State Register (CTL)," for more detail on CTL[FFRA]).

WBBR[low] and WBBR[high] are generally undefined on instructions that do not write back a result and, due to control issues, are also not defined on **lmw** or branch instructions.

To read and write the entire 64 bits of a GPR, both WBBR[low] and WBBR[high] are used. For reads, an **evslwi** $\mathbf{r_n,r_n,0}$ may be used. For writes, the same instruction may be used, but CTL[FFRA] must be set as well. Note that MSR[SPE] must be set in order for these operations to be performed properly.

13.4.9.5 Machine State Register (MSR)

The MSR is a 32-bit register used to read/write the machine state register. Whenever the external command controller needs to save or modify the contents of the machine state register, this register is used. This register is affected by the operations performed during the debug mode and must be restored by the external command controller when returning to normal mode.

13.4.9.6 Exiting Debug Mode and Interrupt Blocking

When exiting debug mode with a Go+Exit, "asynchronous" interrupts are blocked until the first instruction to be executed begins execution. This includes External and Critical input, NMI, machine check, timer, decrementer, and watchdog interrupts. Asynchronous debug interrupts are not blocked however, and the CPU will re-enter debug mode without executing an instruction following Go+Exit, although it may fetch an instruction and discard it. Exceptions due to an illegal instruction or error flags set within the CPUSCR CTL register are not blocked because they apply to the instruction in the CPUSCR IR.

13.4.10 Instruction Address FIFO Buffer (PC FIFO)

To assist debugging and keeping track of program flow, a First-In-First-Out (FIFO) buffer stores the addresses of the last eight instruction change of flow destinations that were fetched. These include exception vectoring to an exception handler and returns, as well as pipeline refills due to execution of the **isync** instruction.

13.4.10.1 PC FIFO

The PC FIFO stores the addresses of the last eight instruction change of flow addresses that were actually taken. The FIFO is implemented as a circular buffer containing eight 32-bit registers and one 3-bit counter. All the registers have the same address, but any access to the FIFO address causes the counter to increment, making it point to the next FIFO register. The registers are serially available to the external command controller through the common FIFO address.

Figure 13-25 shows the block diagram of the PC FIFO.

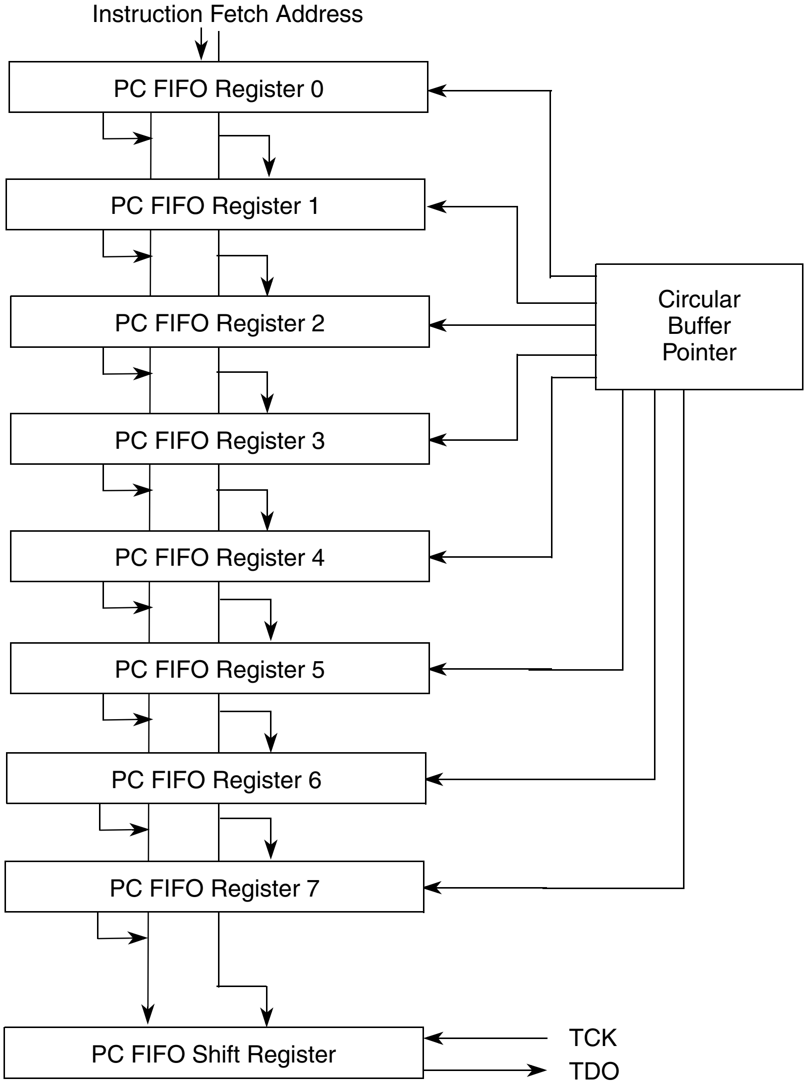


Figure 13-25. OnCE PC FIFO

The FIFO is not affected by the operations performed during a debug session except for the FIFO pointer increment when accessing the FIFO. When entering debug mode, the FIFO counter points to the FIFO register containing the address of the oldest of the eight change of flow prefetches. When the OCMD RS field is loaded with the value corresponding to the PC FIFO (010 1101), the current pointer value is captured into a temporary register. This temporary value (not the actual FIFO counter) is incremented as FIFO reads or writes are performed. The first FIFO read obtains the oldest address, and the following FIFO reads return the other addresses from the oldest to the newest (the order of execution). Writes operate similarly.

Updates to the FIFO by change of flows are frozen whenever the OCMD register contains a command whose RS[0–6] field points to the PC FIFO (010 1101) to allow firmware to access the contents of the PC

FIFO without placing the CPU into debug mode. After completing all accesses to the PC FIFO, another OCMD value that does not select the PC FIFO should be entered to allow the PC FIFO to resume updating.

To ensure FIFO coherence, a complete set of eight accesses of the FIFO should be performed because each access increments the temporary FIFO pointer, thus making it point to the next location. After eight accesses, the pointer points to the same location it pointed to before starting the access procedure. The temporary counter value captures the actual counter each time the OCMD RS field transitions to the value corresponding to the PC FIFO (010 1101).

The FIFO pointer is reset to entry 0 when either **j_trst_b** or **m_por** are asserted.

13.4.11 Reserved Registers (Reserved)

The reserved registers are used to control various test control logic. These registers are not intended for customer use. To preclude device and/or system damage, these registers should not be accessed.

13.5 Watchpoint Support

The e200 supports the generation and signalling of watchpoints when operating in internal debug mode (DBCR0[IDM] = 1) or in external debug mode (DBCR0[EDM] = 1). Watchpoints are indicated with a dedicated set of interface signals. The **jd_watchpt[0:29]** output signals are used to indicate that a watchpoint has occurred. Certain watchpoints (DEVNT-based) are not qualified with DBCR0[EDM] or DBCR0[IDM].

Each debug address compare function (IAC1–8, DAC1–2), debug counter event (DCNT1–2), and other event types are capable of triggering a watchpoint output. The DBCRx control fields are used to configure watchpoints, regardless of whether events are enabled in DBCR0. Watchpoints may occur whenever an associated event would have been posted in the debug status register if enabled. No explicit enable bits are provided for watchpoints; they are always enabled by definition. During a debug session, events (except for DEVT1 and DEVT2) with a corresponding DBSR bit are blocked from asserting a watchpoint. The DEVNT-based watchpoints are not blocked during a debug session. If not desired for address-based events, the base address values for these events may be programmed to an unused system address. MSR[DE] has no effect on watchpoint generation.

External logic may monitor the assertion of these signals for debugging purposes. Watchpoints are signaled in the clock cycle following the occurrence of the actual event. The Nexus3 module also monitors assertion of these signals for various development control purposes (See Section 13.5, “Watchpoint Support”).

Table 13-28. Watchpoint Output Signal Assignments

Signal Name	Type	Description
jd_watchpt[0]	IAC1	Instruction Address Compare 1 watchpoint Asserted whenever an IAC1 compare occurs regardless of being enabled to set DBSR status
jd_watchpt[1]	IAC2	Instruction Address Compare 2 watchpoint Asserted whenever an IAC2 compare occurs regardless of being enabled to set DBSR status

Table 13-28. Watchpoint Output Signal Assignments (continued)

Signal Name	Type	Description
jd_watchpt[2]	IAC3	Instruction Address Compare 3 watchpoint Asserted whenever an IAC3 compare occurs regardless of being enabled to set DBSR status
jd_watchpt[3]	IAC4	Instruction Address Compare 4 watchpoint Asserted whenever an IAC4 compare occurs regardless of being enabled to set DBSR status
jd_watchpt[4]	DAC1[1]	Data Address Compare 1 watchpoint Asserted whenever a DAC1 compare occurs regardless of being enabled to set DBSR status
jd_watchpt[5]	DAC2[1]	Data Address Compare 2 watchpoint Asserted whenever a DAC2 compare occurs regardless of being enabled to set DBSR status
jd_watchpt[6]	DCNT1	Debug Counter 1 watchpoint Asserted whenever Debug Counter 1 decrements to zero regardless of being enabled to set DBSR status
jd_watchpt[7]	DCNT2	Debug Counter 2 watchpoint Asserted whenever Debug Counter 2 decrements to zero regardless of being enabled to set DBSR status
jd_watchpt[8]	IAC5	Instruction Address Compare 5 watchpoint Asserted whenever an IAC5 compare occurs regardless of being enabled to set DBSR status
jd_watchpt[9]	IAC6	Instruction Address Compare 6 watchpoint Asserted whenever an IAC6 compare occurs regardless of being enabled to set DBSR status
jd_watchpt[10]	DEVT1	Debug Event Input 1 watchpoint Asserted whenever a DEVT1 debug event occurs regardless of being enabled to set DBSR status
jd_watchpt[11]	DEVT2	Debug Event Input 2 watchpoint Asserted whenever a DEVT2 debug event occurs regardless of being enabled to set DBSR status
jd_watchpt[12]	DEVNT0	Debug Event Output 0 watchpoint Asserted whenever a ‘1’ is written to the bit of the DEVNT field of the DEVENT debug register corresponding to jd_watchpt[12]
jd_watchpt[13]	DEVNT1	Debug Event Output 1 watchpoint Asserted whenever a ‘1’ is written to the bit of the DEVNT field of the DEVENT debug register corresponding to jd_watchpt[13]
jd_watchpt[14]	IAC7	Instruction Address Compare 7 watchpoint Asserted whenever an IAC7 compare occurs regardless of being enabled to set DBSR status
jd_watchpt[15]	IAC8	Instruction Address Compare 8 watchpoint Asserted whenever an IAC8 compare occurs regardless of being enabled to set DBSR status

Table 13-28. Watchpoint Output Signal Assignments (continued)

Signal Name	Type	Description
jd_watchpt[16]	IRPT	Interrupt watchpoint Asserted whenever an IRPT debug event occurs regardless of being enabled to set DBSR status
jd_watchpt[17]	RET	Return watchpoint Asserted whenever a RET debug event occurs regardless of being enabled to set DBSR status
jd_watchpt[18]	CIRPT	Critical Interrupt watchpoint Asserted whenever a CIRPT debug event occurs regardless of being enabled to set DBSR status
jd_watchpt[19]	CRET	Critical Return watchpoint Asserted whenever a CRET debug event occurs regardless of being enabled to set DBSR status
jd_watchpt[20]	DEVNT2	Debug Event Output 2 watchpoint Asserted whenever a ‘1’ is written to the bit of the DEVNT field of the DEVENT debug register corresponding to jd_watchpt[20]
jd_watchpt[21]	DEVNT3	Debug Event Output 3 watchpoint Asserted whenever a ‘1’ is written to the bit of the DEVNT field of the DEVENT debug register corresponding to jd_watchpt[21]
jd_watchpt[22]	PMEVENT	Performance Monitor Event input watchpoint Asserted whenever **p_pm_event** transitions from a ‘0’ to a ‘1’
jd_watchpt[23]	PMC0	Performance Monitor Counter 0 watchpoint Asserted whenever PMC0 triggers an event based on PMLCa0[PMP]
jd_watchpt[24]	PMC1	Performance Monitor Counter 1 watchpoint Asserted whenever PMC1 triggers an event based on PMLCa1[PMP]
jd_watchpt[25]	PMC2	Performance Monitor Counter 2 watchpoint Asserted whenever PMC2 triggers an event based on PMLCa2[PMP]
jd_watchpt[26]	PMC3	Performance Monitor Counter 3 watchpoint Asserted whenever PMC3 triggers an event based on PMLCa3[PMP]
jd_watchpt[27]	DTC1	Data Trace Control Range 1 watchpoint Asserted whenever an access meets the conditions for DTC Range 1
jd_watchpt[28]	DTC2	Data Trace Control Range 2 watchpoint Asserted whenever an access meets the conditions for DTC Range 2
jd_watchpt[29]	DTC3	Data Trace Control Range 3 watchpoint Asserted whenever an access meets the conditions for DTC Range 3

[1] If the corresponding event is completely disabled in DBCR0, either load-type or store-type data accesses are allowed to generate watchpoints, otherwise watchpoints are generated only for the enabled conditions.

13.6 MMU and Cache Operation During Debug

Normal operation of the MMU may be modified during a debug session via the OnCE control register (OCR). A debug session begins when the CPU initially enters debug mode and ends when an OnCE command with GO + EXIT is executed, releasing the CPU for normal operation. If desired during a debug session, the debug firmware may disable the translation process and may substitute default values for the

Access Protection (UX, UR, UW, SX, SR, SW) bits, and values obtained from the OnCE control register for page attribute (VLE, W, I, M, G, E) bits normally provided by a matching TLB entry. In addition, no address translation is performed, and instead, a 1:1 mapping of effective to real addresses is performed.

When disabled during a debug session, no TLB miss or TLB-related storage interrupt conditions will occur. If the debugger desires to use the normal translation process, the MMU may be left enabled in the OnCE OCR, and normal translation (including the possibility of a TLB Miss or storage interrupt) will remain in effect.

The OCR control bits are used when debug mode is entered. Refer to the bit definitions in the OCR (Section 13.4.6.3, "e200 OnCE Control Register (OCR)," for more detail. When the MMU is disabled for instruction accesses (OCR[I_DMDIS]) or for data accesses (OCR[D_DMDIS]), substituted page attribute bits control operation on respective accesses initiated during debug. No address translation is performed; instead, a 1:1 mapping between effective and real addresses is performed for respective accesses.

13.7 Cache Array Access During Debug

The cache arrays may be read and written during debug mode via the CDACNTL and CDADATA debug registers. This functionality is described in detail in Section 9.19, "Cache Memory Access For Debug/Error Handling."

13.8 Basic Steps for Enabling, Using, and Exiting External Debug Mode

The following steps show one possible scenario for a debugger wishing to use the external debug facilities. Note that this simplified flow is intended to illustrate basic operations, but does not cover all potential methods in depth.

To enabling external debug mode and initializing debug registers, you can use the following procedure:

- The debugger should ensure that the **jd_en_once** control signal is asserted in order to enable OnCE operation
- Select the OCR and write a value to it in which OCR[DR] and OCR[WKUP] are set. The TAP controller must step through the proper states as outlined earlier. This step places the CPU in a debug state in which it is halted and awaiting single-step commands or a release to normal mode
- Scan out the value of the OSR to determine that the CPU clock is running and the CPU has entered the debug state. This can be done in conjunction with a read of the CPUSCR. The OSR is shifted out during the Shift_IR state. The CPUSCR is shifted out during the Shift_DR state. The debugger should save the scanned-out value of CPUSCR for later restoration.
- Select the DBCR0 register and update it with DBCR0[EDM] set
- Clear the DBSR status bits
- Write appropriate values to the DBCR0-6, IAC, DAC, and DBCNT registers. Note that the initial write to DBCR0 only affects the EDM bit, so the remaining portion of the register must now be initialized, keeping the EDM bit set.

At this point the system is ready to commence debug operations. Depending on the desired operation, different steps must occur, as follows.

- Optionally, set the OCR[I_DMDIS, D_DMDIS] control bits to ensure that no TLB misses will occur while performing the debug operations
- Optionally, ensure that the values entered into the MSR portion of the CPUSCR during the following steps cause interrupt to be disabled (clearing MSR[EE] and MSR[CE]). This ensures that external interrupt sources do not cause single-step errors.

To single-step the CPU, use the following procedure:

- Debugger scans in either a new or a previously saved value of the CPUSCR (with appropriate modification of the PC and IR as described in Section 13.4.9.2, “Control State Register (CTL)”), with a Go + Noexit OnCE Command value.
- The debugger scans out the OSR with “no-register selected” and Go cleared. It determines that the PCU has re-entered the debug state and that no ERR condition occurred

To return the CPU to normal operation (without disabling external debug mode), use the following procedure:

- The OCR[I_DMDIS, D_DMDIS] and OCR[DR] control bits should be cleared, leaving OCR[WKUP] set.
- The debugger restores the CPUSCR with a previously saved value of the CPUSCR (with appropriate modification of the PC and IR as described in Section 13.4.9.2, “Control State Register (CTL)”), with a Go + Exit OnCE command value.
- OCR[WKUP] may then be cleared.

To exit external debug mode, use the following procedure:

- The debugger should place the CPU in the debug state via the OCR[DR] with OCR[WKUP] asserted, scanning out and saving the CPUSCR.
- The debugger should write the DBCR0–6 registers as needed, likely clearing every enable except DBCR0[EDM].
- The debugger should write the DBSR to a cleared state.
- The debugger should re-write the DBCR0 with all bits including EDM cleared.
- The debugger should clear OCR[DR].
- The debugger restores the CPUSCR with the previously saved value of the CPUSCR (with appropriate modification of the PC and IR as described in Section 13.4.9.2, “Control State Register (CTL)”), with a Go + Exit OnCE Command value.
- OCR[WKUP] may then be cleared.

NOTE

These steps are meant by way of examples and are not meant to be an exact template for debugger operation.

13.9 Parallel Signature Unit

To support applications requiring system integrity checking during operation, the e200 core provides a parallel signature unit. The parallel signature unit can monitor the internal CPU read and write buses for data accesses and accumulate a pair of 32-bit MISR signatures of the values transferred over these buses for data accesses.

The primitive polynomial used is $P(X) = 1+X^{10}+X^{30}+X^{31}+X^{32}$. Values are accumulated based on an initially programmed seed value and are qualified based on active byte lanes of the CPU internal read and write buses (**p_d_data_in[0:63]**, **p_d_data_out[0:63]**) as indicated via the **p_d_tsiz[0:2], p_d_elsize[0:1]**, and **p_d_addr[29:31]** signals. Inactive byte lanes use a value of all zeros as input data to the MISRs.Note that for read data, the data returned from the cache or BIU is used directly from **p_d_data_in[0:63]** for accumulation. For write cycles, however, the data accumulated is based on the data that is written to the cache or BIU after it has been properly aligned and permuted according to the endian mode of the access. **p_d_data_out[0:63]** is not used directly; instead, the proper memory image is used.

If an external termination error (bus error) occurs on any accumulated read data, the returned read data is ignored, a value of all zeros is used instead, and the error is logged. External termination errors occurring on data writes are not logged, even though the data is accumulated, since the data driven by the CPU was valid.

No data is accumulated for transfer errors signaled due to TLB Error, Cache Parity Error, Byte Ordering Error, DSI or ISI due to permissions violations, or for Alignment Errors.

No accumulation occurs for cache control operations such as **dcba**, **dcbi**, **icbi**, **dcbf**, **dcbst**, **dcbt**, **icbt**, **dcbtst**, **dcbz**, **dcbtls**, **dcbtstls**, **dcblc** or for cache operations initiated via the mtspr L1CSR0 or L1FINV0.

The unit may be independently enabled for data read cycles and data write cycles, allowing for flexible usage. Software may also control accumulation of software provided values via a pair of update registers. In addition, a counter is provided for software use to monitor the number of beats of data which have been compressed.

Updates are performed when the parallel signature registers are initialized, when a qualified internal bus cycle is terminated, when a software update is performed via a high or low update register, and when the parallel signature high or low registers are written with a **mtdcr** instruction.

NOTE

Updates due to qualified bus transfers are suppressed for the duration of a debug session.

Figure 13-26 shows the PSU structure.

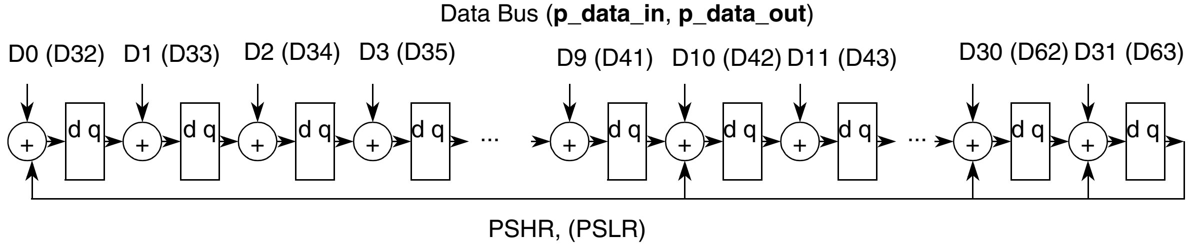


Figure 13-26. PSU Structure

The parallel signature unit consists of seven registers described in the following subsections. Access to these registers is privileged. No user-mode access is allowed.

NOTE

Proper access of the PSU registers requires that the **mfdcr** instruction which reads a PSU register be preceded by either an **mbar** or an **msync** instruction. To ensure that the effects of a **mtdcr** instruction to one of the PSU registers has taken effect, the **mtdcr** should be followed by a context synchronizing instruction (**sc**, **isync**, **rfi**, **rfci**, **rfdi**).

13.9.1 Parallel Signature Control Register (PSCR)

The parallel signature control register (PSCR), shown in Figure 13-27, controls operation of the parallel signature unit.

DCR 272

Access: Read/Write

	0–25	26	27–28	29	30	31
R W	—	CNTEN	—	RDEN	WREN	INIT
Reset	All zeros					

Figure 13-27. Parallel Signature Control Register (PSCR)

Table 13-29 describes the fields.

Table 13-29. PSCR Field Descriptions

Bits	Name	Description
0–25	—	Reserved
26	CNTEN	Counter Enable 0 Counter is disabled. 1 Counter is enabled. Counter is incremented on every accumulated transfer or on a **mtdcr psulr**,**Rn** instruction.
27–28	—	Reserved

Table 13-29. PSCR Field Descriptions (continued)

Bits	Name	Description
29	RDEN	Read Enable 0 Processor data read cycles are ignored. 1 Processor data reads cycles are accumulated. For inactive byte lanes, zeros are used for the data values.
30	WREN	Write Enable 0 Processor write cycles are ignored. 1 Processor write cycles are accumulated. For inactive byte lanes, zeros are used for the data values.
31	INIT	This bit may be written with a 1 to set the values in the PSHR, PSLR, and PSCTR registers to all 0s (0x00_0000_00). This bit always reads as 0.

13.9.2 Parallel Signature Status Register (PSSR)

The parallel signature status register (PSSR), shown in Figure 13-28, provides status relative to operation of the parallel signature unit.

DCR 273 Access: Read/Write

	0–30	31
R	—	TERR
W		
Reset	Unaffected	

Figure 13-28. Parallel Signature Status Register (PSSR)

Table 13-30 describes the fields.

Table 13-30. PSSR Field Descriptions

Bits	Name	Description
0–30	—	These bits are reserved
31	TERR	Transfer Error Status 0 No transfer error has occurred on accumulated read data since this bit was last cleared by software. 1 A transfer error has occurred on accumulated read data since this bit was last cleared by software. This bit indicates whether a transfer error has occurred on accumulated read data, and that the read data values returned were ignored and zeros are used instead. This bit is not cleared by hardware; only a software write of 1 to this bit clears it.

13.9.3 Parallel Signature High Register (PSHR)

The parallel signature high register (PSHR), shown in Figure 13-29, provides signature information for the high word (bits 0–31) of the internal read and write buses. It may be written via a **mtdcr pshr, Rs** instruction (DCR register 274) to initialize a seed value prior to enabling signature accumulation. The

PSCR[INIT] control bit may also be used to clear the PSHR. This register is unaffected by system reset, thus should be initialized by software prior to performing parallel signature operations.

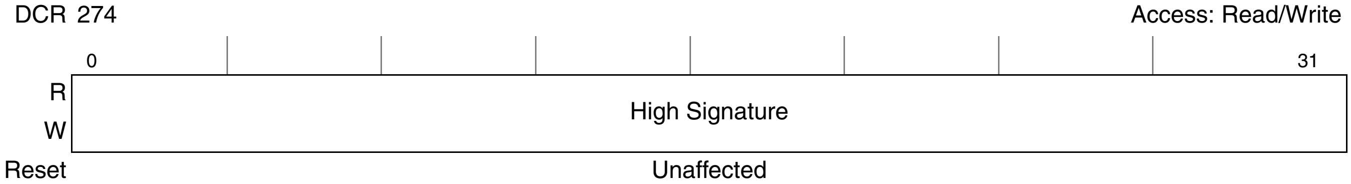


Figure 13-29. Parallel Signature High Register (PSHR)

13.9.4 Parallel Signature Low Register (PSLR)

The parallel signature low register (PSLR), shown in Figure 13-30, provides signature information for the low word (bits 32–63) of the internal read and write buses. It may be written via a **mtdcr pslr, Rs** instruction (DCR register 275) to initialize a seed value prior to enabling signature accumulation. The PSCR[INIT] control bit may also be used to clear the PSLR. This register is unaffected by system reset and should be initialized by software prior to performing parallel signature operations.

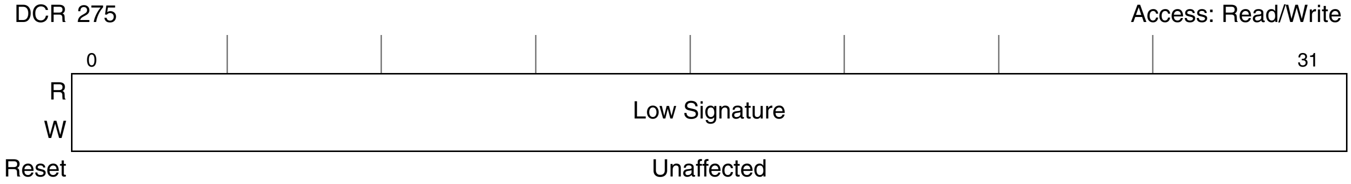


Figure 13-30. Parallel Signature Low Register (PSLR)

13.9.5 Parallel Signature Counter Register (PSCTR)

The parallel signature counter register (PSCTR), shown in Figure 13-31, provides count information for signature accumulation. The counter is incremented on every accumulated transfer or on a **mtdcr psulr,Rn** instruction. It may be written via a **mtdcr psctr, Rs** instruction (DCR register 276) to initialize a value prior to enabling signature accumulation. The PSCR[INIT] control bit may also be used to clear the PSCTR. This register is unaffected by system reset and should be initialized by software prior to performing parallel signature operations.

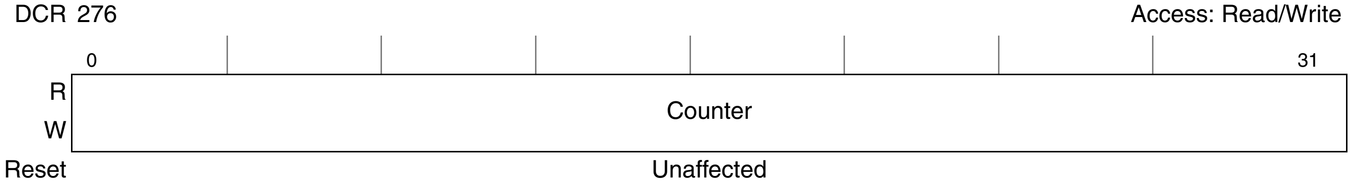


Figure 13-31. Parallel Signature Counter Register (PSCTR)

13.9.6 Parallel Signature Update High Register (PSUHR)

The parallel signature update high register (PSUHR), shown in Figure 13-32, enables using software to update the high signature value. It may be written via a **mtdcr psuhr, Rs** instruction (DCR register 277) to cause signature accumulation to occur in the parallel signature high register (PSHR) using the data value written. This register is write only; attempted reads return a value of all zeros. Writing to this register does not cause the PSCTR to increment.

DCR 277 Access: Write only

	0 31
R	
W	High Signature Update Data
Reset	Unaffected

Figure 13-32. Parallel Signature Update High Register (PSUHR)

13.9.7 Parallel Signature Update Low Register (PSULR)

The parallel signature update low register (PSULR), shown in Figure 13-33, enables using software to update the low signature value. It may be written via a **mtdcr psulr, Rs** instruction (DCR register 278) to cause signature accumulation to occur in the parallel signature low register (PSLR) using the data value written. This register is write only; attempted reads return a value of all zeros. Writing to this register also causes the PSCTR to increment.

DCR 278 Access: Write only

	0 31
R	
W	Low Signature Update Data
Reset	Unaffected

Figure 13-33. Parallel Signature Update Low Register (PSULR)

Chapter 14
Nexus 3 Module

This chapter defines the auxiliary pin functions, transfer protocols, and standard development features of a Class 3 device in compliance with the IEEE-ISTO 5001 standard. The development features supported are program trace, data trace, watchpoint messaging, ownership trace, data acquisition messaging, and read/write access via the JTAG interface. The Nexus 3 module also supports two Class 4 features: watchpoint triggering and processor overrun control.

14.1 Introduction

The e200z7 Nexus 3 module provides real-time development capabilities for the e200 processors in compliance with the IEEE-ISTO 5001 standard. This module provides development support capabilities without requiring the use of address and data pins for internal visibility.

A portion of the pin interface (the JTAG port) is also shared with the OnCE/Nexus 1 unit. The IEEE-ISTO 5001 standard defines an extensible auxiliary port that is used in conjunction with the JTAG port in e200 processors.

14.1.1 Terms and Definitions

Table 14-1 contains a set of terms and definitions associated with the Nexus 3 module.

Table 14-1. Terms and Definitions

Term	Description
IEEE-ISTO 5001-2008	Consortium and standard for real-time embedded system design. World wide Web documentation at http://www.ieee-isto.org/Nexus5001
Auxiliary Port	Refers to Nexus auxiliary port. Used as auxiliary port to the IEEE 1149.1 JTAG interface.
Branch Trace Messaging (BTM)	Visibility of addresses for taken branches and exceptions, and the number of sequential instructions executed between each taken branch.
Data Acquisition Messaging (DQM)	Allows code to be instrumented to export customized information to the Nexus auxiliary output port.
Data Read Message (DRM)	External visibility of data reads to memory-mapped resources.
Data Write Message (DWM)	External visibility of data writes to memory-mapped resources.
Data Trace Messaging (DTM)	External visibility of how data flows through the embedded system. This may include DRM and/or DWM.
JTAG Compliant	Device complying to IEEE 1149.1 JTAG standard

Table 14-1. Terms and Definitions (continued)

Term	Description
JTAG IR and DR Sequence	JTAG Instruction Register (IR) scan to load an opcode value for selecting a development register. The JTAG IR corresponds to the OnCE command register (OCMD). The selected development register is then accessed via a JTAG Data Register (DR) scan.
Nexus1	The e200 (OnCE) debug module. This module integrated with each e200 processor provides all static (core halted) debug functionality. This module is compliant with Class 1 of the IEEE-ISTO 5001 standard.
Ownership Trace Message (OTM)	Visibility of process/function that is currently executing.
Public Messages	Messages on the auxiliary pins for accomplishing common visibility and controllability requirements
SoC	“System-on-a-Chip”. SoC signifies all of the modules on a single die. This generally includes one or more processors with associated peripherals, interfaces & memory modules.
Standard	The phrase ‘according to the standard’ is used to indicate according to the IEEE-ISTO 5001-2001 standard.
Transfer Code (TCODE)	Message header that identifies the number and/or size of packets to be transferred, and how to interpret each of the packets.
Watchpoint	A data or instruction breakpoint or other debug event which does not cause the processor to halt. Instead, a pin is used to signal that the condition occurred. A watchpoint message may also be generated.

14.1.2 Feature List

The Nexus 3 module is compliant with Class 3 of the IEEE-ISTO 5001-2008 standard, with additional Class 4 features available. The following features are implemented:

- Program trace via branch trace messaging (BTM). Branch trace messaging displays program flow discontinuities (direct and indirect branches, exceptions, etc.), allowing the development tool to interpolate what transpires between the discontinuities. Thus static code may be traced.
- Data trace via data write messaging (DWM) and data read messaging (DRM). This provides the capability for the development tool to trace reads and/or writes to selected internal memory resources.
- Ownership trace via ownership trace messaging (OTM). OTM facilitates ownership trace by providing visibility into which process ID or operating system task is activated. An ownership trace message is transmitted when a new process/task is activated, allowing the development tool to trace ownership flow.
- Runtime access to embedded processor memory map via the JTAG port. This allows enhanced download/upload capabilities.
- Watchpoint messaging via the auxiliary pins
- Watchpoint trigger enable of program and/or data trace messaging
- Data acquisition messaging (DQM) allows code to be instrumented to export customized information to the Nexus auxiliary output port.

- Address translation messaging via program correlation messages displays updates to the TLB for use by the debugger in correlating virtual and physical address information.
- Auxiliary interface for higher data input/output
  - Configurable (min/max) Message Data Out pins (**nex_mdo[n:0]**)
  - One (1) or two (2) Message Start/End Out pins (**nex_mseo_b[1:0]**)
  - One (1) Read/Write Ready pin (**nex_rdy_b**) pin
  - One (1) Watchpoint Event output pin (**nex_evto_b**)
  - Three (3) additional Watchpoint Event output pins (**nex_wevto[2:0]**) for SoC use
  - One (1) Event In pin (**nex_evti_b**)
  - One (1) MCKO (Message Clock Out) pin
- Registers for program trace, data trace, ownership trace, and watchpoint trigger.
- All features controllable and configurable via the JTAG port

NOTE

The configuration of the message data out pins is controlled in the following ways:

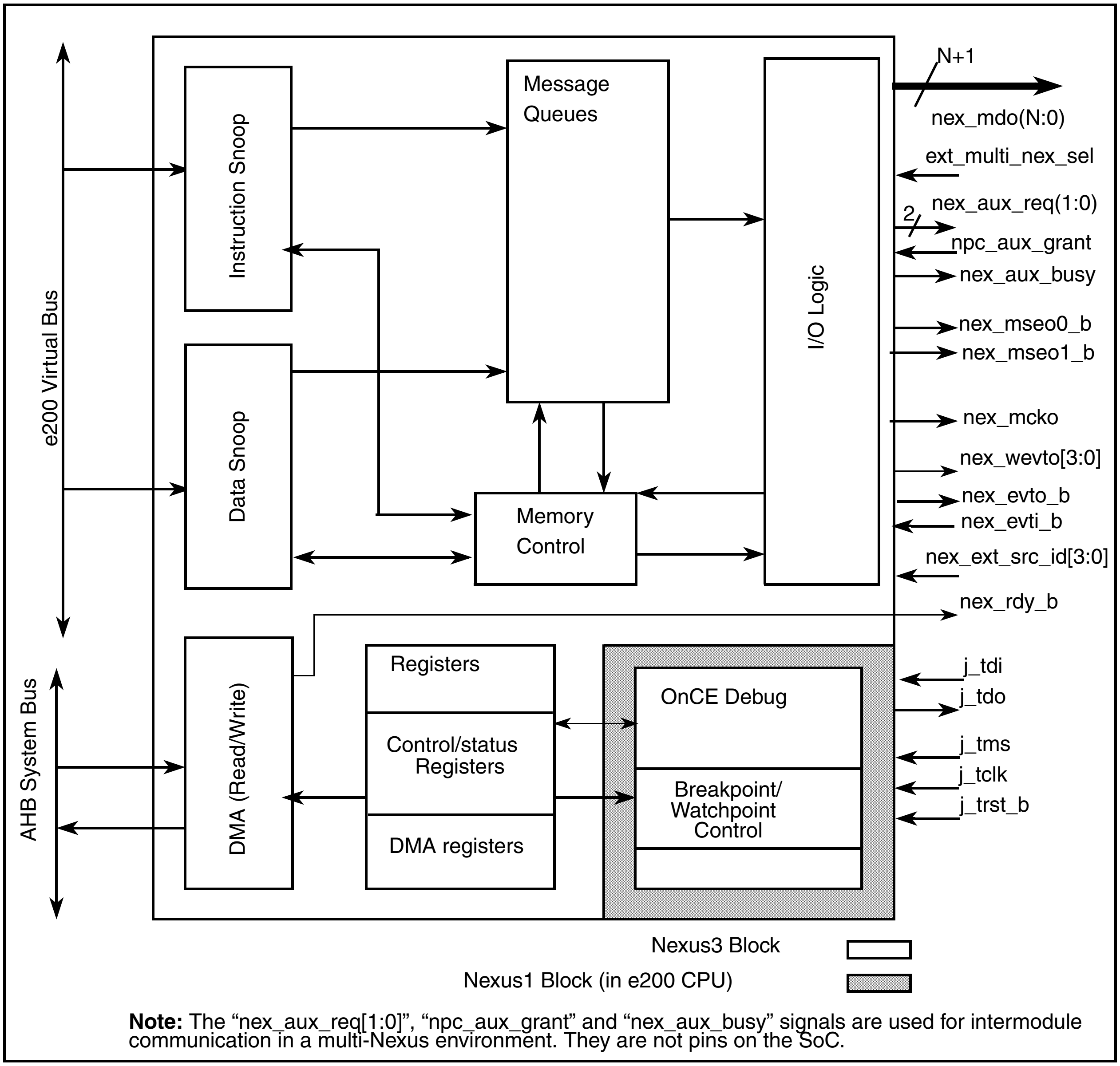
- For multi-Nexus implementations, by the port control register at the SoC level.
- For single Nexus implementations, by development control register 1 (DC1) within the Nexus 3 module.

Both implementations support full port mode (FPM), which uses the maximum number of MDO pins, and reduced port mode (RPM), which uses the minimum number of MDO pins. Do not change this setting while the system is running.

The configuration of the Message Start/End Out pins (1 or 2) is determined at the SoC integration level. This option is hardwired based on SoC bandwidth requirements.

14.1.3 Functional Block Diagram

Figure 14-1 shows the Nexus 3 functional block diagram.



Note: The “nex_aux_req[1:0]”, “npc_aux_grant” and “nex_aux_busy” signals are used for intermodule communication in a multi-Nexus environment. They are not pins on the SoC.

Figure 14-1. Nexus 3 Functional Block Diagram

14.2 Enabling Nexus 3 Operation

The Nexus module is enabled by loading a single instruction (*NEXUS3-ACCESS*) into the JTAG instruction register (IR) (OnCE OCMD register). For the Nexus 3 module, the OCMD value is 0b0001111100. Once enabled, the module is ready to accept control input via the JTAG/OnCE pins.

Enabling the Nexus 3 module automatically enables the generation of debug status messages.

The Nexus module is disabled when the JTAG state machine reaches the Test-Logic-Reset state. This state can be reached by the assertion of the **j_trst_b** pin or by cycling through the state machine using the **j_tms** pin. The Nexus module is also disabled if a Power-on-Reset (POR) event occurs. If the Nexus 3 module is disabled, no trace output is provided, and the module disables (drive inactive) auxiliary port output pins (**nex_mdo[n:0]**, **nex_mseo[1:0]**, **nex_mcko**). Nexus registers will not be available for reads or writes.

NOTE

Please refer to the “Nexus 3 Integration Guide” for details on IEEE-ISTO 5001 standard compliance with regards to output pins and multiple Nexus module configurations.

14.3 TCODEs Supported

The Nexus 3 pins allow flexible transfer operations via public messages. A TCODE defines the transfer format, the number and/or size of the packets to be transferred, and the purpose of each packet. The IEEE-ISTO 5001 standard defines a set of public messages and allocates additional TCODEs for vendor-specific features outside the scope of the public messages. The Nexus 3 block supports the TCODEs shown in Table 14-2.

Table 14-2. Supported TCODEs

Message Name	Min. Field Size (bits)	Max. Field Size (bits)	Field Name	Field Type	Field Description
Debug Status	6	6	TCODE	Fixed	TCODE number = 0
	4	4	SRC	Fixed	Source processor identifier
	8	8	STATUS	Fixed	Debug status register (DS[31–24])
Ownership Trace Message	6	6	TCODE	Fixed	TCODE number = 2
	4	4	SRC	Fixed	Source processor identifier
	1	12	PROCESS	Variable	Task/Process ID tag
Program Trace Direct Branch Message[1]	6	6	TCODE	Fixed	TCODE number = 3
	4	4	SRC	Fixed	Source processor identifier
	1	8	ICNT	Variable	# sequential instructions completed since last predicate instruction, transmitted instruction count, or taken change of flow
Program Trace Indirect Branch Message[1]	6	6	TCODE	Fixed	TCODE number = 4
	4	4	SRC	Fixed	Source processor identifier
	1	1	MAP	Fixed	Address Space (IS) indicator
	1	8	ICNT	Variable	# sequential instructions completed since last predicate instruction, transmitted instruction count, or taken change of flow
	1	32	U-ADDR	Variable	Unique part of target address for taken branches/exceptions

Table 14-2. Supported TCODEs (continued)

Message Name	Min. Field Size (bits)	Max. Field Size (bits)	Field Name	Field Type	Field Description
Data Trace Data Write Message	6	6	TCODE	Fixed	TCODE number = 5
	4	4	SRC	Fixed	Source processor identifier
	1	1	MAP	Fixed	Address Space (DS) indicator
	4	4	DSZ	Fixed	Data size (Refer to Table 14-7)
	1	32	U-ADDR	Variable	Unique portion of the data write address
	1	64	DATA	Variable	data write value(s) (see Data Trace section for details)
Data Trace Data Read Message	6	6	TCODE	Fixed	TCODE number = 6
	4	4	SRC	Fixed	Source processor identifier
	1	1	MAP	Fixed	Address Space (DS) indicator
	4	4	DSZ	Fixed	Data size (Refer to Table 14-7)
	1	32	U-ADDR	Variable	Unique portion of the data read address
	1	64	DATA	Variable	Data read value(s) (see Data Trace section for details)
Data Acquisition Message	6	6	TCODE	Fixed	TCODE number = 7
	4	4	SRC	Fixed	Source processor identifier
	8	8	DQTAG	Fixed	Identification tag taken from $\text{DEVENT}_{\text{DQTAG}}$ register field
	1	32	DQDATA	Variable	Exported data taken from DDAM register
Error Message	6	6	TCODE	Fixed	TCODE number = 8
	4	4	SRC	Fixed	Source processor identifier
	4	4	ETYPE	Fixed	Error type
	8	8	ECODE	Fixed	Error code
Program Trace Direct Branch Message with Sync[1]	6	6	TCODE	Fixed	TCODE number = 11
	4	4	SRC	Fixed	Source processor identifier
	1	8	ICNT	Variable	# sequential instructions completed since last predicate instruction, transmitted instruction count, or taken change of flow
	1	32	F-ADDR	Variable	Full target address (leading zeros truncated)
Program Trace Indirect Branch Message with Sync[1]	6	6	TCODE	Fixed	TCODE number = 12
	4	4	SRC	Fixed	Source processor identifier
	1	1	MAP	Fixed	Address Space (IS) indicator
	1	8	ICNT	Variable	# sequential instructions completed since last predicate instruction, transmitted instruction count, or taken change of flow
	1	32	F-ADDR	Variable	Full target address (leading zeros truncated)

Table 14-2. Supported TCODEs (continued)

Message Name	Min. Field Size (bits)	Max. Field Size (bits)	Field Name	Field Type	Field Description
Data Trace Data Write Message with Sync	6	6	TCODE	Fixed	TCODE number = 13
	4	4	SRC	Fixed	Source processor identifier
	1	1	MAP	Fixed	Address Space (DS) indicator
	4	4	DSZ	Fixed	Data size (Refer to Table 14-7)
	1	32	F-ADDR	Variable	Full access address (leading zeros truncated)
	1	64	DATA	Variable	Data write value(s) (seeSection 14.12, “Data Trace)
Data Trace Data Read Message with Sync	6	6	TCODE	Fixed	TCODE number = 14
	4	4	SRC	Fixed	Source processor identifier
	1	1	MAP	Fixed	Address Space (DS) indicator
	4	4	DSZ	Fixed	Data size (Refer to Table 14-7)
	1	32	F-ADDR	Variable	Full access address (leading zeros truncated)
	1	64	DATA	Variable	Data read value(s) (see Section 14.12, “Data Trace)
Watchpoint Message	6	6	TCODE	Fixed	TCODE number = 15
	4	4	SRC	Fixed	Source processor identifier
	1	32	WPHIT	Variable	Field indicating watchpoint source(s) (leading zeros truncated)
Resource Full Message	6	6	TCODE	Fixed	TCODE number = 27
	4	4	SRC	Fixed	Source processor identifier
	4	4	RCODE	Fixed	Resource code (Refer to Table 14-5) indicates which resource is the cause of this message
	1	32	RDATA	Variable	Branch/predicate instruction history (see Section 14.11.3.4, “Resource Full Messages”)
Program Trace Indirect Branch History Message	6	6	TCODE	Fixed	TCODE number = 28 (see Note below)
	4	4	SRC	Fixed	Source processor identifier
	1	1	MAP	Fixed	Address Space (IS) indicator
	1	8	I-CNT	Variable	# sequential instructions completed since last predicate instruction, transmitted instruction count, or taken change of flow
	1	32	U-ADDR	Variable	Unique part of target address for taken branches/exceptions
	1	32	HIST	Variable	Branch/predicate instruction history (see Section 14.11.1, “Branch Trace Messaging Types”)

Table 14-2. Supported TCODEs (continued)

Message Name	Min. Field Size (bits)	Max. Field Size (bits)	Field Name	Field Type	Field Description
Program Trace Indirect Branch History Message with Sync	6	6	TCODE	Fixed	TCODE number = 29 (see Note below)
	4	4	SRC	Fixed	Source processor identifier
	1	1	MAP	Fixed	Address Space (IS) indicator
	1	8	I-CNT	Variable	# sequential instructions completed since last predicate instruction, transmitted instruction count, or taken change of flow
	1	32	F-ADDR	Variable	Full target address (leading zero (0) truncated)
	1	32	HIST	Variable	Branch/predicate instruction history (see Section 14.11.1, “Branch Trace Messaging Types”)
Program Trace Program Correlation Message	6	6	TCODE	Fixed	TCODE number = 33
	4	4	SRC	Fixed	Source processor identifier
	4	4	EVCODE	Fixed	Event correlated w/ program flow (Refer to Table 14-6)
	2	2	CDF	Fixed	# fields of information in CDATA. 00 Reserved 01 One field (CDATA1) (reserved) 10 Two fields (CDATA1 + CDATA2) 11 Three fields (CDATA1 + CDATA2 + CDATA3)
	1	8	I-CNT	Variable	# sequential instructions completed since last predicate instruction, transmitted instruction count, or taken change of flow
	1	32	CDATA1	Variable	Correlation data field 1: [branch/predicate instruction history or TLB info part1] (see Section 14.11.3.5, “Program Correlation Messages”)
	0	32	CDATA2	Variable	Correlation data field 2: PID/IS info or TLB info (F-ADDR_V for virtual address or tlbivax EA) (see Section 14.11.3.5, “Program Correlation Messages”)
	0	32	CDATA3	Variable	Correlation data field 3: TLB info -ADDR_P for physical address (see Section 14.11.3.5, “Program Correlation Messages”)

[1] If the branch history method is selected, this TCODE is not messaged out.

NOTE

Program trace can be implemented using either branch history/predicate instruction messages or traditional direct/indirect branch messages. The user can select between the two types of program trace. The advantages for each are discussed in Section 14.11.1, “Branch Trace Messaging Types.”

Table 14-3 shows the error code encodings used when reporting an error via the Nexus 3 error message.

Table 14-3. Error Code Encoding (TCODE = 8)

Error Code	Description
xxxxxxx1	Watchpoint Trace Message(s) Lost
xxxxxx1x	Data Trace Message(s) Lost
xxxxx1xx	Program Trace Message(s) Lost
xxxx1xxx	Ownership Trace Message(s) Lost
xxx1xxxx	Status Message(s) Lost (Debug Status messages, etc.)
xx1xxxxx	Data Acquisition Message(s) Lost
x1xxxxxx	Reserved
1xxxxxxx	Reserved

Table 14-4 shows the error type encodings used when reporting an error via the Nexus 3 error message.

Table 14-4. Error Type Encoding (TCODE = 8)

Error Type	Description
0000	Message queue overrun caused one or more messages to be lost
0001	Contention with higher priority messages caused one or more messages to be lost
0010	Reserved
0011	Read/write access error
0100	Reserved
0101	Invalid access opcode (Nexus Register unimplemented)
0110–1111	Reserved

Table 14-5 shows the encodings used for resource codes for certain messages.

Table 14-5. RCODE values (TCODE = 27)

Resource Code	Description
0000	Program trace instruction counter reached 255 and was reset.
0001	Program trace, branch/predicate instruction history full. This type of packet is terminated by a stop bit set to 1 after the last history bit.

Table 14-6 shows the event code encodings used for certain messages.

Table 14-6. Event Code Encoding (TCODE = 33)

Event Code	Description
0000	Entry into Debug Mode
0001	Entry into Low Power Mode (CPU only)

Table 14-6. Event Code Encoding (TCODE = 33) (continued)

Event Code	Description
0010–0011	Reserved for future functionality
0100	Disabling Program Trace
0101	New process ID value is established in PID0 via **mtspr PID0**, or new value for MSR_{IS} is established via a **mtmsr** instruction
0110–1000	Reserved for future functionality
1001	Begin masking of program trace messages due to MSR_{PMM}=0 and $\text{DC4}_{\text{PTMARK}}$=1
1010	Branch and link occurrence (direct branch function call)
1011	New Address Translation established in the TLB via **tlbwe**
1100	Address Translation entries invalidated in the TLB via **tlbivax**
1101	Reserved for future functionality
1110	End of Power ISA tracing (trace disable or entry into a VLE page from a non-VLE page)
1111	End of VLE tracing (trace disabled or entry into a non-VLE page from a VLE page)

Table 14-7 shows the data trace size encodings used for certain messages.

Table 14-7. Data Trace Size Encodings (TCODE = 5,6,13,14)

DTM Size Encoding	Transfer Size
0000	0—no data
0001	Byte
0010	Half word (2 bytes)
0011	Three bytes
0100	Word (4 bytes)
0101	Five bytes
0110	Six bytes
0111	Seven bytes
1000	Double word (8 bytes)
1001–1111	Reserved

14.4 Nexus 3 Programmer’s Model

This section describes the Nexus 3 programmers model. Nexus 3 registers are accessed using the JTAG/OnCE port in compliance with IEEE 1149.1. See Section 14.5, “JTAG/OnCE Nexus 3 Register Access,” for details on Nexus 3 register access.

NOTE

Nexus 3 registers and output signals are numbered using bit 0 as the least significant bit. This bit ordering is consistent with the ordering defined by the IEEE-ISTO 5001–2008 standard.

Table 14-8 details the register map for the Nexus 3 module.

Table 14-8. Nexus 3 Register Map

Nexus Register	Nexus Access Opcode	Read/ Write	Read Address	Write Address
Client Select Control (CSC)[1]	0x1	R	0x02	—
Port Configuration Register (PCR)[1]	PCR_INDEX[2]	R/W	—	—
Development Control 1 (DC1)	0x2	R/W	0x04	0x05
Development Control 2 (DC2)	0x3	R/W	0x06	0x07
Development Control 3 (DC3)	0x4	R/W	0x08	0x09
Development Control 4 (DC4)	0x5	R/W	0x0A	0x0B
Read/Write Access Control/Status (RWCS)	0x7	R/W	0x0E	0x0F
Read/Write Access Address (RWA)	0x9	R/W	0x12	0x13
Read/Write Access Data (RWD)	0xA	R/W	0x14	0x15
Watchpoint Trigger (WT)	0xB	R/W	0x16	0x17
Reserved	0xC	R/W	0x18	0x19
Data Trace Control (DTC)	0xD	R/W	0x1A	0x1B
Data Trace Start Address 1 (DTSA1)	0xE	R/W	0x1C	0x1D
Data Trace Start Address 2 (DTSA2)	0xF	R/W	0x1E	0x1F
Data Trace Start Address 3 (DTSA3)	0x10	R/W	0x20	0x21
Data Trace Start Address 4 (DTSA4)	0x11	R/W	0x22	0x23
Data Trace End Address 1 (DTEA1)	0x12	R/W	0x24	0x25
Data Trace End Address 2 (DTEA2)	0x13	R/W	0x26	0x27
Data Trace End Address 3 (DTEA3)	0x14	R/W	0x28	0x29
Data Trace End Address 4 (DTEA4)	0x15	R/W	0x2A	0x2B
Reserved	0x16–0x2F	—	0x28–0x5E	0x29–5F
Development Status (DS)	0x30	R	0x60	—
Reserved	0x31	R/W	0x62	0x63
Overrun Control (OVCR)	0x32	R/W	0x64	0x65
Watchpoint Mask (WMSK)	0x33	R/W	0x66	0x67
Reserved	0x34	—	0x68	0x69
Program Trace Start Trigger Control (PTSTC)	0x35	R/W	0x6A	0x6B

Table 14-8. Nexus 3 Register Map (continued)

Nexus Register	Nexus Access Opcode	Read/ Write	Read Address	Write Address
Program Trace End Trigger Control (PTETC)	0x36	R/W	0x6C	0x6D
Data Trace Start Trigger Control (DTSTC)	0x37	R/W	0x6E	0x6F
Data Trace End Trigger Control (DTETC)	0x38	R/W	0x70	0x71
Reserved	0x39–0x3F	—	0x72–0x7E	0x73–7F

[1] The CSC and PCR registers are shown in this table as part of the Nexus programmer's model. They are only present at the top level SoC Nexus controller in a multi-Nexus implementation, not in the Nexus 3 module. The SoC's CSC Register is readable through Nexus, but the PCR is shown for reference only here.

2 The "PCR_INDEX" is a parameter determined by the SoC. Refer to the "e200 Nexus 3 Integration Guide" for more information on how this parameter is implemented for each Nexus module.

14.4.1 Client Select Control (CSC)

The CSC register, shown in Figure 14-2, determines which Nexus client is under development. This register is present at the top-level SoC Nexus 3 controller to select one of multiple on-chip Nexus 3 units.

Nexus Reg 0x1 Access: Read only

	7	6	5	4	3	2	1	0
R	—				CS			
W								
Reset	All zeros							

Figure 14-2. Client Select Control Register

Table 14-9 shows the client select control register fields.

Table 14-9. Client Select Control Register Fields

Bits	Name	Description
CSC[7–4]	—	Reserved for future Nexus Clients (read as 0)
CSC[3–0]	CSC	Client Select Control 0xX—Nexus client (SoC level)

14.4.2 Port Configuration Register (PCR)—reference only

The port configuration register (PCR), shown in Figure 14-13, controls the basic port functions for all Nexus modules in a multi-Nexus environment. This includes clock control and auxiliary port width. All bits in this register are writable only once after system reset.

Nexus Reg PCR_INDEX

Access: Read/Write

	31	30	29	28 26	25 0
R W	OPC	—	MCK_EN	MCK_DIV	—
Reset	All zeros				

Figure 14-3. Port Configuration Register (PCR)

Table 14-10 shows the port configuration register fields.

Table 14-10. Port Configuration Register Fields

Bits	Name	Description
31	OPC	OPC–Output Port Mode Control (SoC level) 0 Reduced Port Mode configuration (min# **nex_mdo[n:0]** pins defined by SoC) 1 Full Port Mode configuration (max# **nex_mdo[n:0]** pins defined by SoC)
30	—	Reserved for future functionality
29	MCK_EN	MCK_EN–MCKO Clock Enable (SoC Level) 0 **nex_mcko** is disabled 1 **nex_mcko** is enabled
28:26	MCK_DIV	MCK_DIV–MCKO Clock Divide Ratio (see note below) (SoC Level) 000 **nex_mcko** is 1× processor clock frequency. 001 **nex_mcko** is ½× processor clock frequency. 010 Reserved (default to ½× processor clock frequency.) 011 **nex_mcko** is ¼× processor clock frequency. 100–110 Reserved (default to ½× processor clock frequency.) 111 **nex_mcko** is $\frac{1}{8}$× processor clock frequency.
25:0	—	Reserved for future functionality

NOTE

The CSC and PCR registers exist in a separate module at the SoC level in a multi-Nexus environment. If the e200 Nexus 3 module is the only Nexus module, these registers are not implemented and the e200 Nexus 3 defined development control register 1 (DC1) is used to control the SoC-level Nexus port functionality.

14.4.3 Nexus Development Control Register 1 (DC1)

Nexus development control register 1 is used to control the basic development features of the Nexus 3 module. Figure 14-4 shows development control register 1.

Nexus 0x2
Reg #

Access: Read/Write

	31	30 29	28	27	26 ... 15	14	13 12	11 10	9 8	7 6	5 ... 0
R W	OPC	MCK_DIV	—	PTM	—	POTD	TSEN	EOC	EIC	—	TM
Reset	All zeros										

Figure 14-4. Development Control Register 1

Table 14-11 describes its fields.

Table 14-11. Development Control Register 1 Fields

Bits	Name	Description
31	OPC	OPC—Output Port Mode Control 0 Reduced Port Mode configuration (min# **nex_mdo[n:0]** pins defined 1 Full Port Mode configuration (max# **nex_mdo[n:0]** pins defined
30–29	MCK_DIV	MCK_DIV—MCKO Clock Divide Ratio (see note below) 00 **nex_mcko** is 1× processor clock frequency. 01 **nex_mcko** is ½× processor clock frequency. 10 **nex_mcko** is ¼× processor clock frequency. 11 **nex_mcko** is $\frac{1}{8}$ × processor clock frequency.
28	—	Reserved for future functionality
27	PTM	PTM—Program Trace Method 0 Program Trace uses traditional branch messages. 1 Program Trace uses branch history messages.
26–15	—	Reserved for future functionality
14	POTD	Periodic Ownership Trace Disable 0 Periodic ownership trace message events are enabled. 1 Periodic ownership trace message events are disabled.
13–12	TSEN	Timestamp Enable - (not implemented, write to 00) 00 Timestamp is disabled
11–10	EOC	EOC—EVTO Control 00 **nex_evto_b** upon occurrence of watchpoints (configured in DC2 and DC3) 01 **nex_evto_b** upon entry into debug mode 1x Reserved
9–8	EIC	EIC—EVTI Control 00 **nex_evti_b** is used for synchronization (program trace/data trace) 01 **nex_evti_b** is used for debug request 1X Reserved

Table 14-11. Development Control Register 1 Fields (continued)

7–6	—	Reserved for future functionality
5–0	TM	Trace Mode[1] 000000 All trace disabled XXXXX1 Ownership trace enabled XXXX1X Data trace enabled XXX1XX Program trace enabled XX1XXX Watchpoint trace enabled X1XXXX Reserved 1XXXXX Data acquisition trace enabled

[1] This field may be updated by hardware in response to watchpoint triggering. Writes to this field take precedence over hardware updates in the event of a collision. Refer to Section 14.4.8, "Watchpoint Trigger Registers (WT, PTSTC, PTETC, DTSTC, DTETC)," for more information on watchpoint triggering.

NOTE

The output port mode control bit (OPC) and MCKO clock divide ratio bits (MCK_DIV) must only be modified during system reset or debug mode to insure correct output port and output clock functionality. It is also recommended that all other bits of the DC1 only be modified in one of these two modes.

14.4.4 Nexus Development Control Register 2 (DC2)

Nexus development control registers 2 and 3 are used to control output signaling on the Nexus 3 module. Table 13-28 lists the watchpoints.

Figure 14-5 shows development control register 2.

Nexus Reg 0x3

Access: Read/Write

	31–28	27–24	23–20	19–16	15–0
R W	—	WEVTO[2]C	WEVTO[1]C	WEVTO[0]C	EWC
Reset	All zeros				

Figure 14-5. Development Control Register 2 (DC2)

Table 14-12 describes its fields.

Table 14-12. Development Control Register 2 Fields

Bits	Name	Description
31–28	—	Reserved

Table 14-12. Development Control Register 2 Fields (continued)

Bits	Name	Description
27–24	WEVTO[2]C	WEVTO[2]C- Watchpoint Event Out 2 Configuration 0000—No Watchpoints #0–14 trigger **nex_wevto[2]** 0001—Watchpoint #0 triggers **nex_wevto[2]** 0010—Watchpoint #1 triggers **nex_wevto[2]** 0011—Watchpoint #2 triggers **nex_wevto[2]** 0100—Watchpoint #3 triggers **nex_wevto[2]** 0101—Watchpoint #4 triggers **nex_wevto[2]** 0110—Watchpoint #5 triggers **nex_wevto[2]** 0111—Watchpoint #6 triggers **nex_wevto[2]** 1000—Watchpoint #7 triggers **nex_wevto[2]** 1001—Watchpoint #8 triggers **nex_wevto[2]** 1010—Watchpoint #9 triggers **nex_wevto[2]** 1011—Watchpoint #10 triggers **nex_wevto[2]** 1100—Watchpoint #11 triggers **nex_wevto[2]** 1101—Watchpoint #12 triggers **nex_wevto[2]** 1110—Watchpoint #13 triggers **nex_wevto[2]** 1111—Watchpoint #14 triggers **nex_wevto[2]**
23–20	WEVTO[1]C	WEVTO[1]C- Watchpoint Event Out 1 Configuration 0000—No Watchpoints #0–14 trigger **nex_wevto[1]** 0001—Watchpoint #0 triggers **nex_wevto[1]** 0010—Watchpoint #1 triggers **nex_wevto[1]** 0011—Watchpoint #2 triggers **nex_wevto[1]** 0100—Watchpoint #3 triggers **nex_wevto[1]** 0101—Watchpoint #4 triggers **nex_wevto[1]** 0110—Watchpoint #5 triggers **nex_wevto[1]** 0111—Watchpoint #6 triggers **nex_wevto[1]** 1000—Watchpoint #7 triggers **nex_wevto[1]** 1001—Watchpoint #8 triggers **nex_wevto[1]** 1010—Watchpoint #9 triggers **nex_wevto[1]** 1011—Watchpoint #10 triggers **nex_wevto[1]** 1100—Watchpoint #11 triggers **nex_wevto[1]** 1101—Watchpoint #12 triggers **nex_wevto[1]** 1110—Watchpoint #13 triggers **nex_wevto[1]** 1111—Watchpoint #14 triggers **nex_wevto[1]**

Table 14-12. Development Control Register 2 Fields (continued)

Bits	Name	Description
19–16	WEVTO[0]C	WEVTO[0]C- Watchpoint Event Out 0 Configuration 0000—No Watchpoints #0–14 trigger **nex_wevto[0]** 0001—Watchpoint #0 triggers **nex_wevto[0]** 0010—Watchpoint #1 triggers **nex_wevto[0]** 0011—Watchpoint #2 triggers **nex_wevto[0]** 0100—Watchpoint #3 triggers **nex_wevto[0]** 0101—Watchpoint #4 triggers **nex_wevto[0]** 0110—Watchpoint #5 triggers **nex_wevto[0]** 0111—Watchpoint #6 triggers **nex_wevto[0]** 1000—Watchpoint #7 triggers **nex_wevto[0]** 1001—Watchpoint #8 triggers **nex_wevto[0]** 1010—Watchpoint #9 triggers **nex_wevto[0]** 1011—Watchpoint #10 triggers **nex_wevto[0]** 1100—Watchpoint #11 triggers **nex_wevto[0]** 1101—Watchpoint #12 triggers **nex_wevto[0]** 1110—Watchpoint #13 triggers **nex_wevto[0]** 1111—Watchpoint #14 triggers **nex_wevto[0]**
15–0	EWC	EWC—EVTO Watchpoint Configuration[1] 0000000000000000—No Watchpoints #0–15 trigger **nex_evto_b** XXXXXXXXXXXXXXX1—Watchpoint #0 triggers **nex_evto_b** XXXXXXXXXXXXXX1X—Watchpoint #1 triggers **nex_evto_b** XXXXXXXXXXXXX1XX—Watchpoint #2 triggers **nex_evto_b** XXXXXXXXXXXX1XXX—Watchpoint #3 triggers **nex_evto_b** XXXXXXXXXXX1XXXX—Watchpoint #4 triggers **nex_evto_b** XXXXXXXXXX1XXXXX—Watchpoint #5 triggers **nex_evto_b** XXXXXXXXX1XXXXXX—Watchpoint #6 triggers **nex_evto_b** XXXXXXXX1XXXXXXX—Watchpoint #7 triggers **nex_evto_b** XXXXXXX1XXXXXXXX—Watchpoint #8 triggers **nex_evto_b** XXXXXX1XXXXXXXXX—Watchpoint #9 triggers **nex_evto_b** XXXXX1XXXXXXXXXX—Watchpoint #10 triggers **nex_evto_b** XXXX1XXXXXXXXXXX—Watchpoint #11 triggers **nex_evto_b** XXX1XXXXXXXXXXXX—Watchpoint #12 triggers **nex_evto_b** XX1XXXXXXXXXXXXX—Watchpoint #13 triggers **nex_evto_b** X1XXXXXXXXXXXXXX—Watchpoint #14 triggers **nex_evto_b** 1XXXXXXXXXXXXXXX—Watchpoint #15 triggers **nex_evto_b**

[1] The EOC bits in DC1 must be programmed to trigger $\overline{\text{EVTO}}$ on Watchpoint occurrence for the EWC bits to have any effect.

14.4.5 Nexus Development Control Register 3 (DC3)

Figure 14-6 shows development control register 3.

Nexus Reg 0x4

Access: Read/Write

	31 28	27 24	23 20	19 16	15 11	10 0
R W	—	WEVTO[2]C	WEVTO[1]C	WEVTO[0]C	—	EWC
Reset	All zeros					

Figure 14-6. Development Control Register 3 (DC3)

Table 14-13 describes the fields.

Table 14-13. Development Control Register 3 Fields

Bits	Name	Description
31–28	—	Reserved
27–24	WEVTO[2]C	WEVTO[2]C—Watchpoint Event Out 2 Configuration 0000 No Watchpoints #15–#26 trigger **nex_wevto[2]** 0001 Watchpoint #15 triggers **nex_wevto[2]** 0010 Watchpoint #16 triggers **nex_wevto[2]** 0011 Watchpoint #17 triggers **nex_wevto[2]** 0100 Watchpoint #18 triggers **nex_wevto[2]** 0101 Watchpoint #19 triggers **nex_wevto[2]** 0110 Watchpoint #20 triggers **nex_wevto[2]** 0111 Watchpoint #21 triggers **nex_wevto[2]** 1000 Watchpoint #22 triggers **nex_wevto[2]** 1001 Watchpoint #23 triggers **nex_wevto[2]** 1010 Watchpoint #24 triggers **nex_wevto[2]** 1011 Watchpoint #25 triggers **nex_wevto[2]** 1100 Watchpoint #26 triggers **nex_wevto[2]** 1101–1111 Reserved
23–20	WEVTO[1]C	WEVTO[1]C—Watchpoint Event Out 1 Configuration 0000 No Watchpoints #15–#26 trigger **nex_wevto[1]** 0001 Watchpoint #15 triggers **nex_wevto[1]** 0010 Watchpoint #16 triggers **nex_wevto[1]** 0011 Watchpoint #17 triggers **nex_wevto[1]** 0100 Watchpoint #18 triggers **nex_wevto[1]** 0101 Watchpoint #19 triggers **nex_wevto[1]** 0110 Watchpoint #20 triggers **nex_wevto[1]** 0111 Watchpoint #21 triggers **nex_wevto[1]** 1000 Watchpoint #22 triggers **nex_wevto[1]** 1001 Watchpoint #23 triggers **nex_wevto[1]** 1010 Watchpoint #24 triggers **nex_wevto[1]** 1011 Watchpoint #25 triggers **nex_wevto[1]** 1100 Watchpoint #26 triggers **nex_wevto[1]** 1101–1111 Reserved

Table 14-13. Development Control Register 3 Fields (continued)

Bits	Name	Description
19–16	WEVTO[0]C	WEVTO[0]C—Watchpoint Event Out 0 Configuration 0000 No Watchpoints #15–#26 trigger **nex_wevto[0]** 0001 Watchpoint #15 triggers **nex_wevto[0]** 0010 Watchpoint #16 triggers **nex_wevto[0]** 0011 Watchpoint #17 triggers **nex_wevto[0]** 0100 Watchpoint #18 triggers **nex_wevto[0]** 0101 Watchpoint #19 triggers **nex_wevto[0]** 0110 Watchpoint #20 triggers **nex_wevto[0]** 0111 Watchpoint #21 triggers **nex_wevto[0]** 1000 Watchpoint #22 triggers **nex_wevto[0]** 1001 Watchpoint #23 triggers **nex_wevto[0]** 1010 Watchpoint #24 triggers **nex_wevto[0]** 1011 Watchpoint #25 triggers **nex_wevto[0]** 1100 Watchpoint #26 triggers **nex_wevto[0]** 1101–1111 Reserved
15–11	—	Reserved for watchpoint expansion
10–0	EWC	EWC—EVTO Watchpoint Configuration[1] 00000000000000—No Watchpoints #16–#26 trigger **nex_evto_b** XXXXXXXXXXXXX1—Watchpoint #16 triggers **nex_evto_b** XXXXXXXXXXXX1X—Watchpoint #17 triggers **nex_evto_b** XXXXXXXXXXX1XX—Watchpoint #18 triggers **nex_evto_b** XXXXXXXXXX1XXX—Watchpoint #19 triggers **nex_evto_b** XXXXXXXXX1XXXX—Watchpoint #20 triggers **nex_evto_b** XXXXXXXX1XXXXX—Watchpoint #21 triggers **nex_evto_b** XXXXXXX1XXXXXX—Watchpoint #22 triggers **nex_evto_b** XXXXXX1XXXXXXX—Watchpoint #23 triggers **nex_evto_b** XXXXX1XXXXXXXX—Watchpoint #24 triggers **nex_evto_b** XXXX1XXXXXXXXX—Watchpoint #25 triggers **nex_evto_b** XXX1XXXXXXXXXX—Watchpoint #26 triggers **nex_evto_b**

[1] The EOC bits in DC1 must be programmed to trigger $\overline{\text{EVTO}}$ on watchpoint occurrence for the EWC bits to have any effect.

14.4.6 Nexus Development Control Register 4 (DC4)

Nexus development control register 4 is used to control mark selection for program and data trace messaging and to mask events that initiate program correlation messages on the Nexus 3 module.

Figure 14-7 shows development control register 4.

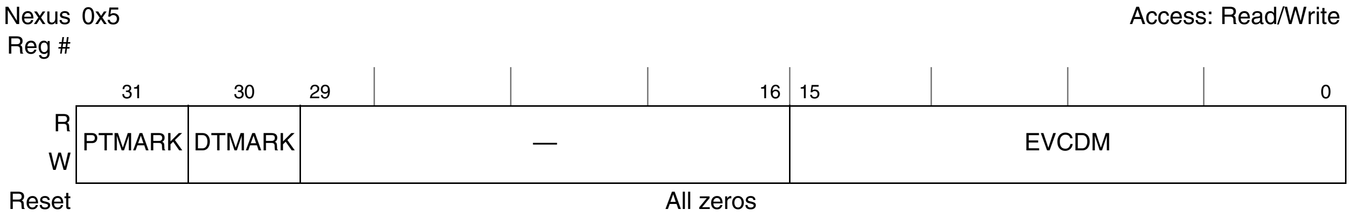


Figure 14-7. Development Control Register 4

Table 14-14 describes its fields.

Table 14-14. Development Control Register 4 Fields

Bits	Name	Description
31	PTMARK	Program Trace Mark 0 Ignore MSR[PMM] for masking program trace messages 1 Mask program trace messages when MSR[PMM] = 0; unmask program trace messages when MSR[PMM] = 1
30	DTMARK	Data Trace Mark 0 Ignore MSR[PMM] for masking data trace messages 1 Mask data trace messages when MSR[PMM] = 0; unmask data trace messages when MSR[PMM] = 1
29–16	—	Reserved
15–0	EVCDM	Event Code (EVCODE) Mask[1] 0000000000000000—No EVCODEs masked for Program Correlation Messages XXXXXXXXXXXXXXX1—EVCODE #0 is masked for Program Correlation Messages XXXXXXXXXXXXXX1X—EVCODE #1 is masked for Program Correlation Messages XXXXXXXXXXXXX1XX—EVCODE #2 is masked for Program Correlation Messages XXXXXXXXXXXX1XXX—EVCODE #3 is masked for Program Correlation Messages XXXXXXXXXXX1XXXX—EVCODE #4 is masked for Program Correlation Messages XXXXXXXXXX1XXXXX—EVCODE #5 is masked for Program Correlation Messages XXXXXXXXX1XXXXXX—EVCODE #6 is masked for Program Correlation Messages XXXXXXXX1XXXXXXX—EVCODE #7 is masked for Program Correlation Messages XXXXXXX1XXXXXXXX—EVCODE #8 is masked for Program Correlation Messages XXXXXX1XXXXXXXXX—EVCODE #9 is masked for Program Correlation Messages XXXXX1XXXXXXXXXX—EVCODE #10 is masked for Program Correlation Messages XXXX1XXXXXXXXXXX—EVCODE #11 is masked for Program Correlation Messages XXX1XXXXXXXXXXXX—EVCODE #12 is masked for Program Correlation Messages XX1XXXXXXXXXXXXX—EVCODE #13 is masked for Program Correlation Messages X1XXXXXXXXXXXXXX—EVCODE #14 is masked for Program Correlation Messages 1XXXXXXXXXXXXXXX—EVCODE #15 is masked for Program Correlation Messages

[1] Refer to Table 14-6 for implemented EVCODEs

14.4.7 Development Status Register (DS)

The development status register, shown in Figure 14-8, is used to report system debug status. When debug mode is entered or exited, or an SoC- or e200-defined low power mode is entered, a debug status message is transmitted with DS[31–24]. The external tool can read this register at any time.

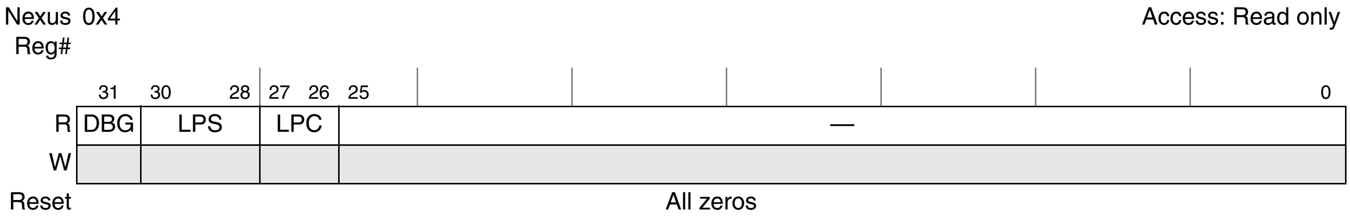


Figure 14-8. Development Status Register

Table 14-15 describes the development status register fields.

Table 14-15. Development Status Register Fields

Bits	Name	Description
31	DBG	DBG—e200 CPU debug mode status 0 CPU not in Debug mode 1 CPU in Debug mode (**jd_debug_b** signal asserted)
30–28	LPS	LPS—e200 system low power mode status 000 Normal (Run) mode XX1 DOZE mode (**p_doze** signal asserted) X1X NAP mode (**p_nap** signal asserted) 1XX SLEEP mode (**p_sleep** signal asserted)
27–26	LPC	LPC—e200 CPU low power mode status 00 Normal (Run) mode 01 CPU in Halted state (**p_halted** signal asserted) 10 CPU in Stopped state (**p_stopped** signal asserted) 11 CPU in Waiting state (**p_waiting** signal asserted)
25–0	—	Reserved for future functionality (read as 0)

14.4.8 Watchpoint Trigger Registers (WT, PTSTC, PTETC, DTSTC, DTETC)

The watchpoint trigger registers allows the watchpoints defined within the e200 Nexus1 logic to trigger actions. These watchpoints can control program and/or data trace enable and disable. The control bits can be used to produce a related window for triggering trace messages. The watchpoint trigger register (WT) is used to control triggering by a single selected watchpoint. The program trace start trigger control (PTSTC), program trace end trigger control (PTETC), data trace start trigger control (DTSTC), and data trace end trigger control (DTETC) are used for extended trigger controls for the respective function. If multiple watchpoints are desired for triggering, or a watchpoint beyond watchpoint #13 is required, then one or more of the extended watchpoint trigger registers may be used. A field encoding of 0b1111 in one of the WT register fields enables the corresponding extended trigger register. For all other WT field encodings, the corresponding extended trigger register is disabled and the contents are ignored.

When a start trigger is detected, the designated trace features become enabled, and the corresponding enable bits of the DC1 register are set. Whenever a stop trigger is detected, the designated trace features become disabled, and the corresponding enable bits of the DC1 register are cleared. If the same trigger condition is used for both start and stop triggering, the designated trace features toggle between being enabled and disabled at each occurrence of the trigger condition. Similarly, if start and stop triggers for a trace feature occur simultaneously, the designated trace feature toggles between enabled and disabled depending on the enable state at the time of the trigger events. For example, if tracing is enabled, and a start and stop trigger occur simultaneously, tracing is disabled. Direct writes of the DC1 register take precedence over any trace feature enable state that is derived from watchpoint triggering. A table of watchpoints can be found in Table 13-28.

Figure 14-9 shows the watchpoint trigger register.

Nexus Reg # 0xB

Access: Read/Write

	31 28	27 24	23 20	19 16	15 0
R W	PTS	PTE	DTS	DTE	—
Reset	All zeros				

Figure 14-9. Watchpoint Trigger (WT) Register

Table 14-16 details the watchpoint trigger register fields.

Table 14-16. Watchpoint Trigger Register Fields

Bits	Name	Description
31–28	PTS	PTS—Program Trace Start Control 0000 Trigger disabled 0001 Use Watchpoint #0 0010 Use Watchpoint #1 . . 1110 Use Watchpoint #13 1111 Use control settings in the PTSTC register
27–24	PTE	PTE—Program Trace End Control 0000 Trigger disabled 0001 Use Watchpoint #0 0010 Use Watchpoint #1 . . 1110 Use Watchpoint #13 1111 Use control settings in the PTETC register
23–20	DTS	DTS—Data Trace Start Control 0000 Trigger disabled 0001 Use Watchpoint #0 0010 Use Watchpoint #1 . . 1110 Use Watchpoint #13 1111 Use control settings in the DTSTC register

Table 14-16. Watchpoint Trigger Register Fields (continued)

19–16	DTE	DTE—Data Trace End Control 0000 Trigger disabled 0001 Use Watchpoint #0 0010 Use Watchpoint #1 . . 1110 Use Watchpoint #13 1111 Use control settings in the DTETC register
15–0	—	Reserved for future functionality (read as 0)

The PTSTC register, shown in Figure 14-10, is used for extended program trace start trigger control.

Nexus 0x35
Reg #

Access: Read/Write

	31	26	0
R	—	PTST	
W			
Reset	All zeros		

Figure 14-10. Program Trace Start Trigger Control (PTSTC) Register

Table 14-17 details the PTSTC register fields.

Table 14-17. Program Trace Start Trigger Control Register Fields

Bits	Name	Description
31–27	—	Reserved for future functionality (read as 0)
26–0	PTST	PTST—Program Trace Start Trigger Control 000000000000000000000000000—Trigger disabled XXXXXXXXXXXXXXXXXXXXXXXXXX1—Use Watchpoint #0 XXXXXXXXXXXXXXXXXXXXXXXXX1X—Use Watchpoint #1 XXXXXXXXXXXXXXXXXXXXXXXX1XX—Use Watchpoint #2 XXXXXXXXXXXXXXXXXXXXXXX1XXX—Use Watchpoint #3 XXXXXXXXXXXXXXXXXXXXXX1XXXX—Use Watchpoint #4 XXXXXXXXXXXXXXXXXXXXX1XXXXX—Use Watchpoint #5 XXXXXXXXXXXXXXXXXXXX1XXXXXX—Use Watchpoint #6 XXXXXXXXXXXXXXXXXXX1XXXXXXX—Use Watchpoint #7 XXXXXXXXXXXXXXXXXX1XXXXXXXX—Use Watchpoint #8 XXXXXXXXXXXXXXXXX1XXXXXXXXX—Use Watchpoint #9 XXXXXXXXXXXXXXXX1XXXXXXXXXX—Use Watchpoint #10 XXXXXXXXXXXXXXX1XXXXXXXXXXX—Use Watchpoint #11 XXXXXXXXXXXXXX1XXXXXXXXXXXX—Use Watchpoint #12 XXXXXXXXXXXXX1XXXXXXXXXXXXX—Use Watchpoint #13 XXXXXXXXXXXX1XXXXXXXXXXXXXX—Use Watchpoint #14 XXXXXXXXXXX1XXXXXXXXXXXXXXX—Use Watchpoint #15 XXXXXXXXXX1XXXXXXXXXXXXXXXX—Use Watchpoint #16 XXXXXXXXX1XXXXXXXXXXXXXXXXX—Use Watchpoint #17 XXXXXXXX1XXXXXXXXXXXXXXXXXX—Use Watchpoint #18 XXXXXXX1XXXXXXXXXXXXXXXXXXX—Use Watchpoint #19 XXXXXX1XXXXXXXXXXXXXXXXXXXX—Use Watchpoint #20 XXXXX1XXXXXXXXXXXXXXXXXXXXX—Use Watchpoint #21 XXXX1XXXXXXXXXXXXXXXXXXXXXX—Use Watchpoint #22 XXX1XXXXXXXXXXXXXXXXXXXXXXX—Use Watchpoint #23 XX1XXXXXXXXXXXXXXXXXXXXXXXX—Use Watchpoint #24 X1XXXXXXXXXXXXXXXXXXXXXXXXX—Use Watchpoint #25 1XXXXXXXXXXXXXXXXXXXXXXXXXX—Use Watchpoint #26

The PTETC register, shown in Figure 14-11, is used for extended program trace end trigger control.

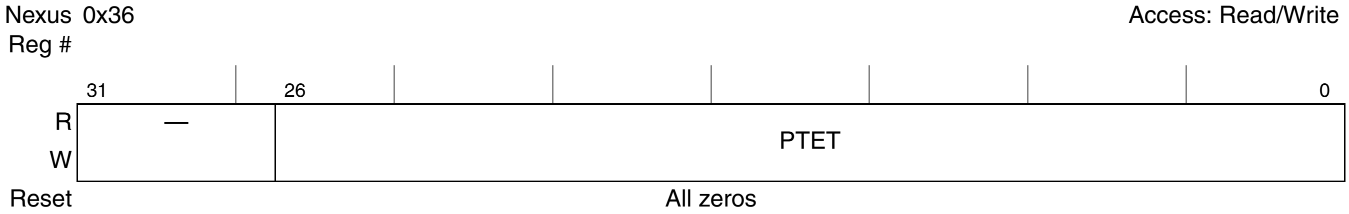


Figure 14-11. Program Trace End Trigger Control (PTETC) Register

Table 14-18 describes the PTETC register fields.

Table 14-18. Program Trace End Trigger Control Register Fields

Bits	Name	Description
31–27	—	Reserved for future functionality (read as 0)
26–0	PTET	PTET—Program Trace End Trigger Control 000000000000000000000000000—Trigger disabled XXXXXXXXXXXXXXXXXXXXXXXXXX1—Use Watchpoint #0 XXXXXXXXXXXXXXXXXXXXXXXXX1X—Use Watchpoint #1 XXXXXXXXXXXXXXXXXXXXXXXX1XX—Use Watchpoint #2 XXXXXXXXXXXXXXXXXXXXXXX1XXX—Use Watchpoint #3 XXXXXXXXXXXXXXXXXXXXXX1XXXX—Use Watchpoint #4 XXXXXXXXXXXXXXXXXXXXX1XXXXX—Use Watchpoint #5 XXXXXXXXXXXXXXXXXXXX1XXXXXX—Use Watchpoint #6 XXXXXXXXXXXXXXXXXXX1XXXXXXX—Use Watchpoint #7 XXXXXXXXXXXXXXXXXX1XXXXXXXX—Use Watchpoint #8 XXXXXXXXXXXXXXXXX1XXXXXXXXX—Use Watchpoint #9 XXXXXXXXXXXXXXXX1XXXXXXXXXX—Use Watchpoint #10 XXXXXXXXXXXXXXX1XXXXXXXXXXX—Use Watchpoint #11 XXXXXXXXXXXXXX1XXXXXXXXXXXX—Use Watchpoint #12 XXXXXXXXXXXXX1XXXXXXXXXXXXX—Use Watchpoint #13 XXXXXXXXXXXX1XXXXXXXXXXXXXX—Use Watchpoint #14 XXXXXXXXXXX1XXXXXXXXXXXXXXX—Use Watchpoint #15 XXXXXXXXXX1XXXXXXXXXXXXXXXX—Use Watchpoint #16 XXXXXXXXX1XXXXXXXXXXXXXXXXX—Use Watchpoint #17 XXXXXXXX1XXXXXXXXXXXXXXXXXX—Use Watchpoint #18 XXXXXXX1XXXXXXXXXXXXXXXXXXX—Use Watchpoint #19 XXXXXX1XXXXXXXXXXXXXXXXXXXX—Use Watchpoint #20 XXXXX1XXXXXXXXXXXXXXXXXXXXX—Use Watchpoint #21 XXXX1XXXXXXXXXXXXXXXXXXXXXX—Use Watchpoint #22 XXX1XXXXXXXXXXXXXXXXXXXXXXX—Use Watchpoint #23 XX1XXXXXXXXXXXXXXXXXXXXXXXX—Use Watchpoint #24 X1XXXXXXXXXXXXXXXXXXXXXXXXX—Use Watchpoint #25 1XXXXXXXXXXXXXXXXXXXXXXXXXX—Use Watchpoint #26

The DTSTC register, shown in Figure 14-12, is used for extended data trace start trigger control.

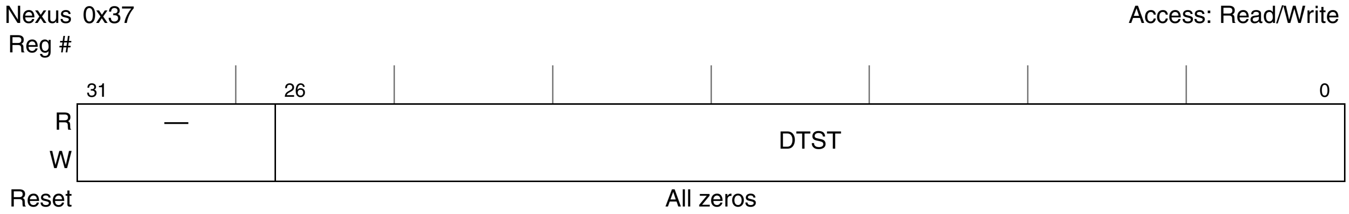


Figure 14-12. Data Trace Start Trigger Control (DTSTC) Register

Table 14-19 details the DTSTC register fields.

Table 14-19. Data Trace Start Trigger Control Register Fields

Bits	Name	Description
31–27	—	Reserved for future functionality (read as 0)
26–0	DTST	DTST—Data Trace Start Trigger Control 000000000000000000000000000—Trigger disabled XXXXXXXXXXXXXXXXXXXXXXXXXX1—Use Watchpoint #0 XXXXXXXXXXXXXXXXXXXXXXXXX1X—Use Watchpoint #1 XXXXXXXXXXXXXXXXXXXXXXXX1XX—Use Watchpoint #2 XXXXXXXXXXXXXXXXXXXXXXX1XXX—Use Watchpoint #3 XXXXXXXXXXXXXXXXXXXXXX1XXXX—Use Watchpoint #4 XXXXXXXXXXXXXXXXXXXXX1XXXXX—Use Watchpoint #5 XXXXXXXXXXXXXXXXXXXX1XXXXXX—Use Watchpoint #6 XXXXXXXXXXXXXXXXXXX1XXXXXXX—Use Watchpoint #7 XXXXXXXXXXXXXXXXXX1XXXXXXXX—Use Watchpoint #8 XXXXXXXXXXXXXXXXX1XXXXXXXXX—Use Watchpoint #9 XXXXXXXXXXXXXXXX1XXXXXXXXXX—Use Watchpoint #10 XXXXXXXXXXXXXXX1XXXXXXXXXXX—Use Watchpoint #11 XXXXXXXXXXXXXX1XXXXXXXXXXXX—Use Watchpoint #12 XXXXXXXXXXXXX1XXXXXXXXXXXXX—Use Watchpoint #13 XXXXXXXXXXXX1XXXXXXXXXXXXXX—Use Watchpoint #14 XXXXXXXXXXX1XXXXXXXXXXXXXXX—Use Watchpoint #15 XXXXXXXXXX1XXXXXXXXXXXXXXXX—Use Watchpoint #16 XXXXXXXXX1XXXXXXXXXXXXXXXXX—Use Watchpoint #17 XXXXXXXX1XXXXXXXXXXXXXXXXXX—Use Watchpoint #18 XXXXXXX1XXXXXXXXXXXXXXXXXXX—Use Watchpoint #19 XXXXXX1XXXXXXXXXXXXXXXXXXXX—Use Watchpoint #20 XXXXX1XXXXXXXXXXXXXXXXXXXXX—Use Watchpoint #21 XXXX1XXXXXXXXXXXXXXXXXXXXXX—Use Watchpoint #22 XXX1XXXXXXXXXXXXXXXXXXXXXXX—Use Watchpoint #23 XX1XXXXXXXXXXXXXXXXXXXXXXXX—Use Watchpoint #24 X1XXXXXXXXXXXXXXXXXXXXXXXXX—Use Watchpoint #25 1XXXXXXXXXXXXXXXXXXXXXXXXXX—Use Watchpoint #26

The DTETC register, shown in Figure 14-13, is used for extended data trace end trigger control.

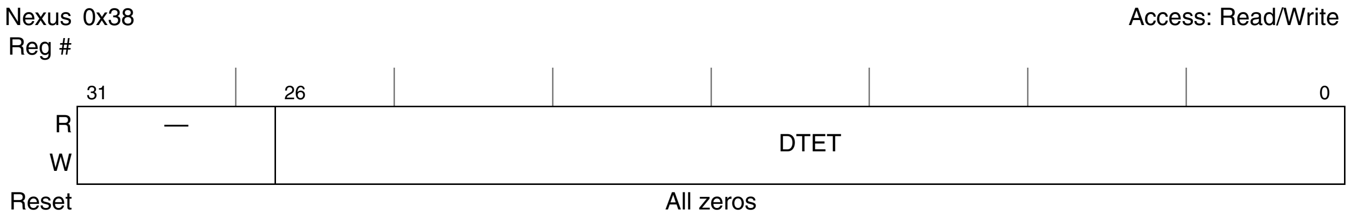


Figure 14-13. Data Trace End Trigger Control (DTETC) Register

Table 14-20 describes the DTETC register fields.

Table 14-20. Data Trace End Trigger Control Register Fields

Bits	Name	Description
31–27	—	Reserved for future functionality (read as 0)
26–0	DTET	DTET—Data Trace End Trigger Control 000000000000000000000000000—Trigger disabled XXXXXXXXXXXXXXXXXXXXXXXXXX1—Use Watchpoint #0 XXXXXXXXXXXXXXXXXXXXXXXXX1X—Use Watchpoint #1 XXXXXXXXXXXXXXXXXXXXXXXX1XX—Use Watchpoint #2 XXXXXXXXXXXXXXXXXXXXXXX1XXX—Use Watchpoint #3 XXXXXXXXXXXXXXXXXXXXXX1XXXX—Use Watchpoint #4 XXXXXXXXXXXXXXXXXXXXX1XXXXX—Use Watchpoint #5 XXXXXXXXXXXXXXXXXXXX1XXXXXX—Use Watchpoint #6 XXXXXXXXXXXXXXXXXXX1XXXXXXX—Use Watchpoint #7 XXXXXXXXXXXXXXXXXX1XXXXXXXX—Use Watchpoint #8 XXXXXXXXXXXXXXXXX1XXXXXXXXX—Use Watchpoint #9 XXXXXXXXXXXXXXXX1XXXXXXXXXX—Use Watchpoint #10 XXXXXXXXXXXXXXX1XXXXXXXXXXX—Use Watchpoint #11 XXXXXXXXXXXXXX1XXXXXXXXXXXX—Use Watchpoint #12 XXXXXXXXXXXXX1XXXXXXXXXXXXX—Use Watchpoint #13 XXXXXXXXXXXX1XXXXXXXXXXXXXX—Use Watchpoint #14 XXXXXXXXXXX1XXXXXXXXXXXXXXX—Use Watchpoint #15 XXXXXXXXXX1XXXXXXXXXXXXXXXX—Use Watchpoint #16 XXXXXXXXX1XXXXXXXXXXXXXXXXX—Use Watchpoint #17 XXXXXXXX1XXXXXXXXXXXXXXXXXX—Use Watchpoint #18 XXXXXXX1XXXXXXXXXXXXXXXXXXX—Use Watchpoint #19 XXXXXX1XXXXXXXXXXXXXXXXXXXX—Use Watchpoint #20 XXXXX1XXXXXXXXXXXXXXXXXXXXX—Use Watchpoint #21 XXXX1XXXXXXXXXXXXXXXXXXXXXX—Use Watchpoint #22 XXX1XXXXXXXXXXXXXXXXXXXXXXX—Use Watchpoint #23 XX1XXXXXXXXXXXXXXXXXXXXXXXX—Use Watchpoint #24 X1XXXXXXXXXXXXXXXXXXXXXXXXX—Use Watchpoint #25 1XXXXXXXXXXXXXXXXXXXXXXXXXX—Use Watchpoint #26

14.4.9 Nexus Watchpoint Mask Register (WMSK)

The Nexus watchpoint mask register, shown in Figure 14-14, controls which watchpoint events are enabled to produce watchpoint trace messages. Note that DC1[TM] must also be programmed to generate watchpoint trace messages.

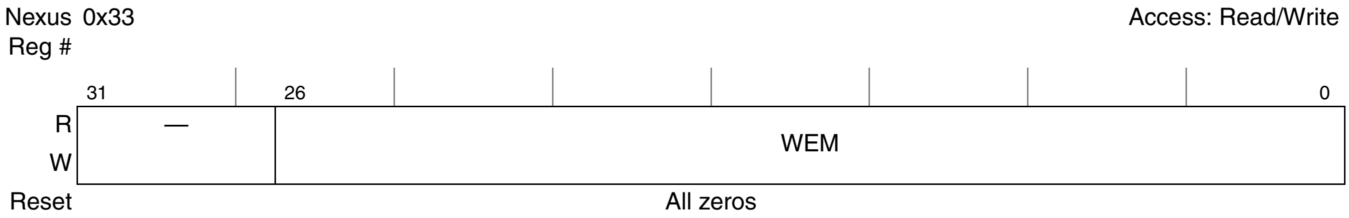


Figure 14-14. Watchpoint Mask Register

Table 14-21 describes the watchpoint trigger register fields.

Table 14-21. Watchpoint Mask Register Fields

Bits	Name	Description
31–27	—	Reserved for future functionality (read as 0)
26–0	WEM	WEM—Watchpoint Enable for Messaging 000000000000000000000000000—No Watchpoints enabled for Watchpoint Trace Messaging XXXXXXXXXXXXXXXXXXXXXXXXXX1—Watchpoint #0 enabled for WTM XXXXXXXXXXXXXXXXXXXXXXXXX1X—Watchpoint #1 enabled for WTM XXXXXXXXXXXXXXXXXXXXXXXX1XX—Watchpoint #2 enabled for WTM XXXXXXXXXXXXXXXXXXXXXXX1XXX—Watchpoint #3 enabled for WTM XXXXXXXXXXXXXXXXXXXXXX1XXXX—Watchpoint #4 enabled for WTM XXXXXXXXXXXXXXXXXXXXX1XXXXX—Watchpoint #5 enabled for WTM XXXXXXXXXXXXXXXXXXXX1XXXXXX—Watchpoint #6 enabled for WTM XXXXXXXXXXXXXXXXXXX1XXXXXXX—Watchpoint #7 enabled for WTM XXXXXXXXXXXXXXXXXX1XXXXXXXX—Watchpoint #8 enabled for WTM XXXXXXXXXXXXXXXXX1XXXXXXXXX—Watchpoint #9 enabled for WTM XXXXXXXXXXXXXXXX1XXXXXXXXXX—Watchpoint #10 enabled for WTM XXXXXXXXXXXXXXX1XXXXXXXXXXX—Watchpoint #11 enabled for WTM XXXXXXXXXXXXXX1XXXXXXXXXXXX—Watchpoint #12 enabled for WTM XXXXXXXXXXXXX1XXXXXXXXXXXXX—Watchpoint #13 enabled for WTM XXXXXXXXXXXX1XXXXXXXXXXXXXX—Watchpoint #14 enabled for WTM XXXXXXXXXXX1XXXXXXXXXXXXXXX—Watchpoint #15 enabled for WTM XXXXXXXXXX1XXXXXXXXXXXXXXXX—Watchpoint #16 enabled for WTM XXXXXXXXX1XXXXXXXXXXXXXXXXX—Watchpoint #17 enabled for WTM XXXXXXXX1XXXXXXXXXXXXXXXXXX—Watchpoint #18 enabled for WTM XXXXXXX1XXXXXXXXXXXXXXXXXXX—Watchpoint #19 enabled for WTM XXXXXX1XXXXXXXXXXXXXXXXXXXX—Watchpoint #20 enabled for WTM XXXXX1XXXXXXXXXXXXXXXXXXXXX—Watchpoint #21 enabled for WTM XXXX1XXXXXXXXXXXXXXXXXXXXXX—Watchpoint #22 enabled for WTM XXX1XXXXXXXXXXXXXXXXXXXXXXX—Watchpoint #23 enabled for WTM XX1XXXXXXXXXXXXXXXXXXXXXXXX—Watchpoint #24 enabled for WTM X1XXXXXXXXXXXXXXXXXXXXXXXXX—Watchpoint #25 enabled for WTM 1XXXXXXXXXXXXXXXXXXXXXXXXXX—Watchpoint #26 enabled for WTM

14.4.10 Nexus Overrun Control Register (OVCR)

The Nexus overrun control register, shown in Figure 14-15, controls Nexus behavior as the internal message queues fill up. Response options include suppressing selected message types or stalling processor instruction execution.

Nexus Reg # 0x32

Access: Read/Write

	31–30	29–28	27–22	21–16	15–14	13–12	11–1	0
R W	—	SPTHOLD	—	SPEN	—	STTHOLD	—	STEN
Reset	All zeros							

Figure 14-15. Nexus Overrun Control Register

Table 14-22 describes the fields.

Table 14-22. Nexus Overrun Control Register Fields

Bits	Name	Description
31–30	—	Reserved, should be cleared
29–28	SPTHOLD	Suppression Threshold 00 Suppression threshold is when message queues are ¼ full. 01 Suppression threshold is when message queues are ½ full. 10 Suppression threshold is when message queues are $\frac{3}{4}$ full. 11 Reserved
27–22	—	Reserved, should be cleared
21–16	SPEN	Suppression Enable 000000Suppression is disabled xxxxx1 Ownership Trace message suppression is enabled. xxxx1x Data Trace message suppression is enabled. xxx1xx Program Trace message suppression is enabled. xx1xxx Watchpoint Trace message suppression is enabled. x1xxxx Reserved 1xxxxx Data Acquisition message suppression is enabled.
15–14	—	Reserved, should be cleared
13–12	STTHOLD	Stall Threshold 00 Stall threshold is when message queues are ¼ full. 01 Stall threshold is when message queues are ½ full. 10 Stall threshold is when message queues are $\frac{3}{4}$ full 11 Reserved
11–1	—	Reserved, should be cleared
0	STEN	Stall Enable 0 Stalling is disabled. 1 Stalling is enabled.

14.4.11 Data Trace Control Register (DTC)

The data trace control register, shown in Figure 14-16, controls whether DTM messages are restricted to reads, writes, or both for a user programmable address range. There are four data trace channels controlled by the DTC for the Nexus 3 module. Channels can be programmed to trace data accesses or instruction accesses, but not independently.

Nexus Reg # 0xD

Access: Read/Write

	31 30	29 28	27 26	25 24	23 8	7	6	5	4	3	2 0
R W	RWT1	RWT2	RWT3	RWT4	—	RC1	RC2	RC3	RC4	DI	—
Reset	All zeros										

Figure 14-16. Data Trace Control Register

Table 14-23 describes the data trace control register fields.

Table 14-23. Data Trace Control Register Fields

Bits	Name	Description
31–30	RWT1	RWT1—Read/Write Trace 1 00 No trace enabled X1 Enable Data Read Trace 1X Enable Data Write Trace
29–28	RWT2	RWT2—Read/Write Trace 2 00 No trace enabled X1 Enable Data Read Trace 1X Enable Data Write Trace
27–26	RWT3	RWT3—Read/Write Trace 3 00 No trace enabled X1 Enable Data Read Trace 1X Enable Data Write Trace
25–24	RWT4	RWT4—Read/Write Trace 4 00 No trace enabled X1 Enable Data Read Trace 1X Enable Data Write Trace
23–8	—	Reserved for future functionality (read as 0)
7	RC1	RC1—Range Control 1 0 Condition trace on address within range 1 Condition trace on address outside of range
6	RC2	RC2—Range Control 2 0 Condition trace on address within range 1 Condition trace on address outside of range
5	RC3	RC3—Range Control 3 0 Condition trace on address within range 1 Condition trace on address outside of range
4	RC4	RC4—Range Control 4 0 Condition trace on address within range 1 Condition trace on address outside of range
3	DI	DI—Data Access/Instruction Access Trace 0 Condition trace on data accesses 1 Condition trace on instruction accesses
2–0	—	RES—Reserved for future functionality (read as 0)

14.4.12 Data Trace Start Address Registers (DTSA1–4)

The data trace start address registers, shown in Figure 14-17, define the start addresses for each trace channel.

Figure 14-17. Data Trace Start Address *n* Register

14.4.13 Data Trace End Address Registers (DTEA1–4)

The data trace end address registers, shown in Figure 14-18, define the end addresses for each trace channel.

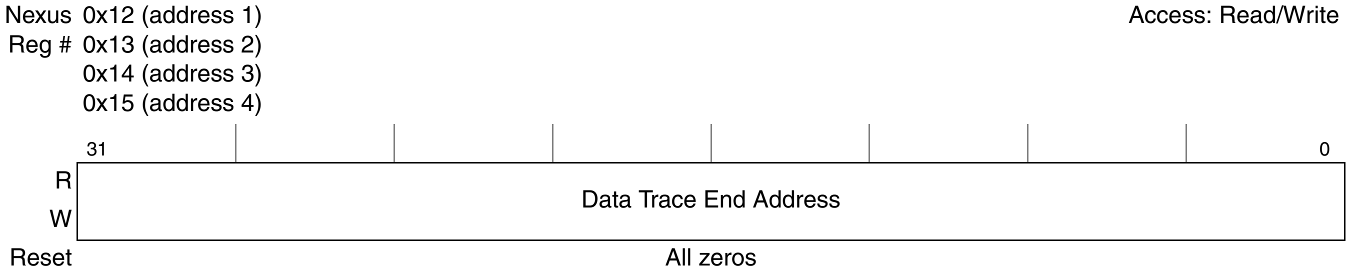


Figure 14-18. Data Trace End Address *n* Register

Table 14-24 illustrates the range that will be selected for data trace for various cases of DTSA being less than, greater than, or equal to DTEA.

Table 14-24. Data Trace—Address Range Options

Programmed Values	Range Control Bit Value	Range Selected
DTSA < DTEA	0	DTSA → ← DTEA
DTSA < DTEA	1	← DTSA DTEA →
DTSA > DTEA	N/A	Invalid Range—no trace
DTSA = DTEA	N/A	Invalid Range—no trace

NOTE

DTSA must be less than DTEA to guarantee correct data write/read traces. Data trace ranges are inclusive of the DTSA and DTEA addresses for Range Control settings indicating “within range.” They are exclusive of the DTSA and DTEA addresses for range control settings indicating “outside of range.”

14.4.14 Read/Write Access Control/Status (RWCS)

The read write access control/status register, shown in Figure 14-19, provides control for read/write access. Read/write access provides DMA-like access to memory-mapped resources on the AHB System bus either while the processor is halted or during runtime. Control is provided over access type, size, count, and certain bus attributes. The RWCS register also provides read/write access status information per Table 14-26.

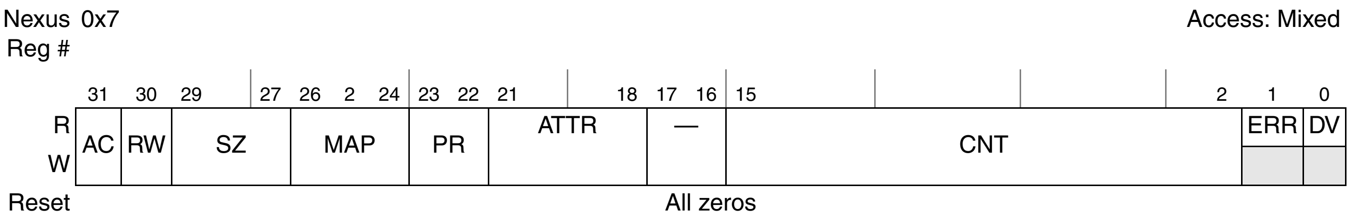


Figure 14-19. Read/Write Access Control/Status Register

Table 14-25 describes the read/write access control/status register fields.

Table 14-25. Read/Write Access Control/Status Register Fields

Bits	Name	Description
RWCS[31]	AC	AC—Access Control 0 End access 1 Start access
RWCS[30]	RW	RW—Read/Write Select 0 Read access 1 Write access
RWCS[29–27]	SZ	SZ—Word Size 000 8-bit (byte) 001 16-bit (half-word) 010 32-bit (word) 011 64-bit (double word, requires two passes through RWD) 100-111 = Reserved (default to word)
RWCS[26–24]	MAP	MAP—MAP Select 000 Primary memory map 001–111 Reserved
RWCS[23–22]	PR[1]	PR—Read/Write Access Priority 00 Reserved (default to highest priority) 01 Reserved (default to highest priority) 10 Reserved (default to highest priority) 11 Highest access priority
RWCS[21–18]	ATTR	ATTR—Access Atrributes 0xxx p_d_gbl driven to 0 for accesses 1xxx p_d_gbl driven to 1 for accesses x0xx p_d_hprot[4] driven to 0 for accesses x1xx p_d_hprot[4] driven to 1 for accesses xx0x p_d_hprot[3] driven to 0 for accesses xx1x p_d_hprot[3] driven to 1 for accesses xxx0 p_d_hprot[2] driven to 0 for accesses xxx1 p_d_hprot[2] driven to 1 for accesses

Table 14-25. Read/Write Access Control/Status Register Fields (continued)

Bits	Name	Description
RWCS[17–16]	—	RES—Reserved for future functionality
RWCS[15–2]	CNT	CNT—Access Control Count hhhh Number of accesses of word size SZ
RWCS[1]	ERR[2]	ERR—Read/Write Access Error (see Table 14-26)
RWCS[0]	DV[2]	DV—Read/Write Access Data Valid (see Table 14-26)

[1] The priority functionality is not currently implemented

[2] ERR and DV are read-only

Table 14-26. Read/Write Access Status Bit Encoding

Read Action	Write Action	ERR	DV
Read Access has not completed	Write Access completed without error	0	0
Read Access error has occurred	Write Access error has occurred	1	0
Read Access completed without error	Write Access has not completed	0	1
Not Allowed	Not allowed	1	1

14.4.15 Read/Write Access Data (RWD)

The read/write access data register (RWD), shown in Figure 14-20 provides the data to/from system bus memory-mapped locations when initiating a read or a write access.

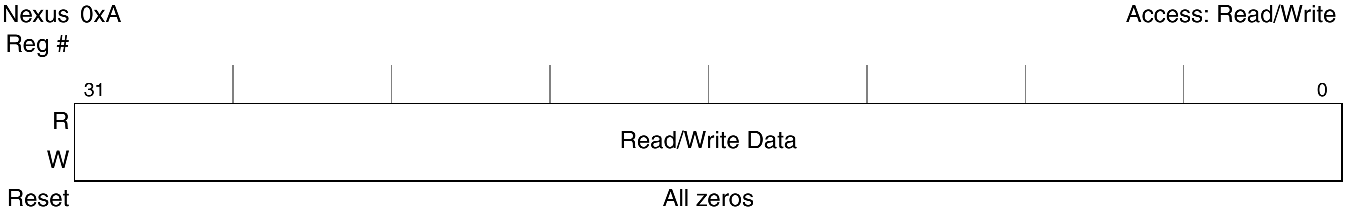


Figure 14-20. Read/Write Access Data Register

Read/write accesses to the AHB require that the debug firmware properly retrieve/place the data in the RWD. Table 14-27 shows the proper placement of data into the RWD. Note that double-word transfers require two passes through RWD.

Table 14-27. RWD Data Placement for Transfers

Transfer Size and byte offset	RWA(2–0)	RWCS[SZ]	RWD			
			31–24	23–16	15–8	7–0
Byte	x x x	0 0 0	—	—	—	X
Half	x x 0	0 0 1	—	—	X	X
Word	x 0 0	0 1 0	X	X	X	X

Table 14-27. RWD Data Placement for Transfers

Transfer Size and byte offset	RWA(2–0)	RWCS[SZ]	RWD 31–24	RWD 23–16	RWD 15–8	RWD 7–0
Double word	0 0 0	0 1 1				
first RWD pass (low order data)			X	X	X	X
second RWD pass (high order data)			X	X	X	X

Notes:
"X" indicates byte lanes with valid data
"-" indicates byte lanes which will contain unused data.

Table 14-28 shows the mapping of RWD bytes to byte lanes of the AHB read and write data buses.

Table 14-28. RWD Byte Lane Mapping

Transfer Size and byte offset	RWA(2–0)	RWD 31–24	RWD 23–16	RWD 15–8	RWD 7–0
Byte = 000	0 0 0	—	—	—	AHB[7–0]
Byte = 001	0 0 1	—	—	—	AHB[15–8]
Byte = 010	0 1 0	—	—	—	AHB[23–16]
Byte = 011	0 1 1	—	—	—	AHB[31–24]
Byte = 100	1 0 0	—	—	—	AHB[39–32]
Byte = 101	1 0 1	—	—	—	AHB[47–40]
Byte = 110	1 1 0	—	—	—	AHB[55–48]
Byte = 111	1 1 1	—	—	—	AHB[63–56]
Half = 000	0 0 0	—	—	AHB[15–8]	AHB[7–0]
Half = 010	0 1 0	—	—	AHB[31–24]	AHB[23–16]
Half = 100	1 0 0	—	—	AHB[47–40]	AHB[39–32]
Half = 110	1 1 0	—	—	AHB[63–56]	AHB[55–48]
Word = 000	0 0 0	AHB[31–24]	AHB[23–16]	AHB[15–8]	AHB[7–0]
Word = 100	1 0 0	AHB[63–56]	AHB[55–48]	AHB[47–40]	AHB[39–32]
Double word = 000	0 0 0				
first RWD pass		AHB[31–24]	AHB[23–16]	AHB[15–8]	AHB[7–0]
second RWD pass		AHB[63–56]	AHB[55–48]	AHB[47–40]	AHB[39–32]

Note: :
— indicates byte lanes which contain unused data.

14.4.16 Read/Write Access Address (RWA)

The read/write access address register, shown in Figure 14-21, provides the system bus address to be accessed when initiating a read or a write access.

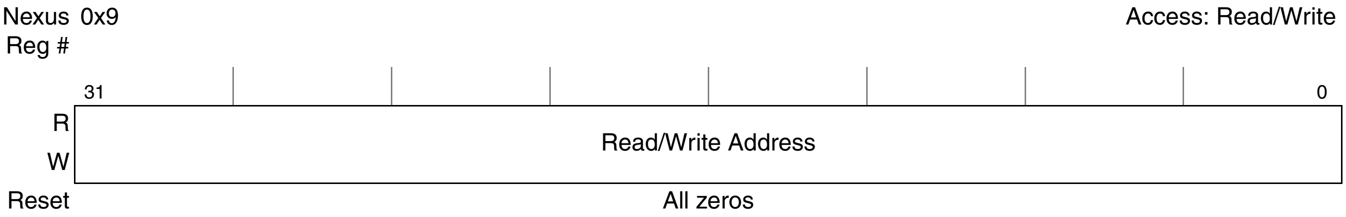


Figure 14-21. Read/Write Access Address Register

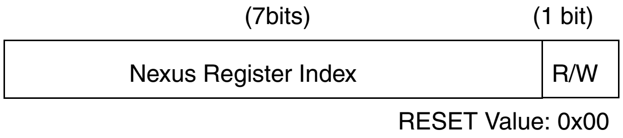
14.5 JTAG/OnCE Nexus 3 Register Access

Access to Nexus 3 register resources is enabled by loading a single instruction ("*NEXUS3-ACCESS*") into the JTAG instruction register (IR) (OnCE OCMD register). For the Nexus 3 block, the OCMD value is 0b0001111100.

Once the "*NEXUS3-ACCESS*" instruction has been loaded, the JTAG/OnCE port allows tool/target communications with all Nexus 3 registers according to the register map in Table 14-8.

The reading/writing of a Nexus 3 register then requires two passes through the data-scan (DR) path of the JTAG state machine (see Section 14.21, "IEEE 1149.1 (JTAG) RD/WR Sequences").

The first pass through the DR selects the Nexus 3 register to be accessed by providing an index (see Table 14-8), and the direction (read/write). This is achieved by loading an 8-bit value into the JTAG Data Register (DR). This register has the following format:



Nexus Register Index:	Selected from values in Table 14-8
Read/Write (R/W):	0 Read 1 Write

The second pass through the DR shifts the data in or out of the JTAG port, LSB first.

1. During a read access, data is latched from the selected Nexus register when the JTAG state machine passes through the Capture-DR state.
2. During a write access, data is latched into the selected Nexus register when the JTAG state machine passes through the Update-DR state.

14.6 Nexus Message Fields

Nexus messages are comprised of fields. Each field contains a distinct piece of information within a message, and each message contains multiple fields. Messages are transferred in packets over the Auxiliary Output protocol. A packet is a collection of fields. A packet may contain any number of fixed length fields, but may contain at most one variable length field. The variable length field must be the last field in a packet. The following subsections describe a subset of the message field types.

14.6.1 TCODE Field

The TCODE field is a 6-bit fixed length field that identifies the type of message and its format. The field encodings are assigned by IEEE-ISTO 5001.

14.6.2 Source ID Field (SRC)

Each Nexus module in a device is identified by a unique client source identification number. The number assigned to each Nexus module is determined by the SoC integrator, and is provided on the **nex3_ext_src_id[0:3]** input signals. Multithreaded processors may assign additional source ID information to indicate which thread a message is associated with. The e200z7 Nexus 3 module implements a 4-bit fixed length Source ID field consisting of a Client Source ID.

14.6.3 Relative Address Field (U-ADDR)

The non-sync forms of the program and data trace messages include addresses that are relative to the address which was transmitted in the previous program or data trace message respectively. The relative address format is compliant with IEEE-ISTO 5001 and is designed to reduce the number of bits transmitted for address fields.

The relative address is generated by XORing the new address with the previous and then using only the results up to the most significant 1. To recreate the original address, the relative address is XORed with the previously decoded address.

The relative address of a program trace message is calculated with respect to the previous program trace message, regardless of any address information that may have been sent in any other trace messages in the interim between the two program trace messages.

The relative address of a data trace message is calculated with respect to the previous data trace message, regardless of any address information that may have been sent in any other trace messages in the interim between the two data trace messages.

Figure 14-22 shows the relative address generation and re-creation, with the previous address (A1) = 0x0003_FC01 and new address (A2) = 0x0003_F365.

Message Generation:

A1 = 0000 0000 0000 0011 1111 1100 0000 0001
A2 = 0000 0000 0000 0011 1111 0011 0110 0101

A1 + A2 = 0000 0000 0000 0000 0000 1111 0110 0100

Address Message (M1) = 1111 0110 0100

Address Re-creation:

A1 + M1 = A2
A1 = 0000 0000 0000 0011 1111 1100 0000 0001
M1 = 0000 0000 0000 0000 0000 1111 0110 0100

A2 = 0000 0000 0000 0011 1111 0011 0110 0101

Figure 14-22. Relative Address Generation and Re-creation

14.6.4 Full Address Field (F-ADDR)

Program trace synchronization messages provide the full address associated with the trace event (leading zeroes may be truncated) with the intent of providing a reference point for development tools to operate from when reconstructing relative addresses. Synchronization messages are generated at significant mode switches and are also generated periodically to ensure that development tools are guaranteed to have a reference address given a sufficiently large sample of trace messages.

14.6.5 Address Space Indication Field (MAP)

Data trace messages and indirect-type program trace messages provide the address space status (DS or IS value) in the address space (MAP) field. For data trace, the MAP field indicates the DS space (MSR_{DS} value) used for the data access. For program trace, the MAP field is used to indicate the future space used for instruction execution (new value of MSR[IS]). A change in instruction address space will only occur on reset, on an exception, or via a **mtmsr**, **rfi**, **rfci**, **rfdi**, or **rfmci** instruction. A potential change in address space via an exception or via an **rfi**, **rfci**, **rfdi**, or **rfmci** instruction will cause a program trace indirect branch message to be generated indicating the new address space (IS) value, along with ICNT and HIST information for instructions executed up to the change (including the **rfi**, **rfci**, **rfdi**, or **rfmci**). A change in address space via a **mtmsr** instruction will cause a program correlation message to be generated indicating the new address space (IS) value, along with ICNT and HIST information for instructions executed prior to the change (including the **mtmsr**).

14.7 Nexus Message Queues

The Nexus 3 module implements internal message queues capable of storing up to three messages per cycle into a small initial queue which then fills a larger queue at up to two messages per cycle. Messages that enter the queues are transmitted in the order in which they are received.

If more than three messages attempt to enter the queue in the same cycle, the highest priority messages are stored and the remaining message(s) are dropped due to a collision. Collision events are expected to be rare.

The overrun control register (OVCR) controls the Nexus behavior as the message queue fills. The Nexus block may be programmed to do the following:

- Allow the queue to overflow, drain the contents, queue an overrun error message and resume tracing.
- Stall the processor when the queue utilization reaches the selected threshold.
- Suppress selected message types when the queue utilization reaches the selected threshold.

14.7.1 Message Queue Overrun

In this mode, the message queue stops accepting messages when an overrun condition is detected. The contents of the queues are allowed to drain until empty. Incoming messages are discarded until the queue is emptied. Once empty, an overrun error message is enqueued that contains information about the types of messages which were discarded due to the overrun condition.

14.7.2 CPU Stall

In this mode, processor instruction issue is stalled when the queue utilization reaches the selected threshold. The processor is stalled long enough drop one threshold level below the level which triggered the stall. For example, if stalling the processor is triggered at ¼ full, the stall will stay in effect until the queue utilization drops to empty. There may be significant skid from the time that the stall request is made until the processor is able to stop completing instructions. This skid should be taken into consideration when programming the threshold. Refer to Section 14.4.10, “Nexus Overrun Control Register (OVCR),” for complete programming options.

14.7.3 Message Suppression

In this mode, the message queue disables selected message types when the queue utilization reaches the selected threshold. This allows lower bandwidth tracing to continue and possibly avoid an overrun condition. If an overrun condition occurs despite this message suppression, the queue will respond according to the behavior described in Section 14.7.1, “Message Queue Overrun.” Once triggered, message suppression remains in effect until queue utilization drops to the threshold below the level selected to trigger suppression.

14.7.4 Nexus Message Priority

Nexus messages may be lost due to contention with other message types under the following circumstances: more than three messages are generated in the same cycle.

Up to three message requests can be queued into the message buffer in a given cycle. If more than three message requests exist in a given cycle, the three highest priority message classes are queued into the message buffer. The remaining messages that did not successfully queue into the message buffer in that cycle generate subsequent responses as detailed in Table 14-29.

The CPU is capable of completing two instructions per cycle. If multiple trace messages need to be queued at the same time, they will be queued with the following priority: Instruction 0 (oldest instruction) (WPM → DQM → PCM[PIDMSG] → OTM → BTM → DTM) → Instruction1 (newer instruction) (WPM → DQM → OTM → BTM → DTM). Up to three messages may be simultaneously queued. Note that for the cycle following a dropped PTM, non-periodic OTM, or DQM message, only two other messages may be queued in addition to the dropped error message.

Watchpoint messages from instructions that complete at the same time or events that occur during the same cycle will be combined.

Table 14-29 lists the various message types and their relative priority from highest to lowest.

Table 14-29. Message Type Priority and Message Dropped Responses

Message Type	Message	Priority	Message Dropped Response
Error	Error	0 (highest)	N/A[1]
WP (Watchpoint Trace)	WPM (Watchpoint Message)	1	N/A[1]
DQ (Data Acquisition)	DQM (Data Acquisition Message)	2	DQM Error Message
Program Trace (PID MSG)	PCM—PID or mtmsr IS update (Program Correlation Message)	2	OTM Error Message
OT (Ownership)	OTM—PID update (Ownership Trace Message)	2	OTM Error Message[2]
Program Trace	BTM (Branch Trace Message)	2	BTM Error Message, Sync upgrade next BTM
	RFM (Resource Full for Instruction counter or history buffer)	3	BTM Error Message Sync upgrade next BTM
	DS (Debug Status Message)	4	Sync upgrade next BTM
	PCM (Program Correlation Message)	5	BTM Error Message Sync upgrade next BTM
DT (Data Trace)	DTM (Data Trace Message)	6	Sync upgrade next DTM
OT (Ownership)	OTM—Periodic update (Ownership Trace Message)	7 (lowest)	None

[1] Error and watchpoint messages are not dropped due to collisions, due to their priority.

[2] Message will always be dropped if program trace is enabled, and program correlation messages for PID0 /mtmsr IS messages are not masked (Event Code = 0101). No error message is sent for this case since the PID value is contained in the higher priority message.

14.7.5 Data Acquisition Message Priority Loss Response

If a data acquisition message (DQM) loses arbitration due to contention with higher priority messages, an error message is generated to indicate that a DQM has been lost due to contention.

14.7.6 Ownership Trace Message Priority Loss Response

The two different types of ownership trace messages (OTMs) have different priorities and message dropped responses. If an OTM is because of software updates to the process ID, an error message is generated if the OTM loses arbitration due to contention with higher priority messages—except for a program correlation message with EVCODE = 0101 (PID or MSR[IS] update). If the pending OTM is a periodic update and loses arbitration, the event is dropped without generating an error message.

14.7.7 Program Trace Message Priority Loss Response

An error message is generated to indicate that branch trace information has been lost if a program trace message (PTM) loses arbitration due to contention with higher priority messages and the discarded PTM is one of the following:

- A program correlation message
- A resource full message for instruction count or history buffer
- A branch trace message

The next branch trace message is upgraded to a sync-type message.

If the discarded PTM is a program correlation message with PID information (EVCODE = 0101), the error message indicates a dropped OTM and a dropped program trace (error code = xxxx11xx).

14.7.8 Data Trace Message Priority Loss Response

If a data trace message (DTM) loses arbitration due to contention with higher priority messages, the DTM event is discarded and the next DTM is upgraded to a sync-type message.

14.8 Debug Status Messages

Debug status messages report low power mode and debug status. Debug status messages are enabled when Nexus 3 is enabled. Entering/exiting debug mode as well as entering, exiting, or changing low power mode(s) trigger a debug status message, indicating the value of the most significant byte in the development status register. Debug status information is sent out in the format shown in Figure 14-23:

(8 bits)	(4 bits)	(6 bits)
DS[31–24]	Src. Proc.	TCODE (000000)

Fixed length = 18 bits

Figure 14-23. Debug Status Message Format

14.9 Error Messages

Error messages are enabled whenever the debug logic is enabled. There are two conditions that produce an error message, each receiving a separate error type designation:

- A message is discarded due to contention with other (higher priority) message types. These errors have an Error Type value of 1.
- The message queue overruns. After the queue is drained, an error message is enqueued with an error code that indicates what types of messages were discarded during the interim. These errors have an Error Type value of 0.

NOTE

The OVCR register can be used in order to alleviate potential overrun situations.

Error information is messaged out in the format shown in Figure 14-24 (see also Table 14-3 and Table 14-4).

(6 bits)	(4 bits)	(4 bits)	(6 bits)
Error Code	Error Type	Src. Proc.	TCODE (001000)

Fixed length = 20 bits

Figure 14-24. Error Message Format

14.10 Ownership Trace

This section describes the ownership trace features of the Nexus 3 module.

14.10.1 Overview

Ownership trace provides a macroscopic view, such as task flow reconstruction, when debugging software written in a high-level (or object-oriented) language. It offers the highest level of abstraction for tracking operating system software execution. This is especially useful when the developer is not interested in debugging at lower levels.

14.10.2 Ownership Trace Messaging (OTM)

Ownership trace information is messaged via the auxiliary port using an ownership trace message (OTM). e200 processors contain a Power ISA embedded category “Process ID” register within the CPU. It is updated by the operating system software to provide task/process ID information. The contents of this register are replicated on the pins of the processor and connected to Nexus. The process ID register value can be accessed using the **mfspr**/**mtspr** instructions.

NOTE

> The CPU includes a process ID register (PID0), and therefore, the Nexus UBA functionality is not implemented.

There are two conditions that cause an ownership trace message when ownership trace is enabled, as follows:

- When new information is updated in the PID0 register by the e200 processor, the data is latched within Nexus. It is messaged out via the auxiliary port, allowing development tools to trace ownership flow. However, if program trace is enabled and program correlation messages for PID0 /**mtmsr** IS messages are not masked (Event Code = 0101), an OTM is not generated for an update to the PID0 register because the program correlation message provides this PID0 update information.
- Periodically, at least once every 256 messages, the most recent state of the PID0 register is messaged out. The resulting OTM indicates in the PID index subfield that PID0 status is being reported. The most recent value of the PID0 register is conveyed in the process ID value subfield. These periodic OTM events can be disabled by setting DC1[POTD].

Ownership trace information is messaged out in the format shown in Figure 14-25:

(1–8 bits)	(4 bits)	(4 bits)	(6 bits)
Process ID	PID Index (0000)	Src. Proc.	TCODE (000010)

Variable length = 15-22 bits

Figure 14-25. Ownership Trace Message Format

14.11 Program Trace

This section details the program trace mechanism supported by Nexus3 for the e200 processor. Program trace is implemented via branch trace messaging (BTM) as per the IEEE-ISTO 5001 standard definition. Branch trace messaging for e200 processors is accomplished by snooping the e200 virtual address bus (between the CPU and MMU), attribute signals, and CPU Status (**p_mode[0:3]**, **p_pstat_pipe{0,1}[0:5]**).

14.11.1 Branch Trace Messaging Types

Traditional branch trace messaging facilitates program trace by providing the following types of information:

- Messaging for taken direct branches includes how many sequential instructions were executed since the last taken branch or exception, including the taken direct branch. Branch instructions are included in the count of sequential instructions.
- Messaging for taken indirect branches and exceptions includes how many sequential instructions were executed since the last taken branch or exception and the unique portion of the branch target address or exception vector address. Branch instructions are included in the count of sequential instructions. For taken indirect branches which trigger generation of a message, the branch is also included in the count.

  Messaging for taken indirect branches and exceptions also include the newly established value of MSR[IS] in the MAP field if the indirect branch message is due to an exception or **rfi**, **rfci**, **rfdi**, or **rfmci** class instruction. For all other indirect branches, the MAP field reflects the current value of MSR[IS].

Branch history messaging facilitates program trace by providing the following information in messaging for taken indirect branches and exceptions:

- How many sequential instructions (I-CNT) were executed since the last predicate instruction, taken/not taken direct branch, taken/not-taken indirect branch, or exception
- The unique portion of the branch target address or exception vector address
- A branch/predicate instruction history field

  Each bit in the history field represents a direct branch or predicated instruction where a value of one indicates taken and a value of zero indicates not taken. Certain instructions (**evsel**) generate a pair of predicate bits that are both reported as consecutive bits in the history field. Not-taken indirect branches generate a history bit with a value of zero (0). Instructions that generate history bits are not included in instruction counts. For taken indirect branches that trigger generation of this message type, the branch is included in the count, but not in the history field.

  Messaging for taken indirect branches and exceptions also include the newly established value of MSR[IS] in the MAP field if the indirect branch message is due to an exception or **rfi**, **rfci**, **rfdi**, or **rfmci** class instruction. For all other indirect branches, the MAP field reflects the current value of MSR[IS].

14.11.1.1 e200 Indirect Branch Message Instructions

Table 14-30 shows the types of instructions and events that cause indirect branch messages or branch history messages to be encoded.

Table 14-30. Indirect Branch Message Sources

Source of Indirect Branch Message	Instructions/Detail
Taken branch relative to a register value	**bcctr**, **bcctrl**, **bclr**, **bclrl, se_bctr, se_bctrl, se_blr, se_blrl**
System Call/Trap exceptions taken	**sc**, **se_sc, tw**, **twi**

Table 14-30. Indirect Branch Message Sources (continued)

Source of Indirect Branch Message	Instructions/Detail
Return from interrupts/exceptions	**rfi**, **rfci**, **rfdi, se_rfi, se_rfci, se_rfdi**
Exit from reset with Program Trace Enabled	Indirect branch with Sync, target address is initial instruction, count = 1

14.11.1.2 e200 Direct Branch Message Instructions

Table 14-31 shows the types of instructions that cause direct branch messages or toggle a bit in the instruction history buffer to be messaged out in a resource full message or branch history message.

Table 14-31. Direct Branch Message Sources

Source of Direct Branch Message	Instructions
Taken direct branch instructions Instruction Synchronize	**b**, **ba**, **bl**, **bla**, **bc**, **bca**, **bcl**, **bcla, se_b. se_bc, se_bl, e_b, e_bc, e_bl, e_bcl, isync, se_isync**

14.11.1.3 BTM Using Branch History Messages

Traditional BTM messaging can accurately track the number of sequential instructions between branches, but cannot accurately indicate which instructions were conditionally executed and which were not.

Branch history messaging solves this problem by providing a predicated instruction history field in each indirect branch message. Each bit in the history represents a predicated instruction or direct branch, or a not-taken indirect branch. A value of one indicates the conditional instruction was executed or the direct branch was taken. A value of zero indicates the conditional instruction was not executed or the branch was not taken. Certain instructions (**evsel**) generate a pair of predicate bits which are both reported as consecutive bits in the history field.

Branch history messages solve predicated instruction tracking and save bandwidth since only indirect branches cause messages to be queued.

14.11.1.4 BTM using Traditional Program Trace Messages

Based on the PTM bit in the DC1 register, program tracing can utilize either branch history messages (PTM = 1) or traditional direct/indirect branch messages (PTM = 0).

Branch history saves bandwidth and keeps consistency between methods of program trace, yet may lose temporal order between BTM messages and other types of messages. Because direct branches are not messaged but are instead included in the history field of the indirect branch history message, other types of messages may enter the FIFO between branch history messages. The development tool cannot determine the ordering of events that occurred with respect to direct branches simply by the order in which messages are sent out.

Traditional BTM messages maintain their temporal ordering because each event that can cause a message to be queued enters the FIFO in the order it occurred and will be messaged out maintaining that order.

14.11.2 BTM Message Formats

The Nexus 3 block supports three types of traditional BTM messages: direct, indirect, and synchronization messages. It supports two types of branch history BTM messages: indirect branch history and indirect branch history with synchronization messages.

14.11.2.1 Indirect Branch Messages (History)

Indirect branches include all taken branches whose destination is determined at run time, interrupts, and exceptions. If DC1[PTM] is set, indirect branch information is messaged out in the format shown in Figure 14-26.

(1–32 bits)	(1–32 bits)	(1–8 bits)	(1 bit)	(4 bits)	(6 bits)
Branch History	Relative Address	Sequence Count	Inst Space	Source Proc.	TCODE (011100)

Max length = 83 bits; Min length = 14 bits

Figure 14-26. Indirect Branch Message (History) Format

14.11.2.2 Indirect Branch Messages (Traditional)

If DC1[PTM] is cleared, indirect branch information is messaged out in the format shown in Figure 14-27.

(1–32 bits)	(1–8 bits)	(1 bit)	(4 bits)	(6 bits)
Relative Address	Sequence Count	Inst Space	Source Proc.	TCODE (000100)

Max length = 51 bits; Min length = 13 bits

Figure 14-27. Indirect Branch Message Format

14.11.2.3 Direct Branch Messages (Traditional)

Direct branches (conditional or unconditional) are all taken branches whose destination is fixed in the instruction opcode. Direct branch information is messaged out as shown in Figure 14-28.

(1–8 bits)	(4 bits)	(6 bits)
Sequence Count	Src. Proc.	TCODE (000011)

Max length = 18 bits; Min length = 11bits

Figure 14-28. Direct Branch Message Format

NOTE

When DC1[PTM] is set, direct branch messages are not transmitted. Instead, each direct branch, not-taken indirect branch, or predicated instruction is recorded in the history buffer.

14.11.3 Program Trace Message Fields

The following subsections describe specific fields used for program trace messages.

14.11.3.1 Sequential Instruction Count Field (ICNT)

Most of the program trace messages include an instruction count field. For traditional branch messages, ICNT represents the number of sequential instructions including non-taken branches since the last direct/indirect branch messages. Branch instructions that trigger message generation are included in the ICNT.

For branch history messages, ICNT represents the number of instructions executed since the last taken/non-taken direct branch, predicate instruction, last taken/not-taken indirect branch, or exception. Branch instructions that trigger message generation are included in the ICNT. Instructions that generate history bits are not included in the ICNT.

The sequential instruction counter overflows after its value reaches 255 and is reset to 0. In addition, the next BTM message (corresponding to the 256th or later instruction) is converted to a synchronization type message.

The instruction counter is reset every time the instruction count is transmitted in a message or whenever there is a branch/predicate history event, as well as on exiting from debug mode.

14.11.3.2 Branch/Predicate Instruction History (HIST)

If DC1[PTM] is set, BTM messaging uses the branch history format. The branch history (HIST) field in these messages provides a history of branch execution used for reconstructing the program flow. The branch/predicate history buffer stores information about branch and predicate instruction execution. The buffer is implemented as a left-shifting register. The buffer is preloaded with a one that acts as a stop bit (the most significant 1 in the history field is a termination bit for the field). The preloaded bit itself is not part of the history, but is transmitted with the packet.

A value of one is shifted into the history buffer for each taken direct branch (program counter relative branch) or predicate instruction whose condition evaluates to true. A value of zero is shifted into the history buffer for each not-taken branch (including indirect branch instructions) or predicate instruction whose condition evaluates to false. For the **evsel** instruction, two bits are shifted in, corresponding to the low element (shifted in first) and the high element (shifted in second) conditions.

This history buffer information is transmitted as part of an indirect branch with history message, as part of a program correlation message, or as part of a resource full message if the history buffer becomes full. The history buffer is reset every time the history information is transmitted in a message, as well as on exiting from debug mode.

Table 14-32 shows the branch/predicate history events.

Table 14-32. Branch/Predicate History Events

Branch/Predicate History Event	History Bit(s)	Relevant Instructions
Not taken register indirect branches	0	bcctr, bcctrl, bclr, bclrl
Not taken direct branches	0	b, ba, bc, bca, bla, bcla, bl, bcl
Taken direct branches	1	b, ba, bc, bca, bla, bcla, bl, bcl[1]
evsel instruction	00,01,10, or 11	evsel

[1] If the EVCODE for direct branch function calls is not masked in DC4, taken **bl** and **bcl** instructions will generate Program Correlation Messages and will not be logged in the history buffer.

14.11.3.3 Execution Mode Indication

In order for a development tool to properly interpret instruction count and history information, it must be aware of the execution mode context of that information. VLE instructions are interpreted differently from non-VLE instructions.

Program trace messages provide the execution mode status in the least significant bit of the **reconstructed** address field. A value of 0 indicates that preceding instruction count and history information should be interpreted in a non-VLE context. A value of 1 indicates that the preceding instruction count and history information should be interpreted in a VLE context. Note that when a branch results in an execution mode switch, the program trace message resulting from that branch will indicate the previous execution state. The new state will not be signaled until the next program trace message.

In some cases, a program correlation message is generated to indicate execution mode status. Refer to Section 14.11.3.5, “Program Correlation Messages,” for more information on these cases.

14.11.3.4 Resource Full Messages

The resource full message is used in conjunction with branch trace and branch history messages. The resource full message is generated when either the internal branch/predicate history buffer is full or if the BTM Instruction sequence counter (I-CNT) overflows. If synchronization is needed at the time this message is generated, the synchronization is delayed until the next branch trace message that is not a resource full message.

For history buffer overflow, the resource full message transmits a resource code (RCODE) of 0b0001 and the current contents of the history buffer, including the stop bit, are transmitted in the resource data (RDATA) field. This history information can be concatenated by the development tool with the branch/predicate history information from subsequent messages to obtain the complete branch/predicate history between indirect changes of flow.

For instruction counter overflow, the resource full message transmits an RCODE of 0b0000 and a value of 0xFF is transmitted in the RDATA field, indicating that 255 sequential instructions have been executed since the last change of flow or if program trace is in history mode, since the last instruction which recorded history information.

Figure 14-29 shows the resource full message format.

(1-32 bits)	(4 bits)	(4 bits)	(6 bits)
RDATA	RCODE	Src. Proc.	TCODE (011011)

Max length = 46 bits; Min length = 15 bits

Figure 14-29. Resource Full Message Format

Table 14-33 shows the RCODE encodings and RDATA information used for Resource Full messages.

Table 14-33. RCODE Encoding

RCODE	Description	RDATA field
0000	Program Trace Instruction counter reached 255 and was reset.	0xFF
0001	Program Trace, Branch/Predicate Instruction History full.	Branch HIstory. This type of packet is terminated by a stop bit set to 1 after the last history bit.

14.11.3.5 Program Correlation Messages

Program correlation messages (PCMs) are used to correlate events to the program flow that may or may not be associated with the instruction stream. The following events result in a PCM when program trace is enabled:

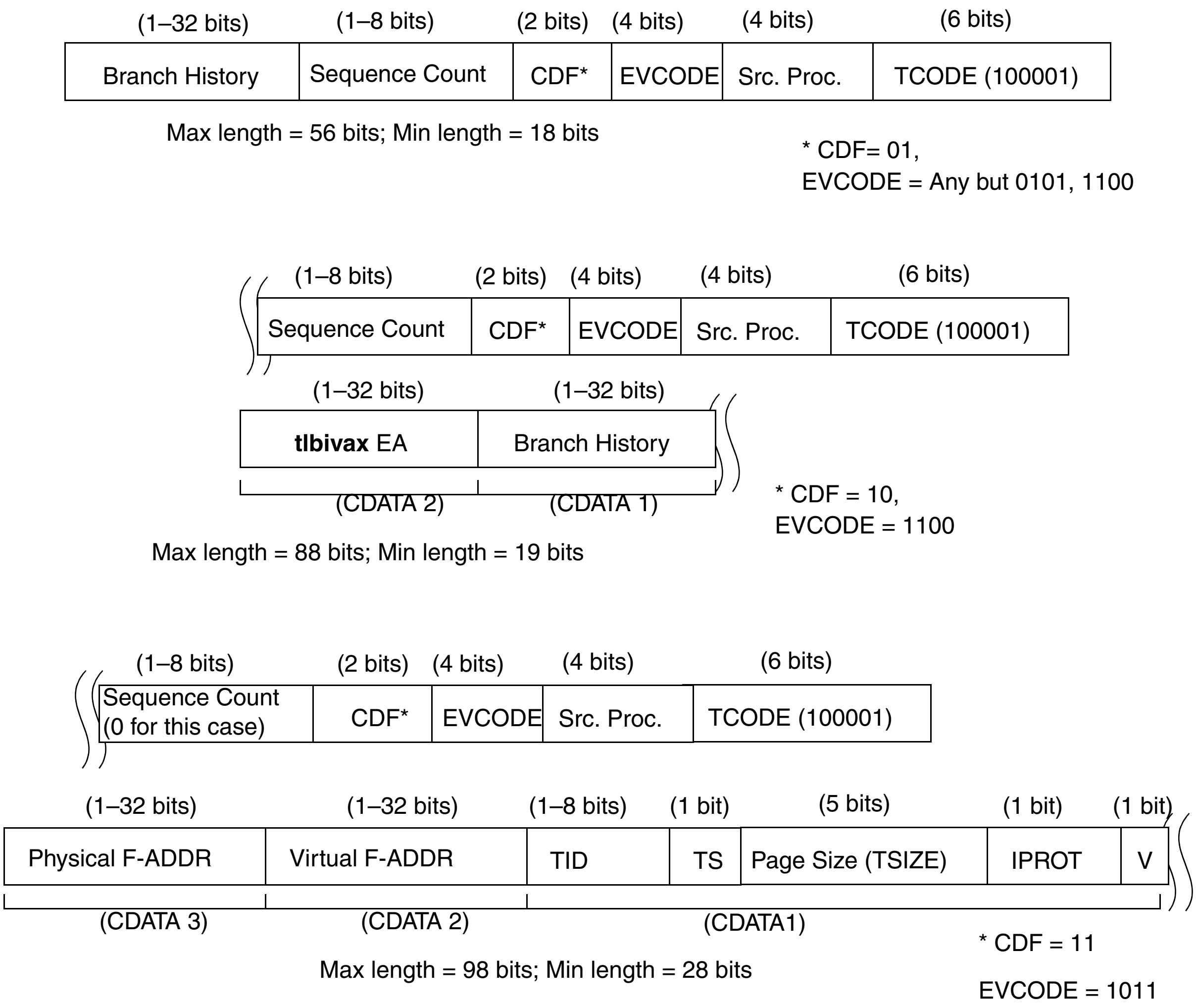
- When the CPU enters debug mode, a PCM is generated. The instruction count and history information provided by the PCM can be used to determine the last sequence of instructions executed prior to debug mode entry.
- When the CPU first enters a low power mode in which instructions are no longer executed, a PCM is generated. The instruction count and history information provided by the PCM can be used to determine the last sequence of instructions executed prior to low power mode entry.
- Whenever program trace is disabled by any means, a PCM is generated. The instruction count and history information provided by the PCM can be used to determine the last sequence of instructions executed prior to disabling program trace. A second PCM is generated on this event if there has been an execution mode switch into or out of a sequence of VLE instructions. This VLE state information allows the development tool to interpret any preceding instruction count or history information in the proper context.
- When a “Branch and Link” instruction executes (direct branch function call - **bl**/**bcl**/**bla**/**bla**-type instructions)
- Whenever the CPU crosses a page boundary that results in an execution mode switch into or out of a sequence of VLE instructions, a PCM is generated. The PCM effectively breaks up any running instruction count and history information between the two modes of operation so that the instruction count and history information can be processed by the development tool in the proper context.
- When using program trace in history mode, when a direct branch results in an execution mode switch into or out of a sequence of VLE instructions, a PCM is generated. The PCM effectively

breaks up any running history information between the two modes of operation so that the history information can be processed by the development tool in the proper context.

- When program trace becomes masked due to MSR[PMM] = 0 and DC4[PTMARK] = 1.
- When a new address translation is established in the TLB via a **tlbwe** instruction.
- When address translation(s) are invalidated in the TLB via a **tlbivax** instruction.
- When a new instruction address space setting (IS) is established in the MSR via a **mtmsr** instruction.
- When an update to the process ID register (PID0) is made via a **mtspr** PID0.

Refer to Table 14-6 for the event codes that are supported in this implementation. Event code masking is available via the EVCDM field of the DC4 register to allow for control over generation of program correlation messages for each event type.

Figure 14-30 shows the program correlation message formats.



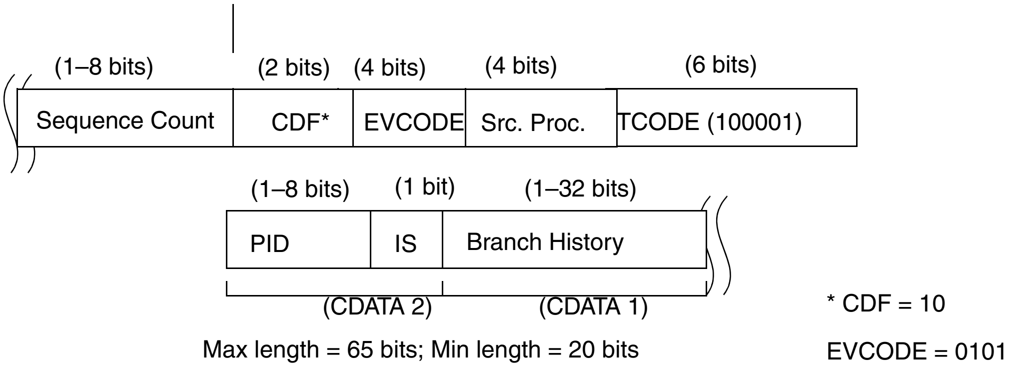


Figure 14-30. Program Correlation Message Formats

14.11.3.6 Program Correlation Message Generation for TLB Update with New Address Translation

When a new address translation is established in the TLB, a Program Correlation message is generated containing the information regarding the new TLB entry using EVCODE = 1011. A PCM with current history and instruction count is also generated using EVCODE = 1011 (unless collapsed with a different EVCODE) and sent just prior to sending the PCM containing the newly established address translation. The messages are provided so that the address translation information can be processed by the development tool in the proper program flow.

14.11.3.7 Program Correlation Message Generation for TLB Invalidate (tlbivax) Operations

When a tlbivax is executed to invalidate one or more entries in the TLB, a Program Correlation message is generated containing the information regarding the **tlbivax** EA used for invalidation using EVCODE = 1100. The current history and instruction count (which includes the **tlbivax** instruction) is also included in the message. The messages are provided so that the address translation information can be processed by the development tool in the proper program flow.

14.11.3.8 Program Correlation Message Generation for PID Updates or MSR[IS] Updates

When a (potentially) new value is established in the PID via a **mtspr** PID0, a program correlation message is generated containing the information regarding the new PID0 value. This PCM also contains the current history and instruction count and the current value of MSR[IS]. The message is provided so that address translation information can be processed by the development tool in the proper program flow. The **mtspr** PID0 is included in the instruction count information. Note that ownership trace messages (other than the periodic OTM) are redundant with the information provided and may be disabled to avoid unnecessary message bandwidth or collisions.

When a new value is established in MSR[IS] via a **mtmsr** instruction, a Program Correlation message is generated containing the information regarding the new MSR[IS] value. This PCM also contains the current history and instruction count, and the current value of PID0. The message is provided so that address translation information can be processed by the development tool in the proper program flow. The **mtmsr** instruction is included in the instruction count information.

14.11.3.9 Program Trace Overflow Error Messages

An error message occurs when a new message cannot be queued due to the message queue being full. The FIFO discards incoming messages until it has completely emptied the queue. Once emptied, an error message is queued. The error encoding indicates which type(s) of messages attempted to be queued while the FIFO was being emptied.

14.11.3.10 Program Trace Synchronization Messages

By default, program trace messages perform XOR compression on the branch target address to produce the address field for the message. This compression is consistent with the specification in IEEE-ISTO 5001.

Under some conditions an uncompressed address is sent to provide development tools with a baseline reference address. A program trace direct/indirect branch with sync message is messaged via the auxiliary port (provided program trace is enabled) for the following conditions (see Table 14-34):

- Initial program trace message upon the first direct/indirect branch after exit from system reset or whenever program trace is enabled.
- Upon direct/indirect branch after returning from a CPU low power state
- Upon direct/indirect branch after returning from debug mode
- Upon direct/indirect branch after occurrence of queue overrun (can be caused by any trace message), provided program trace is enabled
- Upon direct/indirect branch after the periodic program trace counter has expired indicating 255 *without-sync* program trace messages have occurred since the last *with-sync* message occurred
- Upon direct/indirect branch after assertion of the Event In (**nex_evti_b**) pin if the EIC bits within the DC1 register have enabled this feature
- Upon direct/indirect branch after the sequential instruction counter has expired indicating 255 instructions have occurred since the last change of flow
- Upon direct/indirect branch after a BTM Message was lost due to a collision while attempting to enter the message queue
- Upon the first direct/indirect branch message after an execution mode switch into or out of a sequence of VLE instructions
- When program trace becomes unmasked due to MSR[PMM] → 1 with DC4[PTMARK] = 1.

Note that the ICNT and history information for the first message is not meaningful for some of these cases because the temporary masking of program trace may result in ambiguous values. Subsequent with sync messages do not have this issue.

The format for program trace direct/indirect branch with sync messages is as shown in Figure 14-31.

(1–32 bits)	(1–8 bits)	(1 bit)	(4 bits)	(6 bits)
Full Target Address	Sequence Count	Inst Space	Source Proc.	TCODE (001011 or 001100)

Max length = 51 bits; Min length = 13 bits

Figure 14-31. Direct/Indirect Branch with Sync Message Format

The format for program trace indirect branch history with sync messages is as shown in Figure 14-32:

(1–32 bits)	(1–32 bits)	(1–8 bits)	(1 bit)	(4 bits)	(6 bits)
Branch History	Full Target Address	Sequence Count	Inst Space	Source Proc.	TCODE (011101)

Max length = 83 bits; Min length = 14 bits

Figure 14-32. Indirect Branch History with Sync Message Format

Exception conditions that result in program trace synchronization are summarized in Table 14-34.

Table 14-34. Program Trace Exception Summary

Exception Condition	Exception Handling
System Reset Negation	At the negation of JTAG reset (**j_trst_b**), queue pointers, counters, state machines, and registers within the Nexus 3 module are reset. Upon exiting system reset, if program trace is already enabled), a program trace message is sent as an indirect branch with synchronization message.
Program Trace Enabled	The first program trace message (after program trace has been enabled) is a synchronization message.
Exit from Low Power/Debug	Upon exit from a low power mode or debug mode the next direct/indirect branch will be converted to a direct/indirect branch with sync. message.
Queue Overrun	An Error Message occurs when a new message cannot be queued due to the message queue being full. The FIFO will discard messages until it has completely emptied the queue. Once emptied, an Error Message will be queued. The error encoding will indicate which type(s) of messages attempted to be queued while the FIFO was being emptied. The next BTM message in the queue will be a Direct/Indirect Branch w/ Sync. Message.
Periodic Program Trace Sync.	A forced synchronization occurs periodically after 255 non-sync Program Trace Messages have been queued. A Direct/Indirect Branch w/ Sync. Message is queued. The periodic program trace message counter then resets.
Event In	If the Nexus module is enabled, a **nex_evti_b** assertion initiates a Direct/Indirect Branch w/ Sync. Message upon the next direct/indirect branch (if Program Trace is enabled and the EIC bits of the DC1 Register have enabled this feature).
Sequential Instruction Count Overflow	After the sequential instruction counter reaches its maximum count (up to 255 sequential instructions may be executed), a forced synchronization occurs. The sequential counter then resets. A Program Trace Direct/Indirect Branch w/ Sync.Message is queued upon execution of the next branch. A Resource Full Message is Queued on the overflow event. If a branch instruction is the 255th instruction to occur, and causes a Program Trace message to be queued, then no Resource Full Message is queued, and the w/Sync message will be queued for the *next* Program Trace Direct/Indirect Branch Message.

Table 14-34. Program Trace Exception Summary (continued)

Exception Condition	Exception Handling
Collision Priority	All Messages have the following priority: Instruction 0 (WPM → DQM → PCM[PIDMSG] → OTM → BTM → DTM)→ Instruction1 (WPM → DQM → OTM → BTM → DTM), where instruction0 is the oldest instruction. A BTM Message from Instruction1 which attempts to enter the queue at the same time as three higher priority messages from either instruction will be lost. An Error Message will be sent indicating the BTM was lost. The following direct/indirect branch will queue a Direct/Indirect Branch w/ Sync. Message. The count value within this message will reflect the number of sequential instructions executed after the last successful BTM Message was generated. This count will include the branch which did not generate a message due to the collision.
Execution Mode Switch	Whenever the CPU switches execution mode into or out of a sequence of VLE instructions, the next branch trace message will be a Direct/Indirect Branch w/ Sync Message.

14.11.4 Enabling Program Trace

Program trace messaging can be enabled in the following ways:

- Setting the TM field of the DC1 Register to enable program trace
- Using the PTS field of the WT Register to enable program trace on watchpoint hits (e200 watchpoints are configured within the CPU)
- Filtering of Program Trace messages may be performed using MSR[PMM] and the setting of DC4[PTMARK]

14.11.5 Program Trace Timing Diagrams (2 MDO/1 MSEO Configuration)

This section contains program trace timing diagrams for 2MDO/1MSEO configuration.

Figure 14-33 shows the program trace for the traditional indirect branch message.

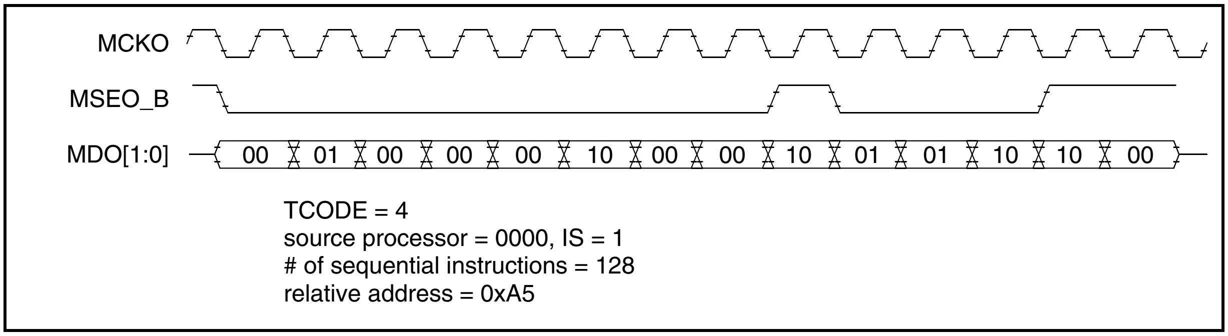


Figure 14-33. Program Trace—Indirect Branch Message (Traditional)

Figure 14-34 shows the program trace for the history indirect branch message.

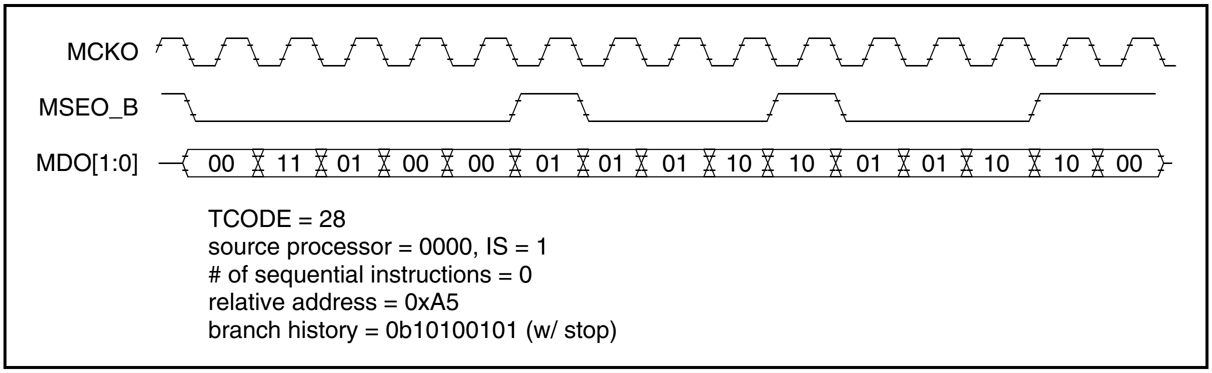


Figure 14-34. Program Trace—Indirect Branch Message (History)

Figure 14-35 shows the program trace for the traditional direct branch and error message.

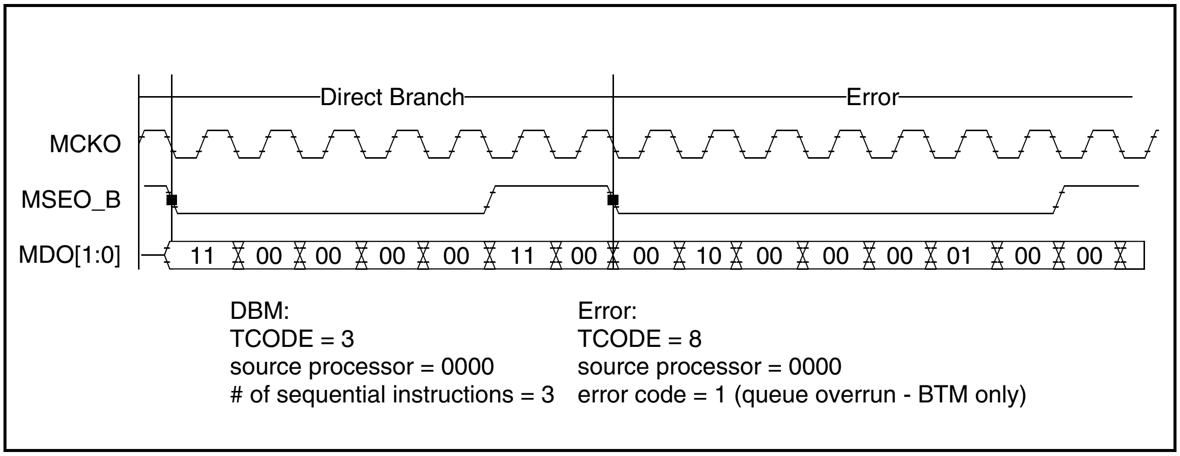


Figure 14-35. Program Trace—Direct Branch (Traditional) and Error Messages

Figure 14-36 shows the program trace for the indirect branch with sync message.

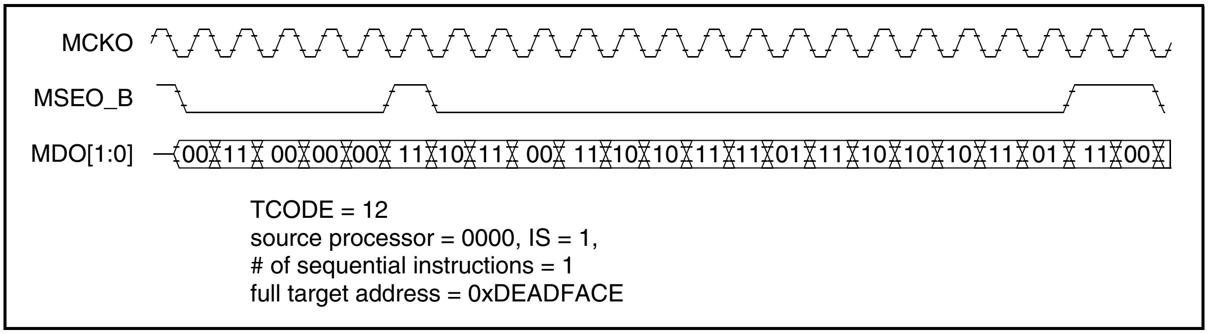


Figure 14-36. Program Trace—Indirect Branch with Sync Message

14.12 Data Trace

This section deals with the data trace mechanism supported by the Nexus 3 module. Data trace is implemented via data write messaging (DWM) and data read messaging (DRM), as per the IEEE-ISTO 5001 standard.

14.12.1 Data Trace Messaging (DTM)

Data trace messaging for e200 is accomplished by snooping the e200 address and internal data buses and storing the information for qualifying accesses (based on enabled features and matching target addresses). The Nexus 3 module traces all data access that meet the selected range and attributes.

NOTE

> Data trace is only performed on the e200 internal data buses. This allows data visibility for e200 processors that incorporate a data cache. Only e200 CPU initiated accesses will be traced. No DMA accesses to the AHB system bus are traced.

Data trace messaging can be enabled in one of two ways.

- Setting the TM field of the DC1 register to enable data trace.
- Using the DTS field of the WT register to enable data trace on watchpoint hits (e200 watchpoints are configured within the Nexus1 module).

14.12.2 DTM Message Formats

The Nexus 3 block supports five types of DTM messages, as follows:

- Data write
- Data read
- Data write synchronization
- Data read synchronization
- Error messages

14.12.2.1 Data Write Messages

The data write message contains the data write value and the address of the write access, relative to the previous data trace message. Data write message information is messaged out in the following format:

(1–64 bits)	(1–32 bits)	(4 bits)	(1 bit)	(4 bits)	(6 bits)
Data Value(s)*	Relative Address	Data Size	Data Space	Src. Proc.	TCODE (000101)

Max length = 111 bits; Min length = 17 bits

Figure 14-37. Data Write Message Format

14.12.2.2 Data Read Messages

The data read message contains the data read value and the address of the read access, relative to the previous data trace message. Data read message information is messaged out in the following format:

(1–64 bits)	(1–32 bits)	(4 bits)	(1 bit)	(4 bits)	(6 bits)
Data Value(s)*	Relative Address	Data Size	Data Space	Src. Proc.	TCODE (000110)

Max length = 111 bits; Min length = 17 bits

Figure 14-38. Data Read Message Format

NOTE

*e200-based CPUs are capable of generating two (2) reads or writes per clock cycle in cases where multiple registers are accessed with a single instruction (lmw/stmw). These have a double word pair size encoding (**p_tsiz** = 0b000). In these cases, the Nexus 3 module will send one (1) data trace message with the two 32-bit data values as one combined 64-bit value for each message.

For e200 based CPUs, the double word encoding (**p_tsiz** = 0b000) may also indicate a double word access and is sent out as a single data trace message with a single 64-bit data value.

The debug/development tool needs to distinguish the two cases based on the family of the e200 processor.

14.12.2.3 Data Trace Synchronization Messages

A data trace write/read with synchronization message is messaged via the auxiliary port (provided data trace is enabled) for the following conditions (see Table 14-35):

- Initial data trace message after exit from system reset or whenever data trace is enabled
- Upon returning from a CPU low power state
- Upon returning from debug mode
- After occurrence of queue overrun (can be caused by any trace message), provided data trace is enabled
- After the periodic data trace counter has expired indicating 255 *without-sync* data trace messages have occurred since the last *with-sync* message occurred
- Upon assertion of the Event In (**nex_evti_b**) pin, the first data trace message is a synchronization message if the EIC bits of the DC1 register have enabled this feature
- Upon data trace write/read after the previous DTM message was lost due to an attempted access to a secure memory location (for SoCs with security).
- Upon data trace write/read after the previous DTM message was lost due to a collision entering the FIFO between the DTM message and any two of the following: watchpoint message, ownership trace message, or program trace message.

Data trace synchronization messages provide the full address (without leading zeros) and insure that development tools fully synchronize with data trace regularly. Synchronization messages provide a reference address for subsequent DTMs, in which only the unique portion of the data trace address is transmitted. The format for data trace write/read with synchronization messages are as follows:

(1–64 bits)	(1–32 bits)	(4 bits)	(1 bit)	(4 bits)	(6 bits)
Data Value	Full Address	Data Size	Data Space	Source Proc.	TCODE (001101 or 001110)

Max length = 111 bits; Min length = 17 bits

Figure 14-39. Data Write/Read with Synchronization Message Format

Exception conditions that result in data trace synchronization are summarized in Table 14-35.

Table 14-35. Data Trace Exception Summary

Exception Condition	Exception Handling
System Reset Negation	At the negation of JTAG reset (**j_trst_b**), queue pointers, counters, state machines, and registers within the Nexus 3 module are reset. If data trace is enabled, the first data trace message is a data write/read with synchronization message.
Data Trace Enabled	The first data trace message (after data trace has been enabled) is a synchronization message.
Exit from Low Power/Debug	Upon exit from a low power mode or debug mode the next data trace message will be converted to a data write/read with synchronization message.
Queue Overrun	An error message occurs when a new message cannot be queued due to the message queue being full. The FIFO discards messages until it has completely emptied the queue. Once emptied, an error message is queued. The error encoding indicates which type(s) of messages attempted to be queued while the FIFO was being emptied. The next DTM message in the queue will be a data write/read with synchronization message.
Periodic Data Trace Sync.	A forced synchronization occurs periodically after 255 data trace messages have been queued. A data write/read with synchronization message is queued. The periodic data trace message counter then resets.
Event In	If the Nexus module is enabled, a **nex_evti_b** assertion initiates a data trace write/read with synchronization. Message upon the next data write/read (if data trace is enabled and the EIC bits of the DC1 register have enabled this feature).
Attempted Access to Secure Memory	For SoCs that implement security, any attempted read or write to secure memory locations temporarily disables data trace and causes the corresponding DTM to be lost. A subsequent read/write queues a data trace read/write with synchronization message.
Collision Priority	All messages have the following priority: Instruction 0 (WPM → DQM → PCM[PIDMSG] → OTM → BTM → DTM)→ Instruction1 (WPM → DQM → OTM → BTM → DTM), where instruction 0 is the oldest instruction. A DTM message that attempts to enter the queue at the same time as three other higher priority messages will be lost. A subsequent read/write queues a data trace read/write with synchronization message.

14.12.3 DTM Operation

This section contains the following subsections:

- Section 14.12.3.1, “Data Trace Windowing”
- Section 14.12.3.2, “Data Access/Instruction Access Data Tracing”

- Section 14.12.3.3, “Data Trace Filtering”
- Section 14.12.3.4, “e200 Bus Cycle Special Cases”

14.12.3.1 Data Trace Windowing

Data write/read messages are enabled via the RWT field in the data trace control register (DTC) for each DTM channel. Data trace windowing is achieved via the address range defined by the DTEA and DTSA registers and by the RC field in the DTC register. All e200 initiated read/write accesses that fall inside or outside these address ranges, as programmed, are candidates to be traced.

14.12.3.2 Data Access/Instruction Access Data Tracing

The Nexus3 module is capable of tracing either instruction access data or data access data and can be configured for either type of data trace by setting the DI1 field within the data trace control register. This setting applies to all DTM channels.

14.12.3.3 Data Trace Filtering

Data trace filtering is available based on the settings of MSR[PMM] and DC4[DTMARK].

14.12.3.4 e200 Bus Cycle Special Cases

Table 14-36 describes the e200 bus cycle cases.

Table 14-36. e200 Bus Cycle Cases

Special Case	Action
e200 bus cycle aborted	Cycle ignored
e200 bus cycle with data error ($\overline{\text{TEA}}$)[1]	Data Trace Message discarded
e200 bus cycle completed without error[1]	Cycle captured & transmitted
e200 (AHB) bus cycle initiated by Nexus 3	Cycle ignored
e200 bus cycle is an instruction fetch	Cycle selectively ignored based on DTC_{DI} setting
e200 bus cycle accesses misaligned data (across 64-bit boundary); both 1st & 2nd transactions within data trace range	1st and 2nd cycle captured and a single or a pair of DTM(s) is (are) transmitted (see Note)
e200 bus cycle accesses misaligned data (across 64-bit boundary); 1st transaction within data trace range; 2nd transaction out of data trace range	1st and 2nd cycle captured & a single or a pair of DTM(s) is (are) transmitted (see Note)
e200 bus cycle accesses misaligned data (across 64-bit boundary); 1st transaction within data trace range; 2nd transaction (regardless of within range or not) receives a bus error	Data Trace Message discarded
e200 bus cycle accesses misaligned data (across 64-bit boundary); 1st transaction out of data trace range; 2nd transaction within data trace range	1st and 2nd cycle captured and a single or a pair of DTM(s) is (are) transmitted (see Note)
e200 bus cycle accesses misaligned data (across 64-bit boundary); 1st transaction out of data trace range; 2nd transaction within range, receives a bus error	Data Trace Message discarded

[1] Buffering of stores in the CPU store buffer may generate a DTM prior to the actual memory access, regardless of an error termination condition from memory.

NOTE

For misaligned accesses (crossing 64-bit boundary), the access is broken into two accesses by the CPU. If either access is within the data trace range, a single DTM will be sent with a size encoding indicating the size of the original access (i.e. word), and the address indicating the original misaligned accesses, unless the misaligned access wraps over the end of a circular buffer when using the SPE2 specialized load or store with modify, mode = 1000 (circular addressing). In this case, since the two portions of the misaligned access are not contiguous, two DTMs are sent, one for each portion. The size encodings and the addresses of the DTMs will indicate the accessed bytes of data.

A store to the cache's store buffer within the data trace range may initiate a DTM message prior to completion of the actual memory access.

14.12.4 Data Trace Timing Diagrams (8 MDO/2 MSEO Configuration)

This section shows the data trace timing diagrams for the 8 MDO/2 MSEO configuration.

Figure 14-40 shows the data trace data write message.

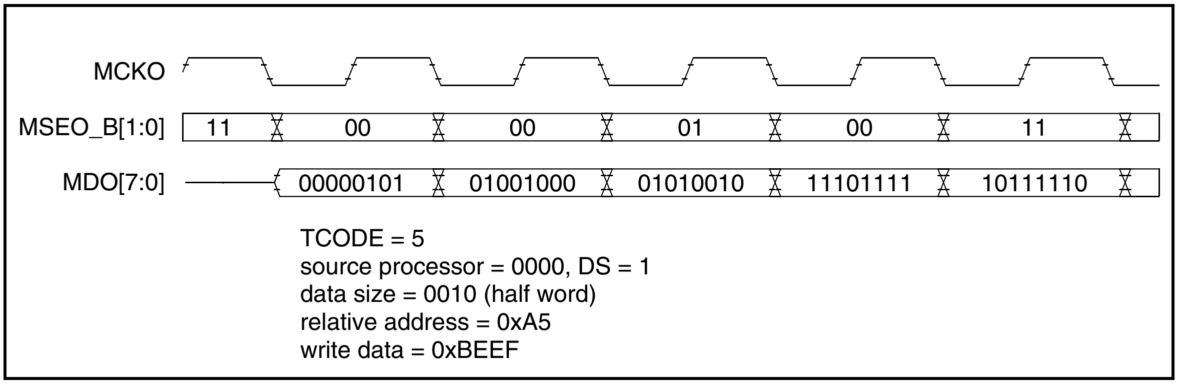
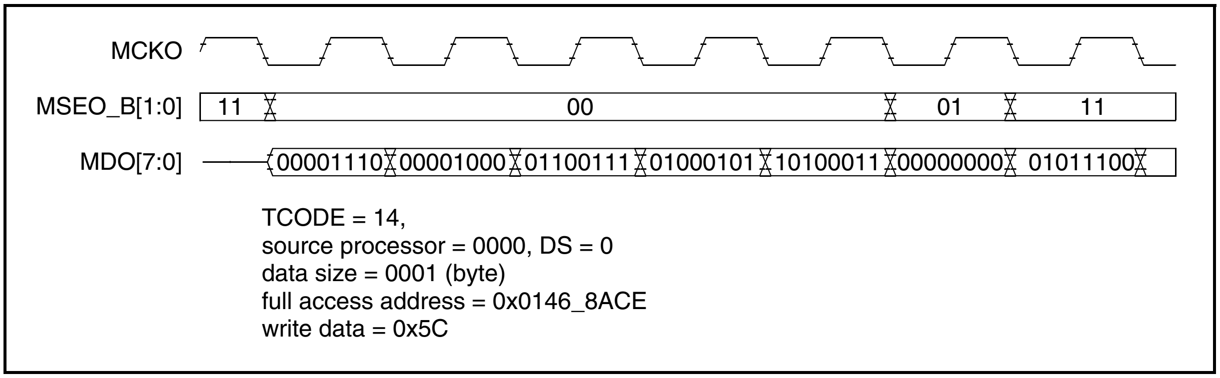


Figure 14-40. Data Trace—Data Write Message

Figure 14-41 shows the data trace data write message.



TCODE = 14,
source processor = 0000, DS = 0
data size = 0001 (byte)
full access address = 0x0146_8ACE
write data = 0x5C

Figure 14-41. Data Trace—Data Read with Sync Message

14.13 Data Acquisition Messaging

This section describes the data acquisition mechanisms supported by the e200z760n3 Nexus 3 module. Data acquisition trace is implemented using data acquisition trace messages in accordance with IEEE-ISTO 5001 definitions. The control mechanism to export the data is different from the recommendations of the standard, however.

Data acquisition trace provides a convenient and flexible mechanism for the debugger to observe the architectural state of the machine through software instrumentation.

14.13.1 Data Acquisition ID Tag Field

The DQTAG tag field (DQTAG) is an 8-bit value specifying control or attribute information for the data included in the data acquisition message. DQTAG is sampled from DEVENT[DQTAG] when a write to DDAM is performed via **mtspr** operations. The usage of the DQTAG is left to the discretion of the development tool to be used in whatever manner is deemed appropriate for the application.

14.13.2 Data Acquisition Data Field

The data acquisition data field (DQDATA) is the data captured from the DDAM write operation via **mtspr** operations. Leading zeros are omitted from the message.

14.13.3 Data Acquisition Trace Event

For DQM, a dedicated SPR has been allocated (DDAM). It is expected that the general use case is to instrument the software and use **mtspr** operations to generate data acquisition messages.

There is no explicit error response for failed accesses as a result of contention between an internal and external debugger. Software may be blocked or given ownership of DDAM and the DQTAG field of the DEVENT register via control in DBERC0 while in external debug mode. Hardware always has access to these registers. Refer to Section 13.3.4, “Debug External Resource Control Register (DBERC0),” for more detail on DBERC0.

Reads from the data acquisition channel do not generate a data acquisition event and return zeroes for the read data.

(1-32 bits)	(8 bits)	(4 bits)	(6 bits)
DQDATA	DQTAG	Src. Proc.	TCODE (011011)

Max length = 50 bits; Min length = 19 bits

Figure 14-42. Data Acquisition Message Format

14.14 Watchpoint Trace Messaging

Enabling watchpoint messaging is done by setting the watchpoint trace enable bit in the DC1 register. The e200 Nexus1 module and the performance monitor unit supports setting the individual watchpoint sources. The e200 Nexus1 module is capable of setting multiple types of watchpoints. Please refer to the debug chapter for details on watchpoint initialization.

When watchpoints occur due to one or more asserted watchpoint event signals and watchpoint trace messaging is enabled, a watchpoint trace message is sent to the message queue to be messaged out. This message includes the watchpoint number indicating which watchpoint(s) caused the message. If more than one enabled watchpoint occurs in a single cycle, only one watchpoint trace message is generated and multiple bits of the watchpoint hit field are set. The WMSK[WEM] settings control which watchpoints are enabled to generate watchpoint trace messages.

The occurrence of any of the e200 defined watchpoints can also be programmed to assert the Event Out (**nex_evto_b**) pin for one (1) period of the output clock (**nex_mcko**) based on settings in the DC2 and DC3 registers. See Table 14-39 for details on **nex_evto_b**.

Watchpoint information is messaged out as shown in Figure 14-43.

(1–27 bits)	(4 bits)	(6 bits)
Watchpoint Source	Src. Proc.	TCODE (001111)

Variable length = 11–37 bits

Figure 14-43. Watchpoint Message Format

Table 14-37 shows the watchpoint source encoding. The watchpoint source message field contains a 1 for each asserted watchpoint. Leading zeros are truncated.

Table 14-37. Watchpoint Source Encoding

Watchpoint Source (1–27 bits)	Watchpoint Description
00000000000000000000000000	No Watchpoints enabled for watchpoint trace messaging
XXXXXXXXXXXXXXXXXXXXXXXXXX1	Watchpoint #0 enabled for WTM
XXXXXXXXXXXXXXXXXXXXXXXXX1X	Watchpoint #1 enabled for WTM
XXXXXXXXXXXXXXXXXXXXXXXX1XX	Watchpoint #2 enabled for WTM
XXXXXXXXXXXXXXXXXXXXXXX1XXX	Watchpoint #3 enabled for WTM
XXXXXXXXXXXXXXXXXXXXXX1XXXX	Watchpoint #4 enabled for WTM
XXXXXXXXXXXXXXXXXXXXX1XXXXX	Watchpoint #5 enabled for WTM
XXXXXXXXXXXXXXXXXXXX1XXXXXX	Watchpoint #6 enabled for WTM
XXXXXXXXXXXXXXXXXXX1XXXXXXX	Watchpoint #7 enabled for WTM
XXXXXXXXXXXXXXXXXX1XXXXXXXX	Watchpoint #8 enabled for WTM
XXXXXXXXXXXXXXXXX1XXXXXXXXX	Watchpoint #9 enabled for WTM
XXXXXXXXXXXXXXXX1XXXXXXXXXX	Watchpoint #10 enabled for WTM
XXXXXXXXXXXXXXX1XXXXXXXXXXX	Watchpoint #11 enabled for WTM
XXXXXXXXXXXXXX1XXXXXXXXXXXX	Watchpoint #12 enabled for WTM
XXXXXXXXXXXXX1XXXXXXXXXXXXX	Watchpoint #13 enabled for WTM
XXXXXXXXXXXX1XXXXXXXXXXXXXX	Watchpoint #14 enabled for WTM
XXXXXXXXXXX1XXXXXXXXXXXXXXX	Watchpoint #15 enabled for WTM
XXXXXXXXXX1XXXXXXXXXXXXXXXX	Watchpoint #16 enabled for WTM
XXXXXXXXX1XXXXXXXXXXXXXXXXX	Watchpoint #17 enabled for WTM
XXXXXXXX1XXXXXXXXXXXXXXXXXX	Watchpoint #18 enabled for WTM
XXXXXXX1XXXXXXXXXXXXXXXXXXX	Watchpoint #19 enabled for WTM
XXXXXX1XXXXXXXXXXXXXXXXXXXX	Watchpoint #20 enabled for WTM
XXXXX1XXXXXXXXXXXXXXXXXXXXX	Watchpoint #21 enabled for WTM
XXXX1XXXXXXXXXXXXXXXXXXXXXX	Watchpoint #22 enabled for WTM
XXX1XXXXXXXXXXXXXXXXXXXXXXX	Watchpoint #23 enabled for WTM
XX1XXXXXXXXXXXXXXXXXXXXXXXX	Watchpoint #24 enabled for WTM
X1XXXXXXXXXXXXXXXXXXXXXXXXX	Watchpoint #25 enabled for WTM
1XXXXXXXXXXXXXXXXXXXXXXXXXX	Watchpoint #26 enabled for WTM

14.14.1 Watchpoint Timing Diagram (2 MDO/1 MSEO configuration)

Figure 14-44 shows the watchpoint message.

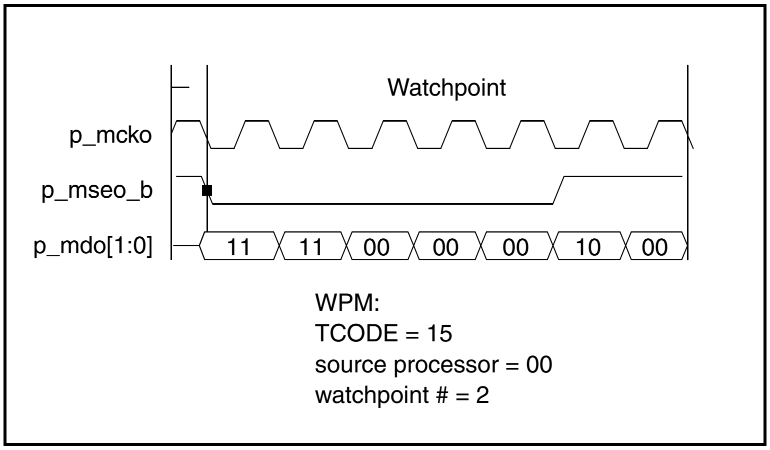


Figure 14-44. Watchpoint Message and Watchpoint Error Message

14.15 Nexus 3 Read/Write Access to Memory-Mapped Resources

The read/write access feature allows access to memory-mapped resources via the JTAG/OnCE port. The read/write mechanism supports single as well as block reads and writes to e200 AHB resources.

The Nexus 3 module is capable of accessing resources on the e200 system bus (AHB). Memory-mapped registers and other non-cached memory can be accessed via the standard memory map settings.

All accesses are setup and initiated by the read/write access control/status register (RWCS), as well as the read/write access address (RWA) and read/write access data registers (RWD).

Nexus 3 read/write accesses are run as privileged data non-cacheable, non-global accesses by default, and drive the **p_d_hprot[5:0]** bus access attributes to 0b000011 and the **p_d_gbl** access attribute to 0 accordingly. The RWCS[ATTR] field is provided to allow a portion of these default values to be modified when performing read or write accesses using the Nexus 3 Read/Write access mechanism.

Using the read/write access registers (RWCS/RWA/RWD), memory-mapped e200 AHB resources can be accessed through Nexus 3. The following subsections describe the steps that are required to access memory-mapped resources.

NOTE

Read/write access can only access memory mapped resources when system reset is de-asserted and clocks are running.

Misaligned accesses are not supported in the e200 Nexus3 module.

14.15.1 Single Write Access

The following sequence shows the steps required for single write access.

1. Initialize the read/write access address register (RWA) through the access method outlined in Section 14.5, “JTAG/OnCE Nexus 3 Register Access.” Configure as follows:
   a) Write Address → 0x*xxxx*_*xxxx* (write address)
2. Initialize the read/write access control/status register (RWCS) through the access method outlined in Section 14.5, “JTAG/OnCE Nexus 3 Register Access.” Configure the bits as follows:
   a) Access Control (AC) → 0b1 (to indicate start access)
   b) Map Select (MAP) → 0b000 (primary memory map)
   c) Access Priority (PR) → 0b00 (lowest priority)
   d) Read/Write (RW) → 0b1 (write access)
   e) Word Size (SZ) → 0b0*xx* (32-bit, 16-bit, 8-bit)
   f) Access Count (CNT) → 0x0000 or 0x0001(single access)

NOTE

Access count (CNT) of 0x0000 or 0x0001 performs a single access.

3. Initialize the read/write access data register (RWD) through the access method outlined in Section 14.5, “JTAG/OnCE Nexus 3 Register Access.” Configure as follows:
   Write Data → 0x*xxxx*_*xxxx* (write data)

4. The Nexus block arbitrates for the AHB system bus and transfer the data value from the data buffer RWD register to the memory mapped address in the read/write access address register (RWA). When the access has completed without error (ERR = 0b0), Nexus asserts the **nex_rdy_b** pin (see Table 14-39 for detail on **nex_rdy_b**) and clears the DV bit in the RWCS register. This indicates that the device is ready for the next access.

NOTE

Only the **nex_rdy_b** pin as well as the DV and ERR bits within the RWCS provide read/write access status to the external development tool.

14.15.2 Block Write Access

The following sequence shows the steps required for block write access.

1. For a block write access, follow Steps 1, 2, and 3 outlined in Section 14.15.1, “Single Write Access,” to initialize the registers, but use a value greater than one (0x0001) for RWCS[CNT].
2. The Nexus block arbitrates for the AHB system bus and transfers the first data value from the RWD Register to the memory mapped address in the read/write access address register (RWA). When the transfer has completed without error (ERR = 0b0), the address from the RWA register is incremented to the next word size (specified in the SZ field) and the number from the CNT field is decremented. Nexus then asserts the **nex_rdy_b** pin, indicating that the device is ready for the next access.
3. Repeat Step 3 in Section 14.15.1, “Single Write Access,” until the internal CNT value is zero (0). When this occurs, RWCS[DV] is cleared to indicate the end of the block write access.

NOTE

The actual RWA value as well as the CNT field within the RWCS are not changed when executing a block write access. The original values can be read by the external development tool at any time.

14.15.3 Single Read Access

The following sequence shows the steps required for single read access.

1. Initialize the read/write access address register (RWA) through the access method outlined in Section 14.5, “JTAG/OnCE Nexus 3 Register Access.” Configure as follows:
   a) Read Address → 0x*xxxx_xxxx* (read address)
2. Initialize the read/write access control/status register (RWCS) through the access method outlined in Section 14.5, “JTAG/OnCE Nexus 3 Register Access.” Configure the bits as follows:
   b) Access Control (AC) → 0b1 (to indicate start access)
   c) Map Select (MAP) → 0b000 (primary memory map)
   d) Access Priority (PR) → 0b00 (lowest priority)
   e) Read/Write (RW) → 0b0 (read access)
   f) Word Size (SZ) → 0b0xx (32-bit, 16-bit, 8-bit)
   g) Access Count (CNT) → 0x0000 or 0x0001 (single access)

NOTE

Access Count (CNT) of 14’h0000 or 14’h0001 performs a single access.

3. The Nexus block then arbitrates for the AHB system bus and the read data is transferred from the AHB to the RWD register. When the transfer is completed without error (ERR = 0b0), Nexus asserts the **nex_rdy_b** pin (see Table 14-39 for detail on **nex_rdy_b**) and sets RWCS[DV], indicating that the device is ready for the next access.
4. The data can be read from the read/write access data register (RWD) through the access method outlined in Section 14.5, “JTAG/OnCE Nexus 3 Register Access.”

NOTE

Only the **nex_rdy_b** pin as well as the DV and ERR bits within the RWCS provide read/write access status to the external development tool.

14.15.4 Block Read Access

The following sequence shows the steps required for block read access.

1. For a block read access, follow Steps 1 and 2 outlined in Section 14.15.3, “Single Read Access,” to initialize the registers, but use a value greater than one (0x0001) for RWCS[CNT].
2. The Nexus block then arbitrates for the AHB system bus and the read data transfers from the AHB to the RWD register. When the transfer has completed without error (ERR = 0b0), the address from the RWA register is incremented to the next word size (specified in the SZ field) and the number from the CNT field is decremented. Nexus then asserts the **nex_rdy_b** pin. This indicates that the device is ready for the next access.
3. The data can then be read from the read/write access data register (RWD) through the access method outlined in Section 14.5, “JTAG/OnCE Nexus 3 Register Access.”
4. Repeat Steps 3 and 4 in Section 14.15.3, “Single Read Access,” until the CNT value is zero (0). When this occurs, RWCS[DV] is set to indicate the end of the block read access.

NOTE

The data values must be shifted out 32-bits at a time LSB first (i.e. double word read = two word reads from the RWD).

NOTE

The actual RWA value as well as the CNT field within the RWCS are not changed when executing a block read access. The original values can be read by the external development tool at any time.

14.15.5 Error Handling

This section describes how the Nexus 3 module handles various error conditions.

14.15.5.1 AHB Read/Write Error

All address and data errors that occur on read/write accesses to the e200 AHB system bus return a transfer error encoding on the **p_hresp[1:0]** signals. If this occurs, perform the following steps:

1. The access is terminated without re-trying (AC bit is cleared)
2. The ERR bit in the RWCS register is set.
3. The error message is sent (TCODE = 8) indicating read/write error.

14.15.5.2 Access Termination

The following cases are defined for sequences of the read/write protocol that differ from those described in the above sections.

1. If RWCS[AC] is set to start read/write accesses and invalid values are loaded into the RWD and/or RWA, an AHB access error may occur. This is handled as described above.
2. If a block access is in progress (all cycles not completed), and the RWCS register is written, the original block access is terminated at the boundary of the nearest completed access.
   a) If the RWCS is written with the AC bit set, the next read/write access will begin and the RWD can be written to/read from.
   b) If the RWCS is written with the AC bit cleared, the read/write access is terminated at the nearest completed access. This method can be used to break (early terminate) block accesses.

14.15.6 Read/Write Access Error Message

The read/write access error message is sent out when an AHB system bus access error (read or write) has occurred.

Error information is messaged out as shown in Figure 14-45:

(5 bits)	(4 bits)	(6 bits)
Error Code (00011)	Src. Proc.	TCODE (001000)

Fixed length = 15 bits

Figure 14-45. Error Message Format

14.16 Nexus 3 Pin Interface

This section details information regarding the Nexus 3 pins and pin protocol.

The Nexus 3 pin interface provides the function of transmitting messages from the messages queues to the external tools. It is also responsible for handshaking with the message queues.

14.16.1 Pins Implemented

The Nexus 3 module implements an auxiliary port consisting of one (1) **nex_evti_b** and one (1) **nex_mseo_b** or two (2) **nex_mseo_b[1:0]**. It also implements a configurable number of **nex_mdo[n:0]** pins, (1) **nex_rdy_b** pin, (1) **nex_evto_b** pin, (3) **nex_wevto[2:0]** pins, and one (1) clock output pin (**nex_mcko**), as well as additional configuration pins described in Table 14-39. The output pins are synchronized to the Nexus 3 output clock (**nex_mcko**).

All Nexus 3 input functionality is controlled through the JTAG/OnCE port in compliance with IEEE 1149.1 (see Section 14.5, "JTAG/OnCE Nexus 3 Register Access," for details). The JTAG pins are incorporated as I/O to the e200 processor, and are further described in Section 13.4.3, "JTAG/OnCE Pins."

Table 14-38 shows the JTAG pins for Nexus 3.

Table 14-38. JTAG Pins for Nexus 3

JTAG Pins	Input/ Output	Description of JTAG Pins (included in e200 Nexus 1)
j_tdo	O	The Test Data Output (**j_tdo**) pin is the serial output for test instructions and data. **j_tdo** is three-stateable and is actively driven in the Shift-IR and Shift-DR controller states. **j_tdo** changes on the falling edge of **j_tclk**.
j_tdi	I	The Test Data Input (**j_tdi**) pin receives serial test instruction and data. TDI is sampled on the rising edge of **j_tclk**.
j_tms	I	The Test Mode Select (**j_tms**) input pin is used to sequence the OnCE controller state machine. **j_tms** is sampled on the rising edge of **j_tclk**.
j_tclk	I	The Test Clock (**j_tclk**) input pin is used to synchronize the test logic, and control register access through the JTAG/OnCE port.
j_trst_b	I	The Test Reset (**j_trst_b**) input pin is used to asynchronously initialize the JTAG/OnCE controller.

The auxiliary pins are used to send and receive messages and are described in Table 14-39.

Table 14-39. Nexus 3 Auxiliary Pins

Auxiliary Pins	Input/ Output	Description of Auxiliary Pins
nex_mcko	O	Message Clock Out (**nex_mcko**) is a free running output clock to development tools for timing of **nex_mdo[n:0]** and **nex_mseo_b[1:0]** pin functions. **nex_mcko** is programmable through the DC1 register.
nex_mdo[n:0]	O	Message Data Out (**nex_mdo[n:0])** are output pins used for OTM, BTM, and DTM. External latching of **nex_mdo[n:0]** shall occur on the rising edge of the Nexus3 clock (**nex_mcko**).
nex_mseo_b[1:0]	O	Message Start/End Out (**nex_mseo_b[1:0]**) are output pins which indicate when a message on the **nex_mdo[n:0]** pins has started, when a variable length packet has ended, and when the message has ended. External latching of **nex_mseo_b[1:0]** shall occur on the rising edge of the Nexus3 clock (**nex_mcko**). One or two pin MSEO functionality is determined at integration time per SoC implementation

Table 14-39. Nexus 3 Auxiliary Pins (continued)

Auxiliary Pins	Input/ Output	Description of Auxiliary Pins
nex_rdy_b	O	Ready (**nex_rdy_b**) is an output pin used to indicate to the external tool that the Nexus block is ready for the next read/write access. If Nexus is enabled, this signal is asserted upon successful (without error) completion of an AHB system bus transfer (Nexus read or write) and is held asserted until the JTAG/OnCE state machine reaches the Capture_DR state. Upon exit from system reset or if Nexus is disabled, **nex_rdy_b** remains de-asserted
nex_evto_b	O	Event Out (**nex_evto_b**)is an output which, when asserted, indicates one of two events has occurred based on the EOC bits in the DC1 register. **nex_evto_b** is held asserted for one (1) cycle of **nex_mcko**: 1) one (or more) watchpoints has occurred (from Nexus1) and EOC = 0b00 2) debug mode was entered (jd_debug_b asserted from Nexus1) and EOC = 0b01
nex_evti_b	I	Event In (**nex_evti_b**) is an input which, when asserted, initiates one of two events based on the EIC bits in the DC1 Register (if the Nexus module is enabled at reset): 1) Program trace and data trace synchronization messages (provided program trace and data trace are enabled and EIC = 0b00). 2) Debug request to e200 Nexus1 module (provided EIC = 0b01 and this feature is implemented).
nex_wevto[2:0]	O	Watchpoint Event Out 2–0 (**nex_wevto[2:0]**) are outputs which, when asserted, indicates one or more watchpoint events has occurred based on the settings in the DC2 and DC3 registers. **nex_wevto[2:0]** is held asserted for one (1) cycle of **nex_mcko**.
nex_ext_src_id[0:3]	I	nex_ext_src_id[0:3] is used to provide the SRC field value used in each message. These pins are tied to a predetermined value at SoC integration time

The Nexus auxiliary port arbitration pins are used when the Nexus 3 module is implemented in a multi-Nexus SoC which shares a single auxiliary output port. The arbitration is controlled by an SoC-level Nexus port control module (NPC). Refer to Section 14.18, "Auxiliary Port Arbitration," for detail on Nexus port arbitration.

Table 14-40 shows the Nexus port arbitration signals.

Table 14-40. Nexus Port Arbitration Signals

Nexus Port Arbitration Pins	Input/ Output	Description of Arbitration Pins
nex_aux_req[1:0]	O	Nexus Auxiliary Request (**nex_aux_req[1:0]**) output signals indicate a request for access to the shared Nexus auxiliary port to an SoC-level Nexus arbiter in a multi-Nexus implementation. The priority encodings are determined by how many messages are currently in the message queues (see Table 14-42).
nex_aux_busy	O	Nexus Auxiliary Busy (**nex_aux_busy**) is an output signal to an SoC-level Nexus arbiter, indicating that the Nexus 3 module is currently transmitting its message after being granted the Nexus auxiliary port.
npc_aux_grant	I	Nexus Auxiliary Grant (**npc_aux_grant**) is an input from the SoC-level Nexus port controller (NPC) that the auxiliary port has been granted to the Nexus 3 module to transmit its message.
ext_multi_nex_sel	I	Multi-Nexus Select (**ext_multi_nex_sel**) is a static signal indicating that the Nexus 3 module is implemented within a multi-Nexus environment. If set, port control and arbitration is controlled by the SoC-level arbitration module (NPC).

14.16.2 Pin Protocol

The protocol for the e200 processor transmitting messages via the auxiliary pins is accomplished with the MSEO pin function outlined in Table 14-41. Both single and dual pin cases are shown.

nex_mseo_b[1:0] is used to signal the end of variable-length packets, and not fixed length packets. **nex_mseo_b[1:0]** is sampled on the rising edge of the Nexus 3 clock (**nex_mcko**).

Table 14-41. MSEO Pin(s) Protocol

nex_mseo_b Function	Single nex_mseo_b data (serial)	Dual nex_mseo_b[1:0] data
Start of message	1-1-0	11-00
End of message	0-1-1-(more 1s)	00 (or 01)-11-(more 1’s)
End of variable length packet	0-1-0	00-01
Message transmission	0’s	00’s
Idle (no message)	1’s	11’s

Figure 14-46 illustrates the state diagram for single pin MSEO transfers.

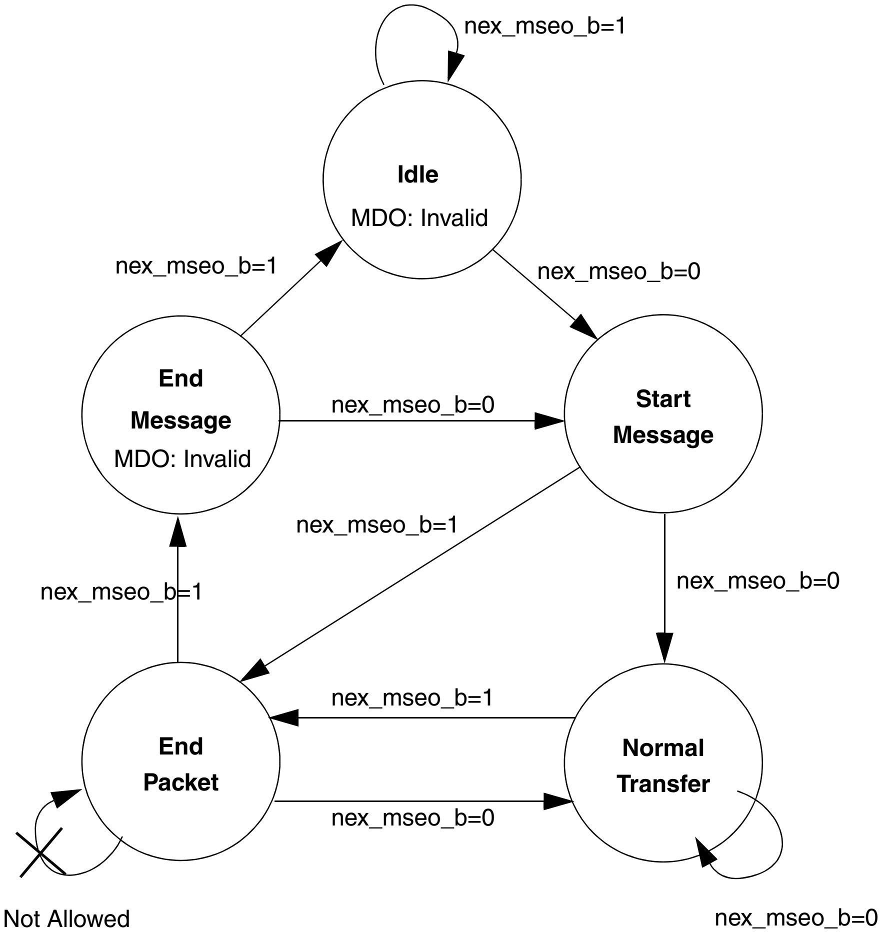


Figure 14-46. Single Pin MSEO Transfers

Note that the End Message state does not contain valid data on the **nex_mdo[n:0]** pins. Also, it is not possible to have two consecutive End Packet messages. This implies the minimum packet size for a variable length packet is 2× the number of **nex_mdo[n:0]** pins. This ensures that a false end of message state is not entered by emitting two consecutive 1s on the **nex_mseo_b** pin before the actual end of message.

Figure 14-47 illustrates the state diagram for dual pin MSEO transfers.

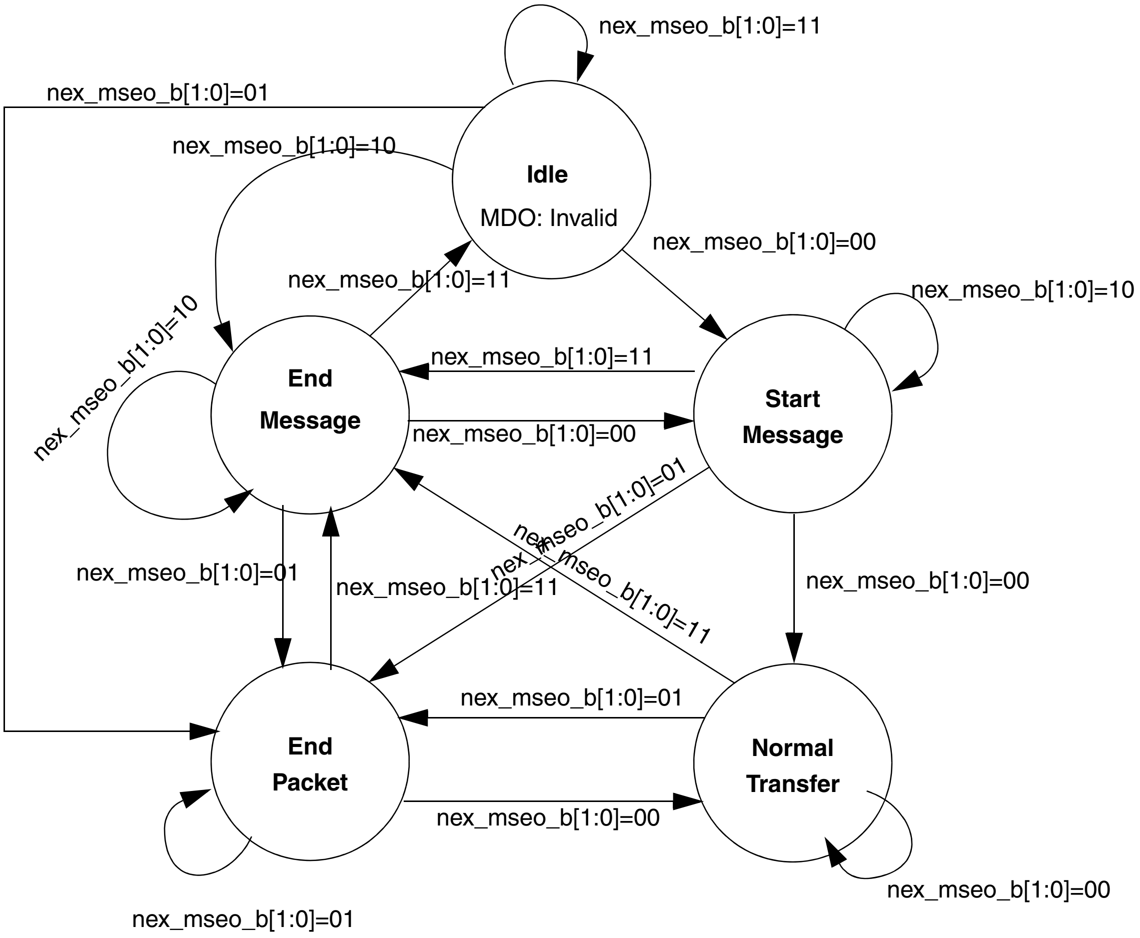


Figure 14-47. Dual Pin MSEO Transfers

The dual pin MSEO option is more robust that the single pin option. Termination of the current message may immediately be followed by the start of the next message on the consecutive clocks. An extra clock to end the message is not necessary as with the one MSEO pin option. The dual pin option also allows for consecutive End Packet states. This can be an advantage when small, variable sized packets are transferred.

NOTE

The “End Message” state may also indicate the end of a variable-length packet as well as the end of the message when using the dual pin option.

14.17 Rules for Output Messages

e200-based Class 3 compliant embedded processors must provide messages via the auxiliary port in a consistent manner as follows:

- A variable-sized packet within a message must end on a port boundary.

- A variable-sized packet may start within a port boundary only when following a fixed length packet. (If two variable-sized packets end and start on the same clock, it is impossible to know which bit is from the last packet and which bit is from the next packet.)
- Whenever a variable-length packet is sized such that it does not end on a port boundary, it is necessary to extend and zero fill the remaining bits after the highest-order bit so that it can end on a port boundary.

  For example, if the **nex_mdo[n:0]** port is 2 bits wide, and the unique portion of an indirect address TCODE is 5 bits, then the remaining 1 bit of **nex_mdo[n:0]** must be packed with a 0.

14.18 Auxiliary Port Arbitration

In a multi-Nexus environment, the Nexus 3 module must arbitrate for the shared Nexus port at the SoC level. The request scheme is implemented as a 2-bit request with various levels of priority. The priority levels are defined in Table 14-42 below. The Nexus 3 module receives a 1-bit grant signal (**npc_aux_grant**) from the SoC-level arbiter. When a grant is received, the Nexus 3 module begins transmitting its message following the protocol outlined in Section 14.16.2, "Pin Protocol." The Nexus 3 module maintains control of the port by asserting the **nex_aux_busy** signal until the $\overline{\text{MSEO}}$ state machine reaches the End Message state.

Table 14-42. MDO Request Encodings

Request Level	MDO Request Encoding (nex_aux_req[1:0])	Condition of Queue
No Request	00	No message to send
Low Priority	01	Message queue less than ½ full
—	10	Reserved
High Priority	11	Message queue ½ full or more

14.19 Examples

The following are examples of program trace and data trace messages.

Note that T0 and S0 are the least significant bits where:

- Tx = TCODE number (fixed)
- Sx = Source processor (fixed)
- MAP = Address Space Value (IS)
- Ix = Number of instructions (variable)
- Ax = Unique portion of the address (variable)

Note that during clock 13, the **nex_mdo[n:0]** pins are ignored in the single MSEO case.

Table 14-43 illustrates an example indirect branch message with 2 MDO/1MSEO configuration.

Table 14-43. Indirect Branch Message Example (2 MDO/1 MSEO)

Clock	nex_mdo[1:0]		nex_mseo_b	State
0	X	X	1	Idle (or end of last message)
1	T1	T0	0	Start message
2	T3	T2	0	Normal transfer
3	T5	T4	0	Normal transfer
4	S1	S0	0	Normal transfer
5	S3	S2	0	Normal transfer
6	I0	MAP	0	Normal transfer
7	I2	I1	0	Normal transfer
8	I4	I3	1	End packet
9	A1	A0	0	Normal transfer
10	A3	A2	0	Normal transfer
11	A5	A4	0	Normal transfer
12	A7	A6	1	End packet
13	0	0	1	End Message
14	T1	T0	0	Start Message

Table 14-44 illustrates the same example with an 8 MDO/2 MSEO configuration.

Table 14-44. Indirect Branch Message Example (8 MDO/2 MSEO)

Clock	nex_mdo[7:0]								nex_mseo_b[1:0]		State
0	X	X	X	X	X	X	X	X	1	1	Idle (or end of last message)
1	S1	S0	T5	T4	T3	T2	T1	T0	0	0	Start Message
2	I4	I3	I2	I1	I0	MAP	S3	S2	0	1	End Packet
3	A7	A6	A5	A4	A3	A2	A1	A0	1	1	End Packet/End Message
4	S1	S0	T5	T4	T3	T2	T1	T0	0	0	Start Message

Table 14-45 and Table 14-46 illustrate examples of direct branch messages: one with 2 MDO/1 MSEO, and one with 8 MDO/2 MSEO.

Note that T0 and I0 are the least significant bits where:

- Tx = TCODE number (fixed)
- Sx = Source processor (fixed)
- Ix = Number of Instructions (variable)

Table 14-45. Direct Branch Message Example (2 MDO/1 MSEO)

Clock	nex_mdo[1:0]		nex_mseo_b	State
0	X	X	1	Idle (or end of last message)
1	T1	T0	0	Start Message
2	T3	T2	0	Normal Transfer
3	T5	T4	0	Normal Transfer
4	S1	S0	0	Normal Transfer
5	S3	S2	0	Normal Transfer
6	I1	I0	1	End Packet
7	0	0	1	End Message

Table 14-46. Direct Branch Message Example (8 MDO/2 MSEO)

Clock	nex_mdo[7:0]								nex_mseo_b[1:0]		State
0	X	X	X	X	X	X	X	X	1	1	Idle (or end of last message)
1	S1	S0	T5	T4	T3	T2	T1	T0	0	0	Start Message
2	0	0	0	0	I1	I0	S3	S2	1	1	End Packet/End Message
3	S1	S0	T5	T4	T3	T2	T1	T0	0	0	Start Message

Table 14-47 illustrates an example data write message with 8 MDO/1 MSEO configuration.

Note that T0, A0, D0 are the least significant bits where:

- Tx = TCODE number (fixed)
- Sx = Source processor (fixed)
- MAP = Address Space Value (DS)
- Zx = Data size (fixed)
- Ax = Unique portion of the address (variable)
- Dx = Write data (variable = 8, 16 or 32-bit)

Table 14-47. Data Write Message Example (8 MDO/1 MSEO)

Clock	nex_mdo[7:0]								nex_mseo_b	State
0	X	X	X	X	X	X	X	X	1	Idle (or end of last message)

Table 14-47. Data Write Message Example (8 MDO/1 MSEO) (continued)

1	S1	S0	T5	T4	T3	T2	T1	T0	0	Start Message
2	A0	Z3	Z2	Z1	Z0	DS	S3	S2	1	End Packet
3	D7	D6	D5	D4	D3	D2	D1	D0	0	Normal Transfer
4	0	0	0	0	0	0	0	0	1	End Packet
5	0	0	0	0	0	0	0	0	1	End Message

Table 14-48 illustrates the same DWM with 8 MDO/2 MSEO configuration.

Table 14-48. Data Write Message Example (8 MDO/2 MSEO)

Clock	nex_mdo[7:0]								nex_mseo_b[1:0]		State
0	X	X	X	X	X	X	X	X	1	1	Idle (or end of last message)
1	S1	S0	T5	T4	T3	T2	T1	T0	0	0	Start Message
2	A0	Z3	Z2	Z1	Z0	DS	S3	S2	0	1	End Packet
3	D7	D6	D5	D4	D3	D2	D1	D0	1	1	End Packet/ End Message

14.20 Electrical Characteristics

For all electrical characteristics related to e200 and Nexus 3 operation, please refer to the appropriate “e200 Integration Guide.”

14.21 IEEE 1149.1 (JTAG) RD/WR Sequences

This section contains example JTAG/OnCE sequences used to access resources.

Table 14-49 shows the sequence for accessing the internal nexus registers.

Table 14-49. Accessing Internal Nexus 3 Registers via JTAG/OnCE

Step #	TMS Pin	Description
1	1	IDLE → SELECT-DR_SCAN
2	0	SELECT-DR_SCAN → CAPTURE-DR (Nexus Command Register value loaded in shifter)
3	0	CAPTURE-DR → SHIFT-DR
4	0	(7) TCK clocks issued to shift in direction (rd/wr) bit and first 6 bits of Nexus reg. addr.
5	1	SHIFT-DR → EXIT1-DR (7th bit of Nexus reg. shifted in)
6	1	EXIT1-DR → UPDATE-DR (Nexus shifter is transferred to Nexus Command Register)
7	1	UPDATE-DR → SELECT-DR_SCAN
8	0	SELECT-DR_SCAN → CAPTURE-DR (Register value is transferred to Nexus shifter)
9	0	CAPTURE-DR → SHIFT-DR
10	0	(31) TCK clocks issued to transfer register value to TDO pin while shifting in TDI value
11	1	SHIFT-DR → EXIT1-DR (MSB of value is shifted in/out of shifter)

Table 14-49. Accessing Internal Nexus 3 Registers via JTAG/OnCE (continued)

Step #	TMS Pin	Description
12	1	EXIT1-DR → UPDATE-DR (if access is write, shifter is transferred to register)
13	0	UPDATE-DR → RUN-TEST/IDLE (transfer complete - Nexus controller to Reg. Select state)

Table 14-50 shows the JTAG sequence for read access of memory-mapped resources.

Table 14-50. Accessing Memory-Mapped Resources (Reads)

Step #	TCLK clocks	Description
1	13	Nexus Command = write to Read/Write Access Address Register (RWA)
2	37	Write RWA (initialize starting read address - data input on TDI)
3	13	Nexus Command = write to Read/Write Control/Status Register (RWCS)
4	37	Write RWCS (initialize read access mode and CNT value - data input on TDI)
5	—	Wait for falling edge of **nex_rdy_b** pin
6	13	Nexus Command = read Read/Write Access Data Register (RWD)
7	37	Read RWD (data output on TDO)
8	—	If CNT > 0, go back to Step #5

Table 14-51 shows the JTAG sequence for write access of memory-mapped resources.

Table 14-51. Accessing Memory-Mapped Resources (Writes)

Step #	TCLK clocks	Description
1	13	Nexus Command = write to read/write access control/status register (RWCS)
2	37	Write RWCS (initialize write access mode and CNT value—data input on TDI)
3	13	Nexus Command = write to read/write address register (RWA)
4	37	Write RWA (initialize starting write address—data input on TDI)
5	13	Nexus Command = read read/write access data register (RWD)
6	37	Write RWD (data output on TDO)
7	—	Wait for falling edge of **nex_rdy_b** pin
8	—	If CNT > 0, go back to Step #5

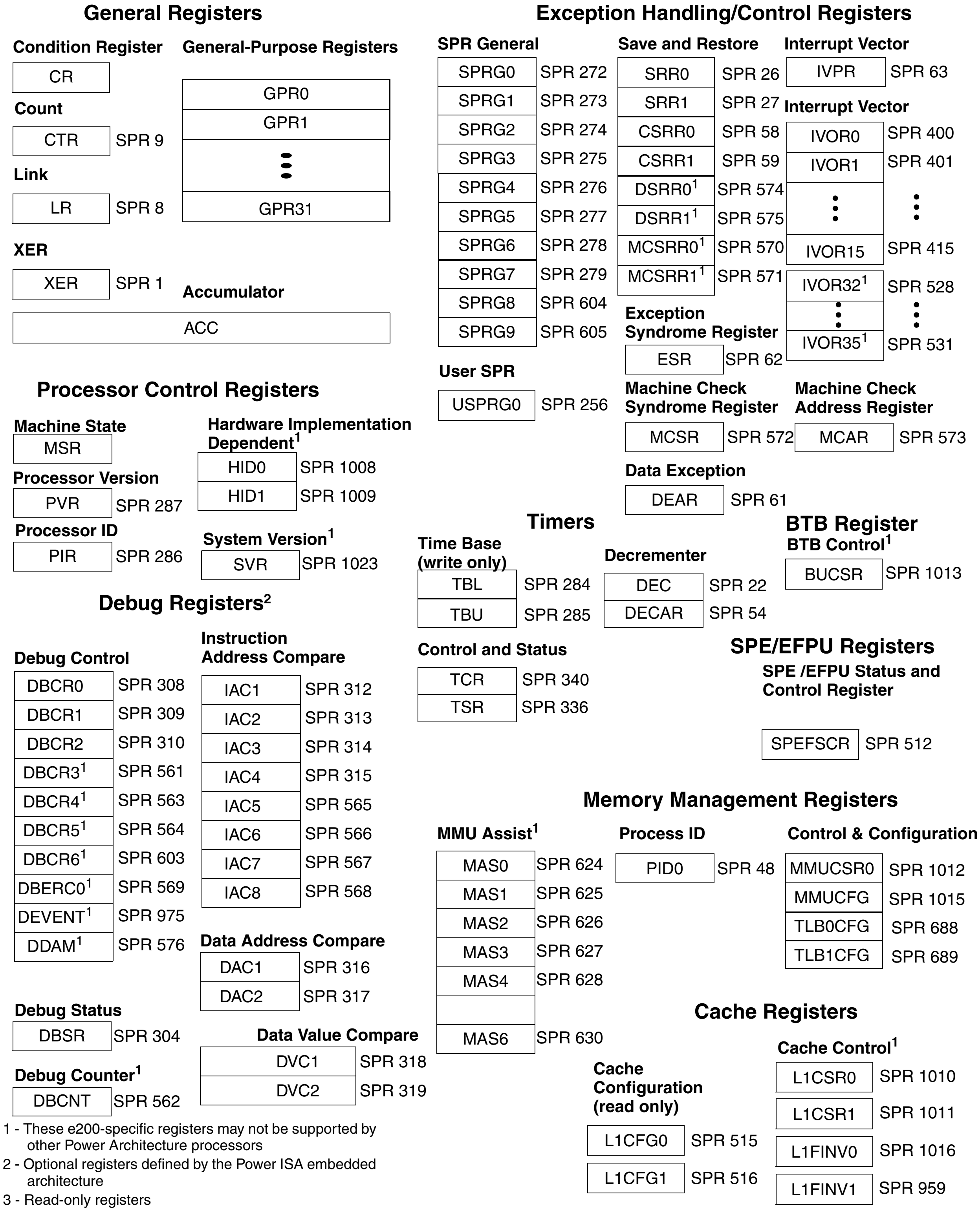
Appendix A
Register Summary

As shown in the following register diagrams, most of the registers implemented are defined by the Power ISA embedded architecture. Additional registers and fields within registers are defined by the EIS and by the implementation.

The Power ISA embedded architecture defines some register fields in a very general way, leaving some details as implementation specific. In some cases, this more specific functionality is defined by the EIS; in others it is left up to the processor. This chapter identifies the level at which each features is defined.

Figure A-1–Figure A-4 show the complete e200z7 register set divided into supervisor and user-level registers and grouped into general-purpose registers (GPRs), special-purpose registers (SPRs), device control registers (DCRs), and any performance monitor registers (PMRs) that may implemented in a particular variation of the e200z7 core family. The number to the right of the special-purpose registers (SPRs) is the decimal number used in the instruction syntax to access the register. For example, the integer exception register (XER) is SPR 1.

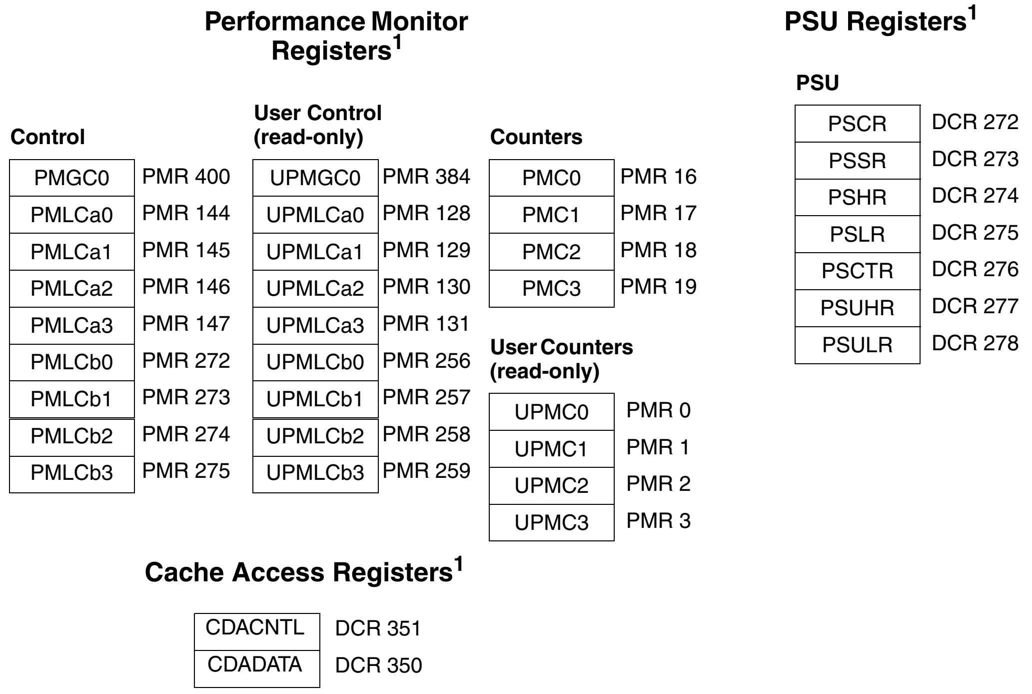
Figure A-1 shows the supervisor mode programmer’s model.



1 - These e200-specific registers may not be supported by other Power Architecture processors
2 - Optional registers defined by the Power ISA embedded architecture
3 - Read-only registers

Figure A-1. e200z760 Supervisor Mode Programmer’s Model

Figure A-2 shows the supervisor mode programmer models’s DCRs and PMRs.



Note:

1) These e200-specific registers may not be supported by other Power ISA embedded category processors

Figure A-2. e200z760 Supervisor Mode Programmer’s Model DCRs and PMRs

Figure A-3 shows the user mode programmer’s model.

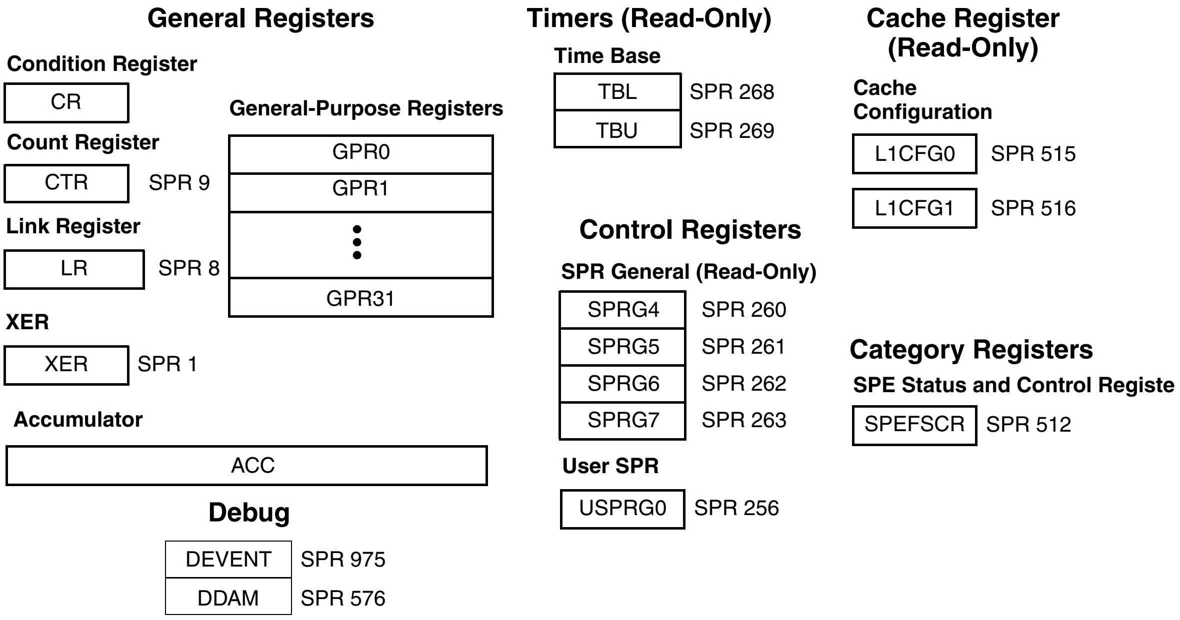


Figure A-3. e200z7 User Mode Programmer’s Model

Figure A-4 shows the user mode programmer’s model PMRs.

Performance Monitor Registers

User Control (read-only)	
UPMGC0	PMR 384
UPMLCa0	PMR 128
UPMLCa1	PMR 129
UPMLCa2	PMR 130
UPMLCa3	PMR 131
UPMLCb0	PMR 256
UPMLCb1	PMR 257
UPMLCb2	PMR 258
UPMLCb3	PMR 259

User Counters (read-only)	
UPMC0	PMR 0
UPMC1	PMR 1
UPMC2	PMR 2
UPMC3	PMR 3

Note:

These e200-specific registers may not be supported by other Power ISA embedded category processors.

Figure A-4. e200 User Mode Programmer’s Model PMRs

Access: Read/Write

	0 4	5	6	7 12	13	14	15	16	17	18	19	20	21	22	23	24 25	26	27	28	29	30	31
R W	—	UCLE	SPE	—	WE	CE	—	EE	PR	FP	ME	FE0	—	DE	FE1	—	IS	DS	—	PMM	RI	—
Reset	All zeros																					

Figure A-5. Machine State Register (MSR)

SPR 286 Access: Read/Write

	0 … 31
R W	ID

Reset 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 *n*[1] *n*[1] *n*[1] *n*[1] *n*[1] *n*[1] *n*[1] *n*[1]

[1] Updated to reflect the values on *p_cpuid*[0:7]

Figure A-6. Processor ID Register (PIR)

SPR 287 Access: Read only

	0 3	4 5	6 11	12 15	16 19	20 23	24 27	28 31
R	MANID	—	Type	Version	MBG Use	Minor Rev	Major Rev	MBG ID
W								

Figure A-7. Processor Version Register (PVR)

SPR 1023 Access: Read only

	0–31
R	System Version
W	

Figure A-8. System Version Register (SVR)

SPR 1 Access: Read/Write

	0	1	2	3–24	25–31
R W	SO	OV	CA	—	Bytecnt
Reset	All zeros				

Figure A-9. Integer Exception Register (XER)

SPR 62 Access: Read/Write

	0–3	4	5	6	7	8	9	10	11	12	13	14	15	16–23	24	25	26	27–29	30	31
R W	—	PIL	PPR	PTR	FP	ST	—	DLK	ILK	AP	PUO	BO	PIE	—	SPE	—	VLEMI	—	MIF	—
Reset	All zeros																			

Figure A-10. Exception Syndrome Register (ESR)

SPR 572 Access: w1c

	0	1	2	3	4	5	6	7	8	9–10	11	12	13	14	15
R	MCP	IC_DPERR	CP_PERR	DC_DPERR	EXCP_ERR	IC_TPERR	DC_TPERR	IC_LKERR	DC_LKERR	—	NMI	MAV	MEA	—	IF
W	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c	w1c		w1c	w1c	w1c		w1c
Reset	All zeros														

	16	17	18	19–25	26	27	28	29	30–31
R	LD	ST	G	—	SNPERR	BUS_IRERR	BUS_DRERR	BUS_WRERR	—
W	w1c	w1c	w1c		w1c	w1c	w1c	w1c	
Reset	All zeros								

Figure A-11. Machine Check Syndrome Register (MCSR)

SPR 340 Access: Read/Write

	0–1	2–3	4	5	6–7	8	9	10	11–14	15–18	19–31
R W	WP	WRC	WIE	DIE	FP	FIE	ARE	—	WPEXT	FPEXT	—
Reset	All zeros										

Figure A-12. Timer Control Register (TCR)

SPR 336 Access: w1c

	0	1	2 3	4	5	6–31
R	ENW	WIS	WRS	DIS	FIS	—
W	w1c	w1c	w1c	w1c	w1c	
Reset	All zeros					

Figure A-13. Timer Status Register (TSR)

SPR 1008 Access: Read/Write

	0	1–7	8	9	10	11–13	14	15
R W	EMCP	—	DOZE	NAP	SLEEP	—	ICR	NHR
Reset	All zeros							

	16	17	18	19	20	21	22	23	24–30	31
R W	—	TBEN	SEL_TBCLK	DCLREE	DCLRCE	CICLRDE	MCCLRDE	DAPUEN	—	NOPTI
Reset	All zeros									

Figure A-14. Hardware Implementation Dependent Register 0 (HID0)

SPR 1009 Access: Read/Write

	0–15	16–23	24	25–31
R W	—	SYSCTL	ATS	—
Reset	All zeros			

Figure A-15. Hardware Implementation Dependent Register 1 (HID1)

SPR 1013 Access: Read/Write

	0–21	22	23–25	26 27	28	29 30	31
R W	—	BBFI	—	BALLOC	—	BPRED	BPEN
Reset	All zeros						

Figure A-16. Branch Unit Control and Status Register (BUCSR)

SPR 512 Access: Read/Write

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R W	SOVH	OVH	FGH	FXH	FINVH	FDBZH	FUNFH	FOVFH	—	RM	FINXS	FINVS	FDBZS	FUNFS	FOVFS	MODE
Reset	All zeros															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30 31
R W	SOV	OV	FG	FX	FINV	FDBZ	FUNF	FOVF	—	FINXE	FINVE	FDBZE	FUNFE	FOVFE	FRMC
Reset	All zeros														

Figure A-17. SPE/EFPU Status and Control Register (SPEFSCR)

SPR 400–415
528–530

Access: Read/Write

	0–15	16–27	28–31
R W	—	Vector Offset	—

Figure A-18. e200 Interrupt Vector Offset Register (IVOR)

PMR 400

Access: Read/Write

	0	1	2	3–18	19–20	21–22	23	24–31
R W	FAC	PMIE	FCECE	—	TBSEL	—	TBEE	—
Reset	All zeros							

Figure A-19. Performance Monitor Global Control Register (PMGC0)

PMR 144–147

Access: Read/Write

	0	1	2	3	4	5	6–7	8–15	16	17–19	20–31
R W	FC	FCS	FCU	FCM1	FCM0	CE	—	EVENT	—	PMP	—
Reset	All zeros										

Figure A-20. Performance Monitor Local Control A Registers (PMLCa0–PMLCa3)

PMR 272–275

Access: Read/Write

	0	1–3	4	5–7	8	9–12	13	14–15
R W	—	TRIGONCTL	—	TRIGOFFCTL	—	TRIGONSEL	—	TRIGOFFSEL
Reset	All zeros							

	16–17	18	19–20	21–23	24–25	26–31
R W	TRIGOFFSEL	TRIGGERED	—	THRESHMUL	—	THRESHOLD
Reset	All zeros					

Figure A-21. Performance Monitor Local Control B Registers (PMLCb0–PMLCb3)

PMR 16–19

Access: Read/Write

	0	1–31
R W	OV	Counter Value
Reset	All zeros	

Figure A-22. Performance Monitor Counter Registers (PMC0–PMC3)

SPR 1010 Access: Read/Write

	0–3	4–7	8–10	11	12–13	14	15
R W	WID	WDD	—	DCWM	DCWA	—	DCECE
Reset	All zeros						

	16	17	18–19	20	21	22	23	24	25–26	27	28	29	30	31
R W	DCEI	—	DCEDT	DCSLC	DCUL	DCLO	DCLFC	DCLOA	DCEA	—	DCBZ32	DCABT	DCINV	DCE
Reset	All zeros													

Figure A-23. L1 Cache Control and Status Register 0 (L1CSR0)

SPR 1011 Access: Read/Write

	0–14	15	16	17	18–19	20	21	22	23	24	25–26	27–28	29	30	31
R W	—	ICECE	ICEI	—	ICEDT	—	ICUL	ICLO	ICLFC	ICLOA	ICEA	—	ICABT	ICINV	ICE
Reset	All zeros														

Figure A-24. L1 Cache Control and Status Register 1 (L1CSR1)

SPR 515 Access: Read only

	0–1	2	3	4	5–6	7–8	9–10	11	12	13–20	21–31
R	CARCH	CWPA	CFAHA	DCFISWA	—	DCBSIZE	DCREPL	DCLA	DCECA	DCNWAY	DCSIZE
W											
Reset	0 0	1	0	1	0 0	0 0	1 0	1	1	0 0 0 0 0 0 1 1	0 0 0 0 0 0 1 0 0 0 0

Figure A-25. L1 Cache Configuration Register 0 (L1CFG0)

SPR 516 Access: Read only

	0–3	4	5–6	7–8	9–10	11	12	13–20	21–31
R	—	ICFISWA	—	ICBSIZE	ICREPL	ICLA	ICECA	ICNWAY	ICSIZE
W									
Reset	0 0 0 0	1	0 0	0 0	1 0	1	1	0 0 0 0 0 0 1 1	0 0 0 0 0 0 1 0 0 0 0

Figure A-26. L1 Cache Configuration Register 1 (L1CFG1)

SPR 1016 Access: Read/Write

	0–5	6–7	8–19	20–26	27–29	30–31
R W	—	CWAY	—	CSET	—	CCMD
Reset	All zeros					

Figure A-27. L1 Flush/Invalidate Register 0 (L1FINV0)

SPR 959 Access: Read/Write

	0–5	6–7	8–19	20–26	27–29	30–31
R	—	CWAY	—	CSET	—	CCMD
W						
Reset	All zeros					

Figure A-28. L1 Flush/Invalidate Register 1 (L1FINV1)

SPR 1015 Access: Read only

	0–7	8–14	15–16	17–20	21–25	26–27	28–29	30–31
R	—	RASIZE	—	NPIDS	PIDSIZE	—	NTLBS	MAVN
W								
Reset								

Figure A-29. MMU Configuration Register (MMUCFG)

SPR 688 Access: Read only

	0–7	8–11	12–15	16	17	18	19	20–31
R	ASSOC	MINSIZE	MAXSIZE	IPROT	AVAIL	P2PSA	—	NENTRY
W								
Reset	All zeros							

Figure A-30. TLB0 Configuration Register (TLB0CFG)

SPR 689 Access: Read only

	0–7	8–11	12–15	16	17	18	19	20–31
R	ASSOC	MINSIZE	MAXSIZE	IPROT	AVAIL	P2PSA	—	NENTRY
W								
Reset								

Figure A-31. TLB1 Configuration Register (TLB1CFG)

SPR 61 Access: Read/Write

	0–31
R	Effective Page Address
W	
Reset	Unaffected

Figure A-32. Data Exception Address Register

SPR 1012 Access: Read/Write

	0–29	30	31
R	—	TLB1_FI	—
W			
Reset	All zeros		

Figure A-33. MMU Control and Status Register 0 (MMUCSR0)

SPR 624 Access: Read/Write

	0 1	2 3	4 9	10 15	16 25	26 31
R W	—	TLBSEL (01)	—	ESEL	—	NV
Reset	Unaffected					

Figure A-34. MMU Assist Register 0 (MAS0)

SPR 625 Access: Read/Write

	0	1	2 7	8 15	16 18	19	20 24	25 31
R W	VALID	IPROT	—	TID	—	TS	TSIZE	—
Reset	Unaffected							

Figure A-35. MMU Assist Register 1 (MAS1)

SPR 626 Access: Read/Write

	0 19 20 21	22 25	26	27	28	29	30	31
R W	EPN	—	VLE	W	I	M	G	E
Reset	Unaffected							

Figure A-36. MMU Assist Register 2 (MAS2)

SPR 627 Access: Read/Write

	0 21	22	23	24	25	26	27	28	29	30	31
R W	RPN	U0	U1	U2	U3	UX	SX	UW	SW	UR	SR
Reset	Unaffected										

Figure A-37. MMU Assist Register 3 (MAS3)

SPR 628 Access: Read/Write

	0 1	2 3	4 13	14 15	16 19	20 23	24 25	26	27	28	29	30	31
R W	—	TLBSELD (01)	—	TIDSELD	—	TSIZED	—	VLED	WD	ID	MD	GD	ED
Reset	Unaffected												

Figure A-38. MMU Assist Register 4 (MAS4)

SPR 630 Access: Read/Write

	0 7	8 15	16 30	31
R W	—	SPID	—	SAS
Reset	Unaffected			

Figure A-39. MMU Assist Register 6 (MAS6)

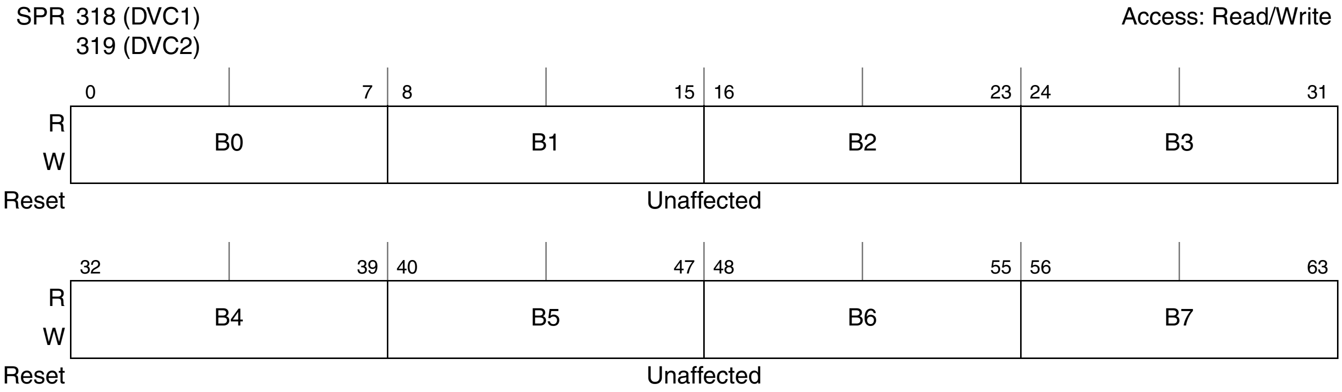


Figure A-40. DVC1, DVC2 Registers

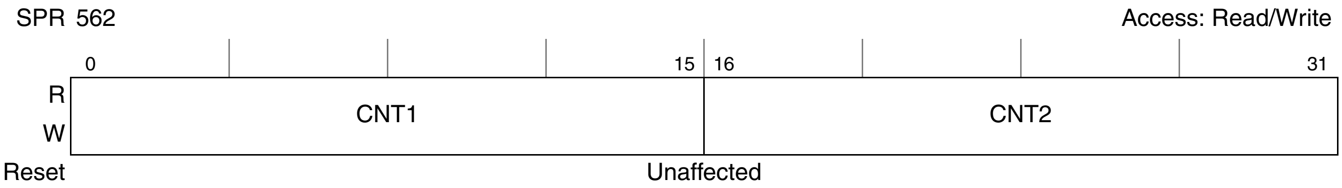


Figure A-41. DBCNT Register

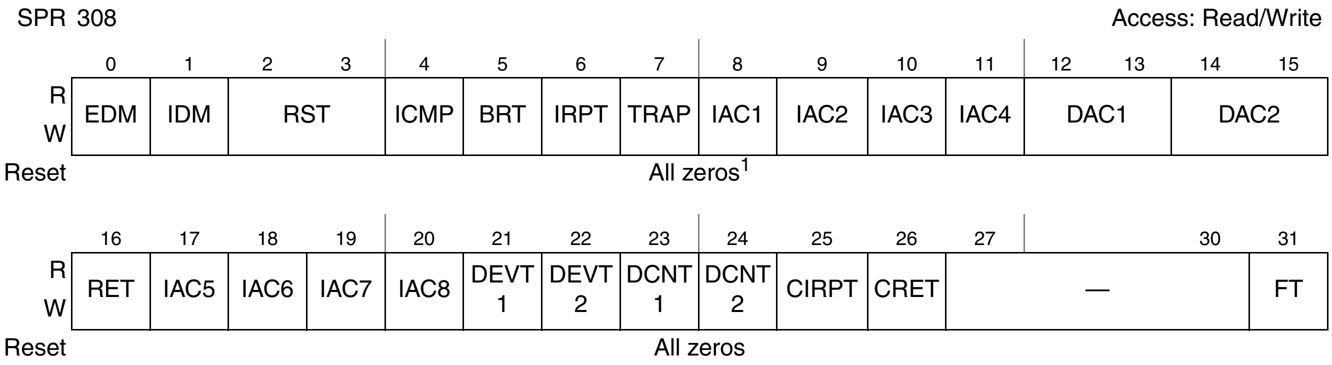


Figure A-42. DBCR0 Register

[1] DBCR0[EDM] is affected by **j_trst_b** or **m_por** assertion and remains reset while in the Test_Logic_Reset state, but it is not affected by **p_reset_b**. All other bits are reset by processor reset **p_reset_b** if DBCR0[EDM] = 0, as well as unconditionally by **m_por**. If DBCR0[EDM]=1, DBERC0 masks off hardware-owned resources (other than RST) from reset by **p_reset_b**, and only software-owned resources indicated by DBERC0 and the DBCR0[RST] field will be reset by **p_reset_b**. DBCR0[RST] is always reset by **p_reset_b** regardless of the value of DBCR0[EDM].

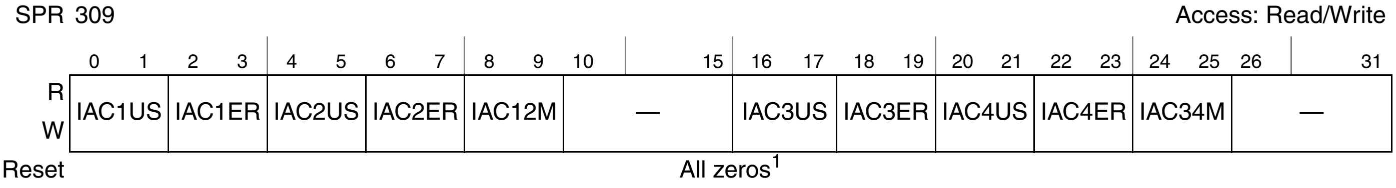


Figure A-43. DBCR1 Register

[1] Reset by processor reset **p_reset_b** if DBCR0[EDM] = 0, as well as unconditionally by **m_por**. If DBCR0[EDM] = 1, DBERC0 masks off hardware-owned resources from reset by **p_reset_b,** and only software-owned resources indicated by DBERC0 are reset by **p_reset_b.**

SPR 310 Access: Read/Write

	0 1	2 3	4 5	6 7	8 9	10	11	12 13	14 15	16 … 23	24 … 31
R W	DAC1 US	DAC1 ER	DAC2 US	DAC2 ER	DAC1 2M	DAC1 LNK	DAC2 LNK	DVC1 M	DVC2 M	DVC1BE	DVC2BE
Reset	All zeros[1]										

Figure A-44. DBCR2 Register

[1] Reset by processor reset **p_reset_b** if DBCR0[EDM] = 0, as well as unconditionally by **m_por**. If DBCR0[EDM] = 1, DBERC0 masks off hardware-owned resources from reset by **p_reset_b,** and only software-owned resources indicated by DBERC0 are reset by **p_reset_b**.

SPR 561 Access: Read/Write

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R W	DEVT 1C1	DEVT 2C1	ICMP C1	IAC1 C1	IAC2 C1	IAC3 C1	IAC4 C1	DAC1 RC1	DAC1 WC1	DAC2 RC1	DAC2 WC1	IRPT C1	RETC 1	DEVT 1C2	DEVT 2C2	ICMP C2
Reset	All zeros[1]															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R W	IAC1 C2	IAC2 C2	IAC3 C2	IAC4 C2	DAC1 RC2	DAC1 WC2	DAC2 RC2	DAC2 WC2	DEVT 1T1	DEVT 2T1	IAC1T 1	IAC3T 1	DAC1 RT1	DAC1 WT1	CNT2 T1	CON FIG
Reset	All zeros[1]															

Figure A-45. DBCR3 Register

[1] Reset by processor reset **p_reset_b** if DBCR0[EDM] = 0, as well as unconditionally by m_por. If DBCR0[EDM] = 1, DBERC0 masks off hardware-owned resources from reset by **p_reset_b** and only software-owned resources indicated by DBERC0 will be reset by **p_reset_b.**

SPR 563 Access: Read/Write

	0	1	2	3	4 … 15	16 … 19	20 … 23	24 … 31
R W	—	DVC1C	—	DVC2C	—	DAC1XM	DAC2XM	—
Reset	All zeros[1]							

Figure A-46. DBCR4 Register

[1] DBCR4 is reset by processor reset **p_reset_b** if DBCR0[EDM] = 0, as well as unconditionally by **m_por**. If DBCR0[EDM] = 1, DBERC0 masks off hardware-owned resources from reset by **p_reset_b.** Only software-owned resources indicated by DBERC0 are reset by **p_reset_b**.

SPR 564 Access: Read/Write

	0 1	2 3	4 5	6 7	8 9	10 … 15	16 17	18 19	20 21	22 23	24 25	26 … 31
R W	IAC5US	IAC5ER	IAC6US	IAC6ER	IAC56M	—	IAC7US	IAC7ER	IAC8US	IAC8ER	IAC78M	—
Reset	All zeros[1]											

Figure A-47. DBCR5 Register

[1] Reset by processor reset **p_reset_b** if DBCR0[EDM] = 0, as well as unconditionally by **m_por**. If DBCR0[EDM] = 1, DBERC0 masks off hardware-owned resources from reset by **p_reset_b** and only software-owned resources indicated by DBERC0 will be reset by **p_reset_b**.

SPR 603 Access: Read/Write

	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
R W	IAC1XM	IAC2XM	IAC3XM	IAC4XM	IAC5XM	IAC6XM	IAC7XM	IAC8XM
Reset	All zeros[1]							

Figure A-48. DBCR6 Register

[1] DBCR6 is reset by processor reset **p_reset_b** if DBCR0[EDM] = 0, as well as unconditionally by **m_por**. If DBCR0[EDM] = 1, DBERC0 masks off hardware-owned resources from reset by **p_reset_b** and only software-owned resources indicated by DBERC0 will be reset by **p_reset_b**.

SPR 304 Access: Read/Write

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R W	IDE	UDE	MRR		ICMP	BRT	IRPT	TRAP	IAC1	IAC2	IAC3	IAC4-8	DAC1 R	DAC1 W	DAC2 R	DAC2 W
Reset[1]	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0

	16	17			20	21	22	23	24	25	26	27	28	29	30	31
R W	RET	—				DEVT 1	DEVT 2	DCNT 1	DCNT 2	CIRP T	CRET	VLES	DAC_OFST			CNT1 TRG
Reset[1]	All zeros															

Figure A-49. DBSR Register

[1] Reset by processor reset **p_reset_b** if DBCR0[EDM] = 0, as well as unconditionally by **m_por**. However, DBSR[MRR] is always updated by **p_reset_b**. If DBCR0[EDM] = 1, DBERC0 masks off hardware-owned resources from reset by **p_reset_b**, and **p_reset_b** only resets the software-owned resources indicated by DBERC0**.** However, **p_reset_b** always updates DBSR[MRR].

SPR 569 Access: Read only

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	—	IDM	RST	UDE	ICMP	BRT	IRPT	TRAP	IAC1	IAC2	IAC3	IAC4	DAC1	—	DAC2	—
W																
Reset	Unaffected[1]															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	RET	IAC5	IAC6	IAC7	IAC8	DEVT 1	DEVT 2	DCNT 1	DCNT 2	CIRPT	CRET	BKPT	DQM	—		FT
W																
Reset	Unaffected[1]															

[1] Unaffected by **p_reset_b**; cleared by **m_por** or while in the test-logic-reset OnCE controller state

Figure A-50. DBERC0 Register

SPR 975

Access: Special

	0–7	8–23	24–31
R W	DQTAG	—	DEVNT
Reset	All zeros[1]		

[1] Reset by processor reset **p_reset_b** if DBCR0[EDM] = 0, as well as unconditionally by **m_por**. If DBCR0[EDM] = 1, DBERC0 masks off hardware-owned resources from reset by **p_reset_b**, and **p_reset_b** only resets software-owned resources indicated by DBERC0**.** Note that DEVNT field is shared by hardware and software but is always reset by **p_reset_b**.

Figure A-51. DEVENT Register

SPR 576

Access: Special

	0–31
R W	DDAM
Reset	All zeros[1]

[1] Reset by processor reset **p_reset_b** if DBCR0[EDM] = 0, as well as unconditionally by **m_por**. If DBCR0[EDM] = 1, DBERC0 masks off hardware-owned resources from reset by **p_reset_b,** and **p_reset_b** only resets software-owned resources indicated by DBERC0.

Figure A-52. DDAM Register

Access: Special

	0	1	2–3	4–31
R	EDM	DNH_EN	DFT	—
W				
Reset	All zeros[1]			

[1] EDBCR0 is affected (reset) by **j_trst_b** or **m_por** assertion and remains reset while in the Test_Logic_Reset state. It is not affected by **p_reset_b**.

Figure A-53. EDBCR0 Register

Access: Read/Write

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R W	IDE	UDE	DNH	0	ICMP	BRT	IRPT	TRAP	IAC1	IAC2	IAC3	IAC4–8	DAC1R	DAC1W	DAC2R	DAC2W
Reset	All zeros															

	16	17–20	21	22	23	24	25	26	27	28–30	31
R W	RET	—	DEVT1	DEVT2	DCNT1	DCNT2	CIRPT	CRET	VLES	DAC_OFST	CNT1TRG
Reset	All zeros										

[1] Reset by **j_trst_b** or **m_por** assertion and remains reset while in the Test_Logic_Reset state or while EDBCR0[EDM] = 0.

Figure A-54. EDBSR0 Register

Access: Read/Write

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R W	—	UDE	DNH	—	ICMP	BRT	IRPT	TRAP	IAC1	IAC2	IAC3	IAC4–8	DAC1R	DAC1W	DAC2R	DAC2W
Reset	All zeros[1]															

	16	17–20	21	22	23	24	25	26	27–31
R W	RET	—	DEVT1	DEVT2	DCNT1	DCNT2	CIRPT	CRET	—
Reset	All zeros[1]								

[1] Reset by **j_trst_b** or **m_por** assertion and remains reset while in the Test_Logic_Reset state or while EDBCR0[EDM] = 0.

Figure A-55. EDBSRMSK0 Register

	0	1	2	3	4	5	6	7	8	9
R W	MCLK	ERR	0	RESET	HALT	STOP	DEBUG	WAIT	0	1

Figure A-56. e200 OnCE Status Register

Access: Read/Write

	0	1	2	3	4	5	6	7	8	9
R W	R/W	GO	EX	RS[0–6]						
Reset[1]	1	0	0	0	0	0	0	0	1	0

[1] On assertion of **j_trst_b** or **m_por**, or while in the Test_Logic_Reset state

Figure A-57. OnCE Command Register

Access: Read/Write

	0–7	8	9–10	11	12	13	14	15
R W	—	I_DMDIS	—	I_DVLE	I_DI	I_DM	—	I_DE
Reset	All zeros[1]							

	16	17–18	19	20	21	22	23	24–28	29	30	31
R W	D_DMDIS	—	D_DW	D_DI	D_DM	D_DG	D_DE	—	WKUP	FDB	DR
Reset	All zeros[1]										

[1] All zeros on **m_por**, **j_trst_b**, or entering Test_logic_Reset state

Figure A-58. OnCE Control Register

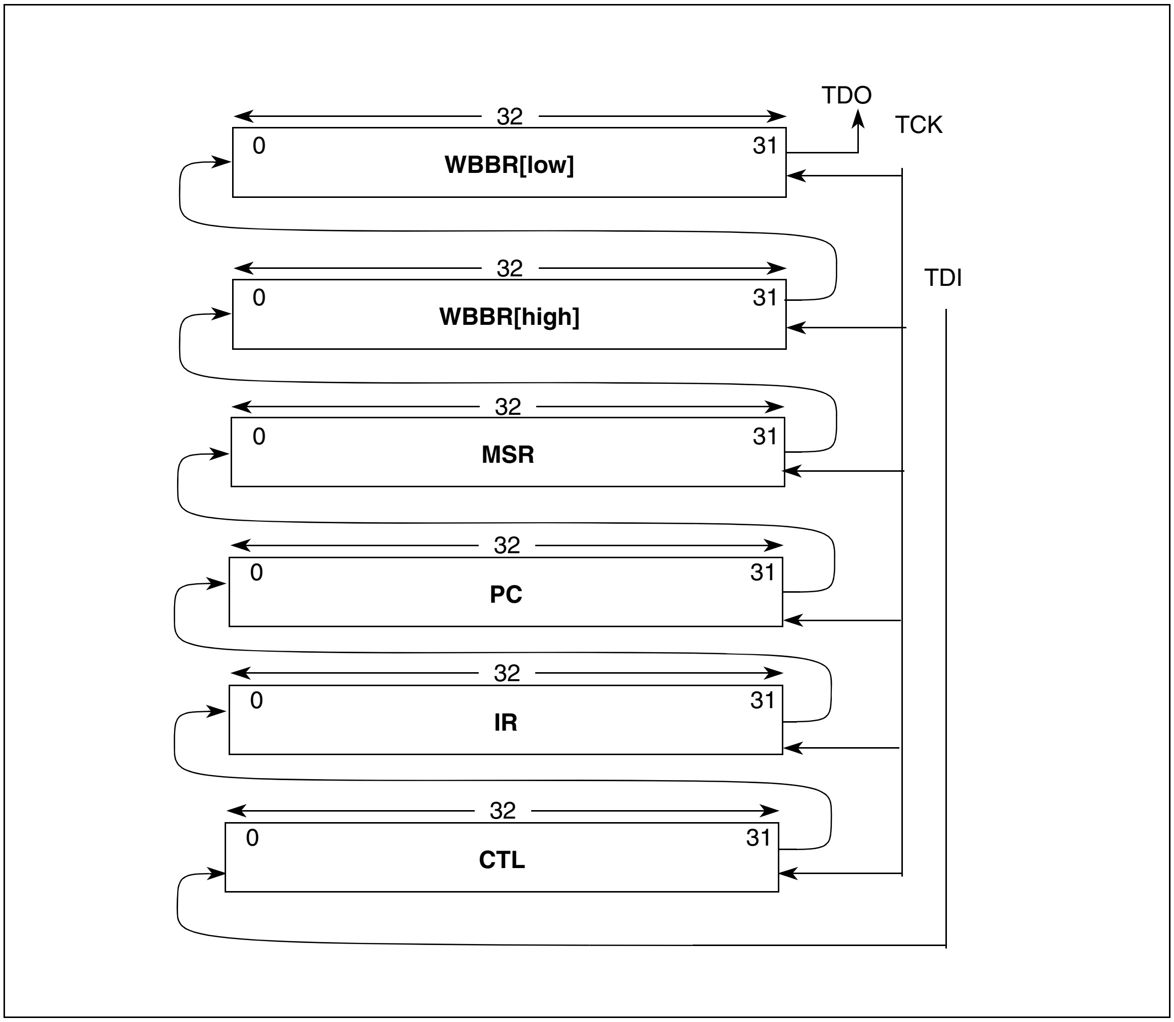


Figure A-59. CPU Scan Chain Register (CPUSCR)

Access: Read/Write

	0		10	11	12	13	14	15
R W	*			IRSTAT 13	IRSTAT 12	IRSTAT 11	IRSTAT 10	WAITING

	16 19	20	21	22	23	24	25	26	27	28	29	30	31
R W	PCOFST	PCINV	FFRA	IRSTAT 0	IRSTAT 1	IRSTAT 2	IRSTAT 3	IRSTAT 4	IRSTAT 5	IRSTAT 6	IRSTAT 7	IRSTAT 8	IRSTAT 9

Figure A-60. Control State Register (CTL)

DCR 272 Access: Read/Write

	0–25	26	27–28	29	30	31
R W	—	CNTEN	—	RDEN	WREN	INIT
Reset	All zeros					

Figure A-61. Parallel Signature Control Register (PSCR)

DCR 273 Access: Read/Write

	0–30	31
R W	—	TERR
Reset	Unaffected	

Figure A-62. Parallel Signature Status Register (PSSR)

DCR 274 Access: Read/Write

	0–31
R W	High Signature
Reset	Unaffected

Figure A-63. Parallel Signature High Register (PSHR)

DCR 275 Access: Read/Write

	0–31
R W	Low Signature
Reset	Unaffected

Figure A-64. Parallel Signature Low Register (PSLR)

DCR 276 Access: Read/Write

	0–31
R W	Counter
Reset	Unaffected

Figure A-65. Parallel Signature Counter Register (PSCTR)

DCR 277 Access: Write only

	0–31
R	
W	High Signature Update Data
Reset	Unaffected

Figure A-66. Parallel Signature Update High Register (PSUHR)

DCR 278 Access: Write only

	0 — 31
R	
W	Low Signature Update Data
Reset	Unaffected

Figure A-67. Parallel Signature Update Low Register (PSULR)

Appendix B
Revision History

This appendix provides a list of the major differences between revisions of the *e200z760n3 Power Architecture® Core Reference Manual.*

B.1 Changes between revisions 1 and 2

Table B-1. Changes between revisions 1and 2

Chapter	Description
Chapter 3, "Instruction Model"	In Section 3.14, "Volatile Context Save/Restore Unit" • For instruction "e_lmvsprw" added NOTE: "If the EA is misaligned and the e_lmvsprw is followed by either a branch to link register or branch to count register within 4 instructions, the core can lock up during exception handling for the misalignment. To avoid this issue, do not do misaligned on e_lmvsprw or ensure there are at least 4 instructions in between the e_lmvsprw and the branch to LR or CTR. This issue does not apply to Book E applications."
Chapter 4, "Instruction Pipeline and Execution Timing"	Section 4.3.2, "Instruction Prefetch Buffers and Branch Target Buffer" • Added NOTE: "- The e200z7 core can prefetch up to 2 cache lines (64 bytes total) beyond the current instruction execution point. Executing code within the last 64 bytes of a memory region such as internal SRAM or Flash may cause a bus error when pre-fetching occurs past the end of memory. Do not place code to be executed within the last 64 bytes of a memory region. - An ECC exception can occur if pre-fetches occur at locations that are valid but not yet initialized for ECC. When executing code from internal ECC SRAM, initialize memory beyond the end of the code until the next 32-byte aligned address and then an additional 64 bytes to ensure that pre-fetches cannot land in uninitialized SRAM. - The Boot Assist Module (BAM) is located at the end of the address space and so may cause instruction pre-fetches to wrap-around to address 0 in internal flash memory. If this first block of flash memory contains ECC errors, such as from an aborted program or erase operation, a machine-check exception will be asserted. At this point in the boot procedure, exceptions are disabled, but the machine-check will remain pending and the exception vector will be taken if user application code subsequently enables the machine check interrupt. To guard against the possibility of the BAM causing a machine-check exception to be taken, user application code should write all 1s to the Machine Check Syndrome Register (MCSR) to clear it before enabling the machine check interrupt." • Added NOTE: "Under certain conditions, if a static branch prediction and a dynamic return prediction (which uses the subroutine return address stack) occur simultaneously in the BTB, the e200z7 core can issue an errant fetch address to the memory system (instruction fetched from wrong address). This can only happen when the static branch prediction is "taken" but the branch actually resolves to "not taken". To prevent the issue from occurring, set BUCSR[BPRED] = 0b11 to configure static branch prediction to "not taken". This issue does not apply to VLE."
Section Appendix B, "Revision History"	Added Section B.1, "Changes between revisions 1 and 2"

B.2 Changes between revisions 0 and 1

Table B-2. Changes between revisions 0 and 1

Chapter	Description
Overall	Editorial changes
Chapter 1, “e200z7 Core Complex Overview”	Section 1.1, “e200z7 Overview” • Deleted paragraph “The e200z7 platform can be configured in specific processor implementations...” • In Figure 1-1, “e200z7 Block Diagram”, corrected “Single-Instruction, In-Order Dispatch” to “Dual-Instruction, In-Order Dispatch” and “Single-Instruction, In-Order Write Back” to “Dual-Instruction, In-Order Write Back” Section 1.1.1, “Features” • Changed “Branch acceleration using a branch target buffer (BTB)” to “Branch target prefetching using a branch target buffer (BTB)”. • Changed “Two-cycle load latency” to “Three-cycle load latency”. • Changed “Big- and little-endian support on a per-page basis” to “Big- and little-endian support”. Section 1.2.1, “Register Set” • In Figure 1-2, “e200z760 Supervisor Mode Programmer’s Model”, corrected misaligned register names for Memory Management Registers, Control & Configuration. Section 1.2.2, “Instruction Set” • In Table 1-2, “Scalar and Vector Embedded Floating-Point Instructions”, added missing floating-point instructions (efscfh/evfscfh, efscth/evfscth, efsmax/evfsmax, efsmin/evfsmin, efssqrt/evfssqrt, evfsaddsub, evfsaddsubx, evfsaddx, evfsdiffsum, evfsdiff, evfsmule, evfsmulo, evfsmulx, evfssubaddx, evfssubx, evfssubadd, evfssumdiff, and evfssum). Section 1.2.3.3, “Interrupt Types” • In Table 1-3, “Interrupt Types”, — For the Description of the Critical interrupts Category, changed “If the debug interrupt is not enabled, it is also treated as a critical interrupt.” to “If the debug feature is not enabled, a debug interrupt is treated as a critical interrupt.” — For the Programming Resources of the Debug interrupts Category, changed “Can be masked by the machine check enable bit, MSR[DE]. Includes the debug syndrome register (DBSR).” to “Can be masked by the debug interrupt enable bit, MSR[DE]. Includes the debug status register (DBSR).” Section 1.2.3.4, “Interrupt Registers” • In Table 1-4, “Interrupt Registers”, added entry for the DBSR (Debug status register). Section 1.3, “Microarchitecture Summary” • Changed “Prefetched instructions are placed into an instruction buffer capable of holding six instructions.” to “Prefetched instructions are placed into an instruction buffer.” • Deleted mistakenly repeated paragraph that starts with “Conditional branches which are not taken...” Section 1.3.2, “Integer Unit Features” • Changed “Divider logic for signed and unsigned divide in 4 to 15 clocks with minimized execution timing” to “Divider logic for signed and unsigned divide in 4 to 15 clocks with minimized execution timing (EU1 only)”. Section 1.3.5, “Memory Management Unit (MMU) Features”, deleted “Byte ordering (endianness) configurable on a per-page basis” Section 1.3.6, “System Bus (Core Complex Interface) Features” • Changed “32-bit address bus plus attributes and control on each bus” to “32-bit address bus, 64-bit data bus, plus attributes and control on each bus”. • Added “Support for HCLK running at a slower rate than CPU clock”.

Table B-2. Changes between revisions 0 and 1 (continued)

Chapter	Description
Chapter 2, “Register Model”	Chapter 2, “Register Model” • In Figure 2-1, “e200z760 Supervisor Mode Programmer’s Model”, corrected misaligned register names for Memory Management Registers, Control & Configuration. Section 2.4.1, “Machine State Register (MSR)” • In Figure 2-5, “Machine State Register (MSR)”, corrected bit 28 to “—” and bit 29 to “PMM”.
Chapter 3, “Instruction Model”	Section 3.13, “Enhanced Reservations” • Throughout, corrected code listings to display arrow symbols, subscripts, and superscripts properly, such as “if X-mode then EA ¨ 320 || (a + GPR(RB))32:63” to “if X-mode then EA ¨ 320 || (a + GPR(RB))32:63”.
Chapter 4, “Instruction Pipeline and Execution Timing”	Section 4.1, “Overview of Operation” • In Figure 4-1, “e200z7 Block Diagram”, corrected “Single-Instruction, In-Order Dispatch” to “Dual-Instruction, In-Order Dispatch” and “Single-Instruction, In-Order Write Back” to “Dual-Instruction, In-Order Write Back” Section 4.3.2, “Instruction Prefetch Buffers and Branch Target Buffer” • Changed “Certain other branches do not allocate BTB entries: blr, bclr, bctr, bcctr.” to “Certain other branches do not allocate BTB entries: bctr, bcctr.”
Chapter 5, “Embedded Floating-Point Unit”	Section 5.3.1.1, “Single-Precision Floating-point Format” • Added missing title to Figure 5-2, “Single-Precision Data Format”. Section 5.3.4, “Embedded Scalar Single-Precision Floating-Point Instructions” • For the evfscth instruction, corrected code listing as follows: — Changed line 15 “resulth ¨ fhsign || 151 // like-signed zero value “ to “resulth ¨ fhsign || 0b11110 || 100 // max value “. — Changed line 18 “resulth ¨ fhsign || 0b00001 || 100 // min value “ to “resulth ¨ fhsign || 150 // like-signed zero value “. Section 5.6, “Instruction Forms and Opcodes” In Table 5-16, “Embedded Vector Floating-Point Instruction Opcodes” and Table 5-17, “Embedded Scalar Single-Precision Floating-Point Instruction Opcodes” for the lines whose Instruction value is “—“, changed the value of Opcode Bits, 0-5 to “4”
Chapter 6, “Signal Processing Extension (SPE)”	Section 6.1, “Nomenclature and Conventions” Added “Due to historical precedent, the terms SPE and SIMD are sometimes used interchangeably.”
Chapter 7, “Interrupts and Exceptions”	Section 7.5, “Interrupt Vector Offset Registers (IVORxx)” • In Figure 7-5, “e200 Interrupt Vector Offset Register (IVOR)”, added missing “—” for bits 28-31. Section 7.6, “Interrupt Definitions” • In Table 7-8, “Interrupts”, changed IVOR Number “IVOR17”, “IVOR18”, “IVOR19”, and “IVOR20” to “IVOR32”, “IVOR33”, “IVOR34”, and “IVOR35”, respectively. Section 7.6.17, “System Reset Interrupt” • In Table 7-30, “System Reset Interrupt—Register Settings” — Added row for Register “CSRR0” with Setting Description “Undefined”. — Changed Vector notation “[p_rstbase[0:29]] || 2'b00” to “[p_rstbase[0:29]] || 0b00” for consistency. Section 7.7.1, “Exception Priorities” • In Table 7-35, “Zen Exception Priorities”, for entries whose Exception describes the Debug procedure, added numbers for Cause entries.

Table B-2. Changes between revisions 0 and 1 (continued)

Chapter	Description
Chapter 8, “Performance Monitor”	Section 8.3.7, “Local Control B Registers (PMLCb0–PMLCb3)” • In Table 8-6, “PMLCb0–PMLCb3 Field Descriptions”, for Bits “9:12” and “14:17”, in the Description field, changed “1100Trigger-on based on watchpoint #28 occurrence”, “1101Trigger-on based on watchpoint #29 occurrence”, and “1110-1111 Reserved” to “1000Trigger-on based on watchpoint #24 occurrence”, “1001Trigger-on based on watchpoint #25 occurrence”, and “1100-1111 Reserved”, respectively. Section 8.7, “Event Selection” • In Table 8-10, “Performance Monitor Event Selection”, for items Number “Com:146”, “Com:147”, and “Com:148”, changed all values to “—”.

Table B-2. Changes between revisions 0 and 1 (continued)

Chapter	Description
Chapter 9, “L1 Cache”	Section 9.3, “Cache Lookup” • Changed “Subsequent double-words may be streamed to the CPU if they have been requested and streaming is enabled via the L1CSR0 register, ...” to “Subsequent double-words may be streamed to the CPU if they have been requested, ...”. Section 9.4.2, “L1 Cache Control and Status Register 1 (L1CSR1)” • Added “The SPR number for L1CSR1 is 1011 in decimal.” • In Table 9-2, “L1CSR1 Field Descriptions”, deleted Bits “14” row, ISTRM. Section 9.4.4, “L1 Cache Configuration Register 1 (L1CFG1)” • Changed “e200z7” to “ e200z760n3”. • In Figure 9-7, “L1 Cache Configuration Register 1 (L1CFG1)”, and changed “Access: Read/Write” to “Access: Read only”. Section 9.7.4, “Cache Miss Access Ordering” • Changed “e200z7” to “ e200z760n3”. Section 9.7.9.2, “L1 Flush/Invalidate Register 1 (L1FINV1)” • Corrected title of Figure 9-9, “L1 Flush/Invalidate Register 1 (L1FINV1)”. Section 9.9, “Push and Store Buffers” • Deleted “For the AXI interface version, the burst write bus transaction to write the contents of the push buffer into memory is performed in parallel with miss linefill operations, without waiting for the linefill to complete.” Section 9.12, “Cache Line Locking/Unlocking Unit” • In Table 9-9, “Cache Line Locking/Unlocking Unit Instructions”, corrected Acronym “dcbtsls” to “dcbtstls”. Section 9.12.2, “Instruction Details” • In Table 9-10, “Cache Line Locking/Unlocking Unit Instructions”, corrected Acronym “dcbtsls” to “dcbtstls”. Section 9.13.1, “Exception Conditions for Cache Instructions” • Added the following Notes to Table 9-11, “Special Case Handling”: “Notes: - Priority decreases from left to right - Cache operations that do not set or clear locks ignore the value of the CT field - “dash” indicates executes normally - “NOP” indicates treated as a no-op - DSI = data storage interrupt; ALI = alignment interrupt; DTLB = data TLB interrupt - DCUL, ICUL = no-op, and set L1CSR0[CUL] - DCLO, ICLO = no-op, and set L1CSR0[CLO] - DLK, ILK = data storage interrupt (DSI) and set ESR[DLK] or ESR[ILK] - MC = Machine Check and update MCAR” Section 9.13.2, “Transfer Type Encodings for Cache Management Instructions” • In Table 9-12, “Transfer Type Encoding”, added “[1]” to table header “p_d_ttype[0:5]” and added footnote “[1] p_ttype[5] ’e’ is set to set to 0.” Section 9.19.3, “Cache Debug Access Control Register (CDACNTL)” • In Figure 9-10, “Cache Debug Access Control Register (CDACNTL)”, and added DCR number “DCR 351”. Section 9.19.3.1, “Cache Debug Access Data Register (CDADATA)” • In Figure 9-11, “Cache Debug Access Data Register (CDADATA)”, and added DCR number “DCR 350”. Added new Section 9.20, “Hardware Debug (Cache) Control Register 0”.

Table B-2. Changes between revisions 0 and 1 (continued)

Chapter	Description
Chapter 10, “Memory Management Unit”	Section 10.4.1, “MMU Configuration Register (MMUCFG)” • In Figure 10-4, “MMU Configuration Register (MMUCFG)”, deleted Reset value “All zeros”. Section 10.4.3, “TLB1 Configuration Register (TLB1CFG)” • In Figure 10-6, “TLB1 Configuration Register (TLB1CFG)”, deleted Reset value “All zeros”. Section 10.7.3, “MMU Assist Registers (MAS)” • In Figure 10-10, “MMU Assist Register 1 (MAS1)”, added missing field name for bits 20-24 “TSIZE”. Section 10.9.1, “Transfer Type Encodings for MMU Control Instructions” • In Table 10-16, “Transfer Type Encoding”, added “[1]” to table header “p_d_ttype[0:5]” and added footnote “[1] p_ttype[5] ’e’ is set to set to 0.” Section 10.11, “External Translation Alterations for Realtime Systems” • Corrected “Those entries within entries 0–15 programmed with a TID value of 0b1111nm11l...” to “Those entries within entries 0–15 programmed with a TID value of 0b1111nm11...”

Table B-2. Changes between revisions 0 and 1 (continued)

Chapter	Description
Chapter 11, "External Core Complex Interfaces"	Section 11.1, "Signal Index" • In Table 11-1, "Interface Signal Definitions", added section "External Translation Alteration Signals" with two entries, Signal Name "p_extpid_en" and "p_extpid[6:7]". • For Signal Name "p_stop", added "—" for Reset Value. • For "JTAG-Related Signals" section, added entry Signal Name "j_key_in". Section 11.2.13.1, "Cache Tag Error Out (p_[d,i]_cache_tagerr_out)" • Added "When L1CSR0[DCEA]/L1CSR1[ICEA] indicates machine check generation on error, assertion of this signal indicates a machine check will be signaled for the access, or for dcbi/icbi operations, indicates that a remote invalidation of one or more cache lines should occur. When L1CSR0[DCEA]/L1CSR1[ICEA] indicates auto-invalidation on error, assertion of this signal indicates that the cache will insert an additional cycle to perform auto-invalidation on cache ways with uncorrectable tag errors, and to correct tags in ways with correctable errors." Added new section Section 11.2.14, "External Translation Alteration Signals" with subsections Section 11.2.14.1, "External PID Enable (p_extpid_en)" and Section 11.2.14.2, "External PID In (p_extpid[6:7])". Section 11.2.22.6, "Watchpoint Events (jd_watchpt[0:26])" • Corrected title from "jd_watchpt[0:29]" to "jd_watchpt[0:26]". • Deleted sentence "DEVNT-, DTC-based, and performance monitor watchpoints are also supported." Section 11.2.24, "Development Support (Nexus 3) Signals" • In Table 11-21, "e200 Development Support (Nexus) Signals", changed Signal "nex_wevto[3:0]" to "nex_wevto[2:0]". Section 11.2.25.5, "JTAG/OnCE Test Reset (j_trst_b)" • In Table 11-23, "JTAG Signals Used to Support External Registers", added entry Signal Name "j_key_in". Added new Section 11.2.25.12, "Key Data In (j_key_in)". Section 11.3.2.1, "Basic Read Transfer Cycles" • In Figure 11-7, "Basic Read Transfers", added caption "Single cycle reads, full pipelining**".** Section 11.3.2.7, "Burst Accesses" • In Figure 11-25, "Burst Read Error Termination, Burst Write Substituted", added caption "Burst Read with error termination, Burst write**".** Section 11.3.3, "Memory Synchronization Control Operation", inserted missing figures Figure 11-28, "Memory Sync Operation (snoop queue empty)" and Figure 11-29, "Memory Sync Operation (2nd msync back-to-back)". Section 11.3.5.1, "Stop Mode Entry/Exit and Snoop Ready Signaling" • In Figure 11-54, "Stop Mode Exit, p_snp_rdy Operation", added caption "Snoop interface operation - stop mode exit operation". Section 11.3.6.1, "Debug Entry Cross-signaling" • In Figure 11-55, "Debug Entry Cross-Signaling Interface, non-Lockstep Mode", added caption "Debug exit, non-Lockstep operation, CPU 0 sees OCMD "go" first, debug mode exit not synchronized".

Table B-2. Changes between revisions 0 and 1 (continued)

Chapter	Description
Chapter 13, “Debug Support”	Section 13.3.3.1, “Debug Control Register 0 (DBCR0)” • Added to footnote to Figure 13-4, “DBCR0 Register” “If DBCR0[EDM]=1, DBERC0 masks off hardware-owned resources (other than RST) from reset by p_reset_b, and only software-owned resources indicated by DBERC0 and the DBCR0[RST] field will be reset by p_reset_b.” • In Table 13-6, “DBCR0 Bit Definitions” — For the Descritpion of Bit “0”, changed “{DBCR0–6, DBCNT, IAC1–8, DAC1–2} “ to “{DBCR0, DBCNT, IAC1, DAC1–2} “. — For the Descritpion of Bit “1”, changed “Debug events do not affect DBSR unless EDM is set. “ to “Debug events do not affect DBSR“. Section 13.3.3.8, “Debug Status Register (DBSR)” • In Figure 13-11, “DBSR Register” changed bit field name “IAC4” to “IAC4-8”. Section 13.4, “External Debug Support” • In NOTE, changed “EDBCR0[EDM]” to "EDBCR0[EDM]/DBCR0[EDM]". Section 13.4.5.5, “Watchpoint Events (jd_watchpt[0:29])” • Changed “Watchpoint events are conditioned by the settings in the DBCR0, DBCR1, and DBCR2 registers, as well as by the DEVENT register, the DTC/DTSA/DTEA registers, and the Performance Monitor control register settings” to “Watchpoint events are conditioned by the settings in the DBCR0, DBCR1, and DBCR2 registers, as well as by the DEVENT register, and the Performance Monitor control register settings”. Section 13.4.6.2, “e200 OnCE Command Register (OCMD)” • Changed “EDBCR0[EDM]” to "EDBCR0[EDM]/DBCR0[EDM]". Section 13.5, “Watchpoint Support” • Changed “Certain watchpoints (DEVNT-based and DTC-based)...” to “Certain watchpoints (DEVNT-based)...” Changed “The DEVNT-based and DTC-based watchpoints are...” to “The DEVNT-based watchpoints are...”
Chapter 14, “Nexus 3 Module”	Chapter 14, “Nexus 3 Module” • Throughout chapter, changed “the IEEE-ISTO 5001-2008 standard” to “the IEEE-ISTO 5001 standard” except for certain occurrences. Section 14.1.1, “Terms and Definitions” • In Table 14-1, “Terms and Definitions” changed Term “Nexus3” to “Nexus1”. Section 14.1.2, “Feature List” • Changed “Four (4) additional Watchpoint Event output pins (nex_wevto[3:0]) for SoC use” to “Three (3) additional Watchpoint Event output pins (nex_wevto[2:0]) for SoC use” Section 14.4.4, “Nexus Development Control Register 2 (DC2)” • In Figure 14-5, “Development Control Register 2 (DC2)” changed “WEVTO[3]C” to “—”. • In Table 14-12, “Development Control Register 2 Fields” changed Bits “DC2[31-24]” Name “EWC” to Bits “31-28” Name “—” and Description to “Reserved”. Section 14.4.5, “Nexus Development Control Register 3 (DC3)” • In Figure 14-6, “Development Control Register 3 (DC3)” changed “WEVTO[3]C” to “—”. • In Table 14-13, “Development Control Register 3 Fields” changed Bits “31-28” from Name “WEVTO[3]C” to “—” and Description to “Reserved”.

Table B-2. Changes between revisions 0 and 1 (continued)

Chapter	Description
Chapter 14, "Nexus 3 Module" (continued)	Section 14.4.8, "Watchpoint Trigger Registers (WT, PTSTC, PTETC, DTSTC, DTETC)" • In Figure 14-10, "Program Trace Start Trigger Control (PTSTC) Register", Figure 14-11, "Program Trace End Trigger Control (PTETC) Register", Figure 14-12, "Data Trace Start Trigger Control (DTSTC) Register", and Figure 14-13, "Data Trace End Trigger Control (DTETC) Register", moved start of PTST, PTET, DTST, and DTET bit field, respectively, from bit 29 to 26. • In Table 14-17, "Program Trace Start Trigger Control Register Fields", Table 14-18, "Program Trace End Trigger Control Register Fields", Table 14-19, "Data Trace Start Trigger Control Register Fields", and Table 14-20, "Data Trace End Trigger Control Register Fields", changed start of PTST, PTET, DTST, and DTET bit field, respectively, from bit 29 to 26. Changed Description to represent 27 bits. Section 14.4.9, "Nexus Watchpoint Mask Register (WMSK)" • In Figure 14-14, "Watchpoint Mask Register", moved start of WEM bit field from bit 29 to 26. • In Table 14-21, "Watchpoint Mask Register Fields", changed start of WEM bit field from bit 29 to 26. Changed Description to represent 27 bits. Section 14.4.14, "Read/Write Access Control/Status (RWCS)" • In Table 14-25, "Read/Write Access Control/Status Register Fields", corrected Bits "RWCS[21-16]" to "RWCS[17-16]". Section 14.7.4, "Nexus Message Priority" • In Table 14-29, "Message Type Priority and Message Dropped Responses", added footnote for Message Dropped Response of 5th entry: "Message will always be dropped if program trace is enabled, and program correlation messages for PID0 /mtmsr IS messages are not masked (Event Code = 0101). No error message is sent for this case since the PID value is contained in the higher priority message." Section 14.11.3.10, "Program Trace Synchronization Messages" • In Table 14-34, "Program Trace Exception Summary", for Exectpion Condition "Collision Priority" changed the Instruction 0 priority description from "WPM → PCM[PIDMSG] → DQM ..." to "WPM → DQM → PCM[PIDMSG]...". Section 14.11.5, "Program Trace Timing Diagrams (2 MDO/1 MSEO Configuration)" • In Figure 14-33, "Program Trace—Indirect Branch Message (Traditional)", for MDO[1:0] changed the 6th value from "00" to "10". • In Figure 14-34, "Program Trace—Indirect Branch Message (History)", for MDO[1:0] changed the 6th value from "00" to "01". Section 14.12.4, "Data Trace Timing Diagrams (8 MDO/2 MSEO Configuration)" • In Figure 14-40, "Data Trace—Data Write Message", for MDO[7:0] changed the 2nd value from "01001000" to "10010100" and changed the 3rd value from "000101001" to "01010010". • In Figure 14-41, "Data Trace—Data Read with Sync Message", deleted "addr type = 0, " Section 14.13.3, "Data Acquisition Trace Event" • Added Figure 14-42, "Data Acquisition Message Format". Section 14.14, "Watchpoint Trace Messaging" • In Figure 14-43, "Watchpoint Message Format", changed "(1–30 bits)" to "(1–27 bits)" and "Variable length = 11–40 bits" to "Variable length = 11–37 bits". In Table 14-37, "Watchpoint Source Encoding", changed column header "Watchpoint Source (1-30 bits)" to "Watchpoint Source (1-27 bits)". Changed Description to represent 27 bits.

Table B-2. Changes between revisions 0 and 1 (continued)

Chapter	Description
Chapter 14, "Nexus 3 Module" (continued	Section 14.14.1, "Watchpoint Timing Diagram (2 MDO/1 MSEO configuration)" • Changed title of Figure 14-44, "Watchpoint Message and Watchpoint Error Message" to "Watchpoint Message and Watchpoint Error Message". Section 14.15, "Nexus 3 Read/Write Access to Memory-Mapped Resources" • Added paragraph "Nexus 3 read/write accesses are run as privileged data non-cacheable, non-global accesses by default, and drive the p_d_hprot[5:0] bus access attributes to 0b000011 and the p_d_gbl access attribute to 0 accordingly. The RWCS[ATTR] field is provided to allow a portion of these default values to be modified when performing read or write accesses using the Nexus 3 Read/Write access mechanism." Section 14.16.1, "Pins Implemented" • Changed "(4) nex_wevto[3:0] pins" to "(3) nex_wevto[2:0] pins" • In Table 14-39, "Nexus 3 Auxiliary Pins", changed "nex_wevto[3:0]" to "nex_wevto[2:0]" and "Watchpoint Event Out 3–0" to "Watchpoint Event Out 2–0".
Section Appendix A, "Register Summary"	Section Appendix A, "Register Summary" • In Figure A-1, "e200z760 Supervisor Mode Programmer's Model", corrected misaligned register names for Memory Management Registers, Control & Configuration. • In Figure A-5, "Machine State Register (MSR)", corrected bit 28 to "—" and bit 29 to "PMM". • In Figure A-18, "e200 Interrupt Vector Offset Register (IVOR)", added missing "—" for bits 28-31. • In Figure A-26, "L1 Cache Configuration Register 1 (L1CFG1)", changed "Access: Read/Write" to "Access: Read only". • Corrected title of Figure A-28, "L1 Flush/Invalidate Register 1 (L1FINV1)". • In Figure A-29, "MMU Configuration Register (MMUCFG)", deleted Reset value "All zeros". • In Figure A-31, "TLB1 Configuration Register (TLB1CFG)", deleted Reset value "All zeros". • In Figure A-35, "MMU Assist Register 1 (MAS1)", added missing field name for bits 20-24 "TSIZE". • Added to footnote to Figure A-42, "DBCR0 Register" "If DBCR0[EDM]=1, DBERC0 masks off hardware-owned resources (other than RST) from reset by p_reset_b, and only software-owned resources indicated by DBERC0 and the DBCR0[RST] field will be reset by p_reset_b." • In Figure A-49, "DBSR Register", changed bit field name "IAC4" to "IAC4-8".
Section Appendix B, "Revision History"	Added Section B.1, "Changes between revisions 0 and 1"