e200z4 Power Architecture™ Core Reference Manual

Supports
e200z446n3

e200z4RM
Rev. 0
10/2009

freescale™
semiconductor

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
P.O. Box 5405
Denver, Colorado 80217
+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.

Contents

Paragraph Number | Title | Page Number

Chapter 1
e200z4 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

Chapter 5
Interrupts and Exceptions

Contents

Contents

Paragraph Number | Title | Page Number

Chapter 6
Embedded Floating-Point Unit, Version 2

Chapter 7
Signal Processing Extension Unit

Contents

Paragraph Number | Title | Page Number

Chapter 8
Power Management

Chapter 9
L1 Cache

Contents

Paragraph Number | Title | Page Number

Contents

Paragraph Number | Title | Page Number

Chapter 10
Memory Management Unit

Contents

Paragraph Number | Title | Page Number

Chapter 11
Debug Support

Contents

Paragraph Number | Title | Page Number

Contents

Paragraph Number | Title | Page Number

Chapter 12 Nexus 3+ Module

Contents

Contents

Paragraph Number	Title	Page Number

Chapter 13 External Core Complex Interfaces

Contents

Paragraph Number | Title | Page Number

Contents

Contents

Contents

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

Appendix A
Register Summary

Appendix B
Revision History

Contents

Paragraph Number	Title	Page Number

Figures

Figures

Figures

Figure Number | Title | Page Number

Figures

Figure Number | Title | Page Number

Figures

Figure Number | Title | Page Number

Figures

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

Figures

Figure Number | Title | Page Number

Figures

Figure Number | Title | Page Number

Figures

Figure Number | Title | Page Number

Figures

Figure Number | Title | Page Number

Tables

Tables

Tables

Tables

Tables

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

Tables

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

About This Book

The primary objective of this manual is to describe the functionality of the e200z4 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 “Auxilliary 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.
- e200z4 implementation details—The next level of architectural specificity describes those features that are shared across the cores in the e200z4 subfamily but that may be in the other members of the e200 product line.
- e200z4xx implementation details—The e200z4 subfamily will eventually include one or more specific cores with unique combinations of functionality. Each processor core in the e200z4 product line, such as the e200z450n3 for example, typically 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 e200z4 cores, they are specifically noted as such.

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

Audience

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

Organization

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

- Chapter 1, “e200z4 Core Complex Overview,” provides a general description of e200z4 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 e200z4.
- Chapter 5, “Interrupts and Exceptions,” describes how the e200z4 implements the interrupt model as it is defined by the Power ISA embedded category architecture.
- Chapter 6, “Embedded Floating-Point Unit, Version 2,” describes the instruction set architecture of the Embedded Floating-point (EFPU) implemented on the e200z4. This unit implements scalar and vector single-precision floating-point instructions to accelerate signal processing and other algorithms. The e200z446n3 implements version 2 of the Embedded Floating-Point Unit (EFPU2).
- Chapter 7, “Signal Processing Extension Unit,” describes the instruction set architecture of the SPE and implements instructions to accelerate signal processing and other algorithms.
- Chapter 8, “Power Management,” describes the power management facilities as they are defined by the Power ISA embedded category architecture and implemented in the e200z4 core.
- Chapter 9, “L1 Cache,” This chapter 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 e200z4 MMU implementation.
- Chapter 11, “Debug Support,” describes the internal debug facilities as they are implemented in the e200z4 core.
- Chapter 12, “Nexus 3+ Module,” describes the Nexus 3+ module, which provides real-time development capabilities for e200z4 processors in compliance with the proposed IEEE-ISTO Nexus 5001-2008™ standard.
- Chapter 13, “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.
- Appendix A, “Register Summary,” contains the register diagrams for the 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 2.06,* by Power.org™, 2009, 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
OnCE	On-chip emulation logic
PTE	Page table entry
PVR	Processor version register

Table i. Acronyms and Abbreviated Terms (continued)

Term	Meaning
RISC	Reduced instruction set computing
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
e200z4 Core Complex Overview

This chapter provides an overview of the e200z4 microprocessor core built on Power Architecture™ technology for embedded processors. It includes the following:

- An overview of the core, including the block diagram (Figure 1-1)
- A summary of the feature set for this core (see Section 1.2, “Features”)
  - — A description of the execution units (see Section 1.2.1, “Execution Unit Features”)
  - — A description of the memory management architecture (see Section 1.2.3, “Memory Management Unit Features”)
  - — High-level details of the core memory and coherency model (see Section 1.2.4, “System Bus (Core Complex Interface) Features”)
  - — High-level details of the Nexus 3+ features (see Section 1.2.5, “Nexus 3+ Features”)
- A summary of the programming model for this core (see Section 1.3, “Programming Model”)
  - — An overview of the register set (see Section 1.3.1, “Register Set”)
  - — An overview of the instruction set (see Section 1.3.2, “Instruction Set”)
  - — An overview of interrupts and exception handling (see Section 1.3.3, “Interrupts and Exception Handling”)
- A summary of instruction pipeline and flow (see Section 1.4, “Microarchitecture Summary”)

1.1 Overview

The e200z4 processor family is a set of CPU cores that implement low-cost versions of Power Architecture technology. The e200z4 core is a dual-issue, 32-bit design with 64-bit general-purpose registers (GPRs). The e200z446n3 integrates an e200z4 CPU core, a memory management unit (MMU), a 4-Kbyte instruction cache, and a Nexus Class 3+ real-time debug unit. Separate instruction and data AHB 2.v6 system interfaces are provided.

The e200z4 is compliant with the PowerPC™ instruction set architecture (ISA). It does not support Power ISA floating-point instructions in hardware, but traps them so they can be emulated by software.

Instructions of the embedded floating-point category are provided to support real-time single-precision embedded numerics operations using the general-purpose registers.

Instructions of the signal processing extension (SPE) category are provided to support real-time SIMD fixed-point and single-precision embedded numerics operations using the general-purpose registers. All arithmetic instructions that execute in the core operate on data in the general-purpose registers (GPRs). The GPRs have been extended to 64-bits in order to support vector instructions defined by the SPE category. These instructions operate on a vector pair of 16-bit or 32-bit data types and deliver vector and scalar results.

In addition to the base Power ISA embedded category instruction set, the core also implements the variable-length encoding category (VLE), which provides improved code density. See the *EREF* and supplementary VLE Programming Environments Manual (*VLEPEM)* for more information about the VLE extension.

The processor integrates a pair of integer execution units, a branch control unit, instruction fetch unit and load/store unit, and a multi-ported register file capable of sustaining six read and three write operations per clock cycle. Most integer instructions execute in a single clock cycle. Branch target prefetching is performed by the branch unit to allow single-cycle branches in many cases.

Throughout the remainder of this document, the core is referred to as the “e200z4” when speaking of e200z4-specific implementations, the “e200z4xx” when speaking of a specific variety of e200z4 core, or “e200” when referring to the whole e200 family.

Figure 1-1 shows the block diagram for the device.

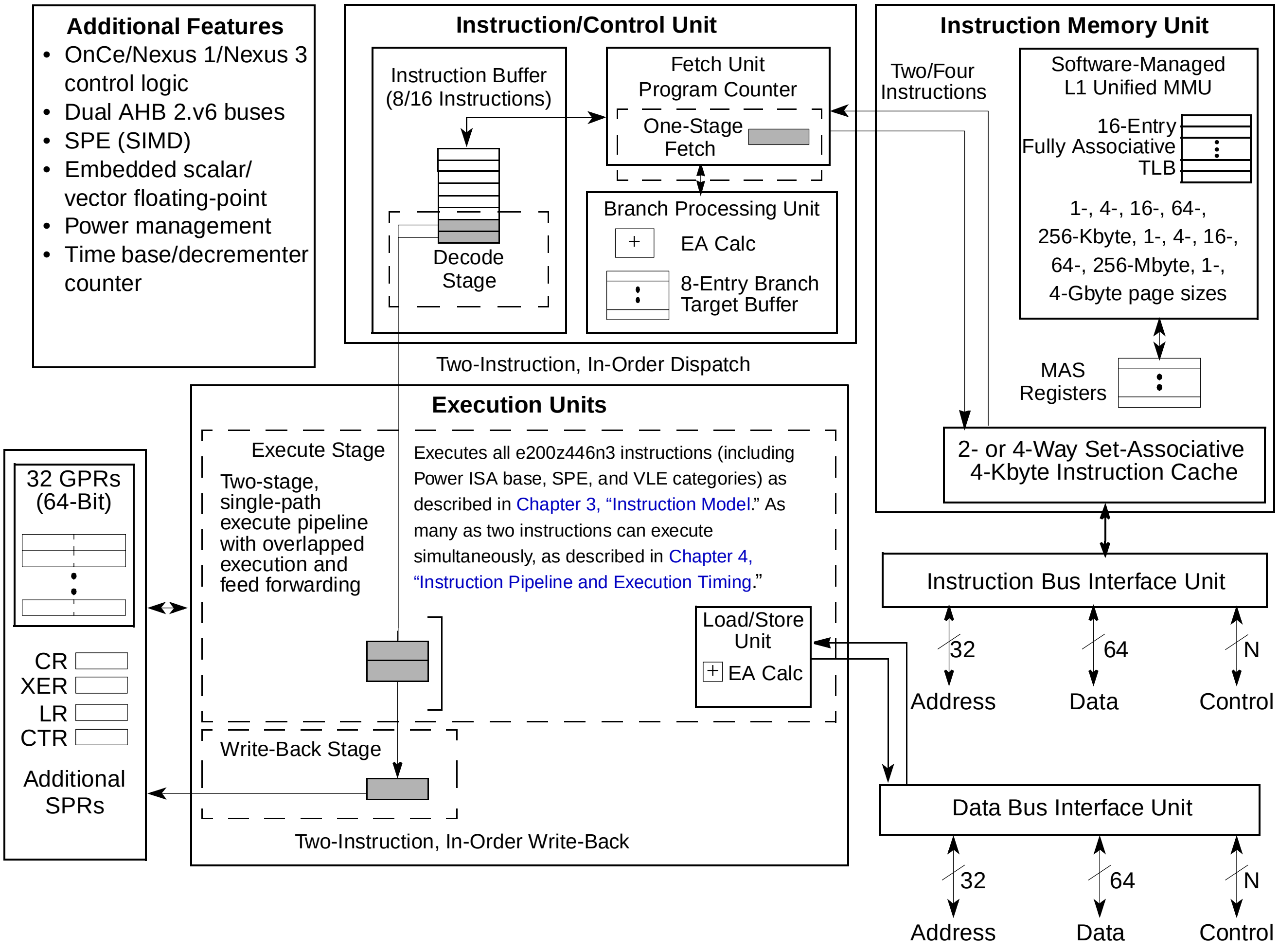

Additional Features
• OnCe/Nexus 1/Nexus 3 control logic
• Dual AHB 2.v6 buses
• SPE (SIMD)
• Embedded scalar/ vector floating-point
• Power management
• Time base/decrementer counter
Instruction/Control Unit
Instruction Buffer (8/16 Instructions)
Decode Stage
Fetch Unit
Program Counter
One-Stage Fetch
Branch Processing Unit
+ EA Calc
8-Entry Branch Target Buffer
Instruction Memory Unit
Two/Four Instructions
Software-Managed L1 Unified MMU
16-Entry Fully Associative TLB
1-, 4-, 16-, 64-, 256-Kbyte, 1-, 4-, 16-, 64-, 256-Mbyte, 1-, 4-Gbyte page sizes
MAS Registers
2- or 4-Way Set-Associative 4-Kbyte Instruction Cache
Two-Instruction, In-Order Dispatch
Execution Units
Execute Stage
Two-stage, single-path execute pipeline with overlapped execution and feed forwarding
Executes all e200z446n3 instructions (including Power ISA base, SPE, and VLE categories) as described in Chapter 3, “Instruction Model.” As many as two instructions can execute simultaneously, as described in Chapter 4, “Instruction Pipeline and Execution Timing.”
Load/Store Unit
+ EA Calc
Write-Back Stage
Two-Instruction, In-Order Write-Back
32 GPRs (64-Bit)
CR
XER
LR
CTR
Additional SPRs
Instruction Bus Interface Unit
32
64
N
Address
Data
Control
Data Bus Interface Unit
32
64
N
Address
Data
Control

Figure 1-1. e200z446n3 Block Diagram

1.2 Features

Key features of the e200z446n3 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 an 8-entry branch target buffer (BTB)
- Supports independent instruction and data accesses to different memory subsystems, such as SRAM and flash memory by means of independent instruction and data bus interface units.
- Load/store unit
- 64-bit general-purpose register file
- Dual advanced high-performance (AHB) 2.v6 64-bit system buses
- Memory management unit (MMU) with 16-entry fully associative TLB and multiple page-size support
- 4 Kbyte, 2/4-way set-associative instruction cache
- Signal processing extension unit, version 1.1 supporting SIMD fixed-point operations using the 64-bit general-purpose register file.
- Embedded floating-point (FPU) unit, version 2 supporting scalar and vector SIMD single-precision floating-point operations using the 64-bit general-purpose register file.
- Nexus Class 3+ real-time development unit
- Power management
  - Low power design—extensive clock gating
  - Power saving modes: doze, nap, sleep, wait
  - Dynamic power management of execution units, cache, and MMU
- Testability
  - Synthesizeable, MuxD scan design
  - ABIST/MBIST for arrays
  - Built-in parallel signature unit

See the following sections for more details about specific units.

1.2.1 Execution Unit Features

The following subsections describes the execution units’ main features.

1.2.1.1 Instruction Unit Features

The instruction unit features the following:

- 64-bit path to cache supports fetching of two 32-bit Power ISA instructions or four 16-bit VLE instructions per clock cycle.
- Instruction buffer holds up to eight 32-bit Power ISA instructions or sixteen 16-bit VLE instructions.
- Dedicated program counter (PC) incrementer supports instruction prefetches.
- Branch unit with dedicated branch address adder and branch target buffer supports single-cycle execution of successfully predicted branches.

1.2.1.2 Integer Unit Features

The integer units feature support for single-cycle execution of most integer instructions, as follows:

- 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 ≤ 14 clock cycles with minimized execution timing (integer unit 1 only)
- Pipelined 32×32 hardware multiplier array supports $32 \times 32 \rightarrow 32$ multiply with 2 clock latency, 1 clock throughput

1.2.1.3 Load/Store Unit Features

The load/store unit supports load, store, and load multiple/store multiple instructions by means of the following:

- 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 two registers per cycle for load multiple and store multiple word instructions
- Fully pipelined
- Two-cycle load latency
- Big- and little-endian support
- Misaligned access support

1.2.2 L1 Cache Features

The L1 cache features the following:

- 4 Kbyte, 2- or 4-way configurable set-associative instruction cache
- 64-bit data, 32-bit address bus plus attributes and control

- 32-byte line size
- Cache line locking
- Way allocation
- Tag and data parity or multi-bit EDC protection with correction/auto-invalidation capability
- Virtually indexed, physically tagged
- Pseudo round-robin replacement algorithm
- Line-fill buffer
- Hit under fill
- Supports tag and data parity
- Supports tag and data double error detection
- Correction/auto-invalidation capability

1.2.3 Memory Management Unit Features

The memory management unit features the following:

- Virtual memory support
- 32-bit virtual and physical addresses
- 8-bit process identifier
- 16-entry fully associative TLB
- Hardware assist for TLB miss exceptions
- Per-entry multiple page size support from 1 Kbyte to 4 Gbyte
- Entry flush protection
- Software managed by **tlbre**, **tlbwe**, **tlbsx**, **tlbsync**, and **tlbivax** instructions
- Freescale EIS MMU architecture compliant
- Support for external control of entry matching for a subset of TID values to support non-intrusive runtime mapping modifications

1.2.4 System Bus (Core Complex Interface) Features

The core complex interface features the following:

- Independent instruction and data buses
- Advanced microcontroller bus architecture (AMBA) AHB 2.v6 protocol
- 32-bit address bus, 64-bit data bus, plus attributes and control
- Separate unidirectional 64-bit read and write data buses
- Support for HCLK running at a slower rate than CPU clock

1.2.5 Nexus 3+ Features

The Nexus 3+ module provides real-time development capabilities for e200z4 processors in compliance with the IEEE-ISTO 5001-2008™ standard. The ‘3+’ suffix indicates that some Nexus Class 4 features are available. A portion of the pin interface (the JTAG port) is also shared with the OnCE/Nexus 1 unit.

The following features are implemented:

- Program trace by means of branch trace messaging.
  - 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 by means of data write messaging and data read messaging.
  - Provides the capability for the development tool to trace reads and/or writes to selected internal memory resources.
- Ownership trace by means of ownership trace messaging (OTM).
  - OTM facilitates ownership trace by providing visibility of which process ID or operating system task is activated.An ownership trace message is transmitted when a new process/task is activated, allowing the development tool to trace ownership flow.
  - Allows enhanced download/upload capabilities.
- Data acquisition messaging
  - Allows code to be instrumented to export customized information to the Nexus auxiliary output port.
- Watchpoint messaging by means of the auxiliary interface
- Watchpoint trigger enable of program and/or data trace messaging
- Run-time access to the processor memory map by means of the JTAG port
- Auxiliary interface for higher data input/output
  - Configurable (min/max) message data out pins (**nex_mdo[n:0]**)
  - One or two message start/end out pins (**nex_mseo_b[1:0]**)
  - One read/write ready pin (**nex_rdy_b**) pin
  - One watchpoint event pin (**nex_evto_b**)
  - Three additional watchpoint event output pins (**nex_wevt[2:0]**) for SoC use
  - One event-in pin (**nex_evti_b**)
  - One MCKO (Message Clock Out) pin

All features are controllable and configurable by means of the JTAG port.

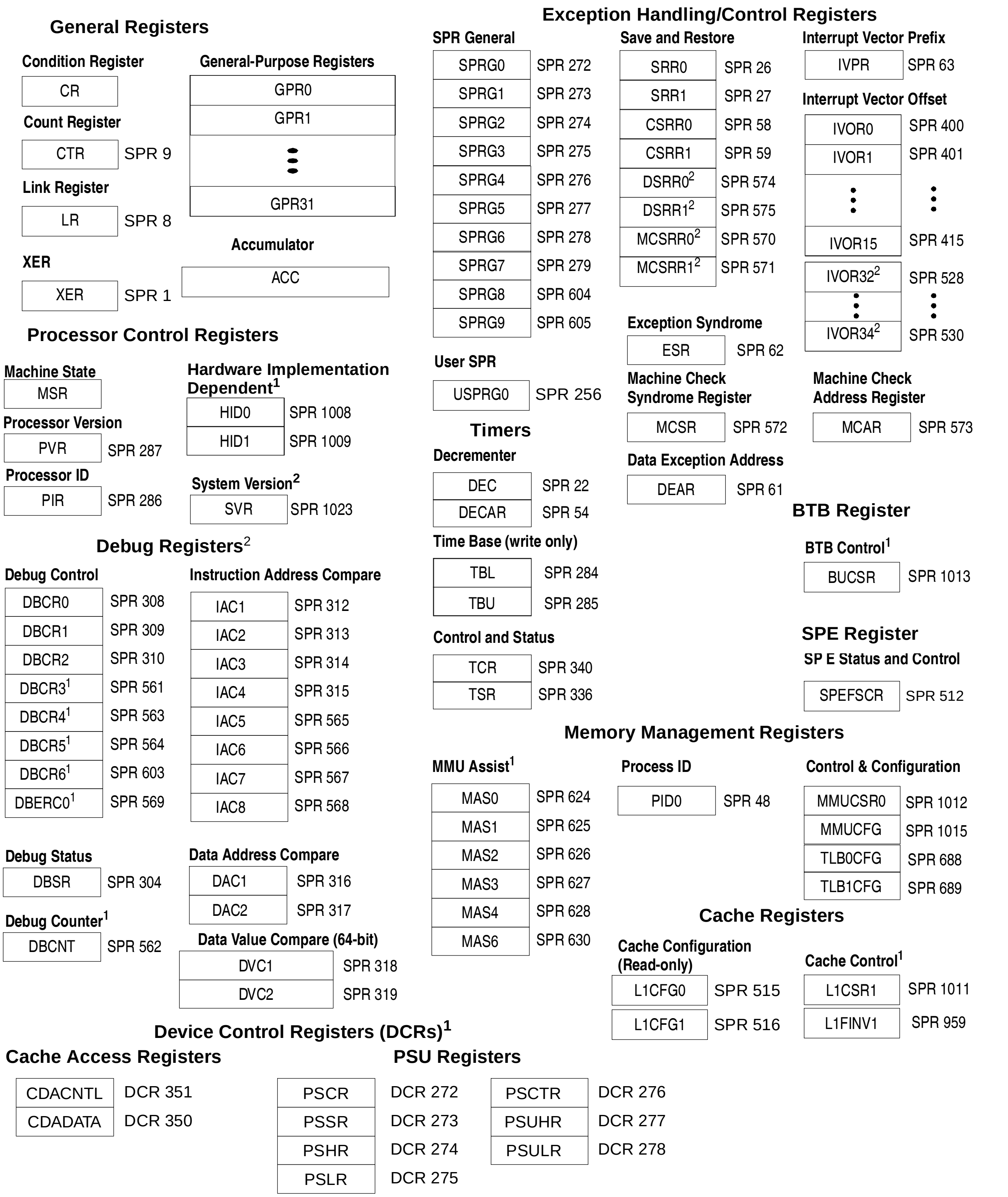
1.3 Programming Model

This section describes the register model, instruction model, and the interrupt model as they are defined by the Power ISA, Freescale EIS, and the e200z446n3 implementation.

1.3.1 Register Set

Figure 1-2 and Figure 1-3 show the complete e200z446n3 register set, including the sets of the registers that are accessible in supervisor mode and the set of registers that are accessible in user mode. 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 registers that can be accessed by supervisor-level software. User-level software can access only those registers listed in Figure 1-3.



1 - These e200-specific registers may not be supported by other processors built on Power Architecture technology
2 - Optional registers defined by the Power ISA embedded architecture
3 - Read-only registers

Figure 1-2. e200z446n3 Supervisor Mode Programmer's Model

Figure 1-3 shows the user-mode special-purpose registers.

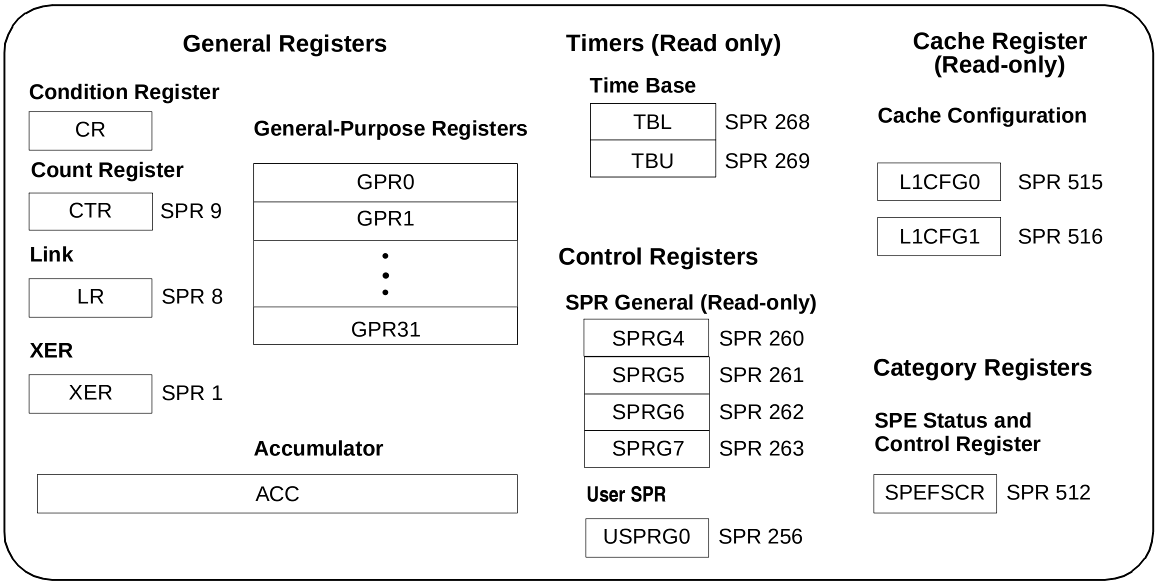


Figure 1-3. e200z446n3 User Mode Programmer's Model SPRs

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. Access to other registers can also be implicit, as part of the execution of an instruction. Some registers are accessed both explicitly and implicitly.

1.3.2 Instruction Set

The e200z4 supports 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 e200z4 does not include the Power ISA floating-point, load string, or store string instructions.

The e200z446n3 core implements the following architectural extensions:

- The VLE category
- The integer select category (ISEL)
- Enhanced debug and the debug notify halt instruction categories
- The machine check category
- The WAIT category
- The volatile context save/restore category
- The embedded floating-point unit, version 2
- The signal processing extension unit, version 1.1
- The cache line locking category
- The enhanced reservations category

1.3.3 Interrupts and Exception Handling

The e200z4 core supports an extended exception handling model with nested interrupt capability and extensive interrupt vector programmability. 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 begins 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) to verify the specific cause of the exception and take appropriate action.

The core complex supports the interrupts described in Table 1-1.

Table 1-1. Interrupt Registers

Register	Description
Noncritical Interrupt Registers	
SRR0	Save/restore register 0—On noncritical interrupts, stores either the address of the instruction causing the exception or the address of the instruction that executes after the **rfi** instruction.
SRR1	Save/restore register 1—Saves machine state on noncritical interrupts and restores machine state after an **rfi** instruction is executed.
Critical Interrupt Registers	
CSRR0	Critical save/restore register 0—On critical interrupts, stores either the address of the instruction causing the exception or the address of the instruction that executes after the **rfci** instruction.
CSRR1	Critical save/restore register 1—Saves machine state on critical interrupts and restores machine state after an **rfci** instruction is executed.
Debug Interrupt Registers	
DSRR0	Debug save/restore register 0—On debug interrupts, stores either the address of the instruction causing the exception or the address of the instruction that executes after the **rfdi** instruction.
DSRR1	Debug save/restore register 1—Saves machine state on debug interrupts and restores machine state after 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.
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.

Table 1-1. Interrupt Registers (Continued)

Register	Description
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. See Table 1-2 for a list of the associated IVORs.
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 1-2 and Chapter 5, “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

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 1-2 lists IVOR registers implemented on the e200z446n3 and the associated interrupts.

Table 1-2. Exceptions and Conditions

IVOR*n*	Interrupt Type	IVOR*n*	Interrupt Type
None[1]	System reset (not an interrupt)	9	AP unavailable (not used by this core)
0[2]	Critical input	10	Decrementer
1	Machine check	11	Fixed-interval timer
	Machine check (non-maskable interrupt)	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

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

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

1.4 Microarchitecture Summary

The e200z4 processor utilizes a five-stage pipeline for instruction execution. These stages operate in an overlapped fashion, allowing single clock-cycle instruction execution for most instructions. The stages are as follows:

1. Instruction fetch
2. Instruction decode/register file read/effective address calculation
3. Execute 0/memory access 0
4. Execute 1/memory access 1
5. Register write-back

The integer execution units consist 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, and result feed-forward hardware. Integer unit 1 also supports hardware division.

Most arithmetic and logical operations are executed in a single cycle with the exception of multiply, which is implemented with a 2-cycle 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 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 using the BTB is performed to accelerate taken branches. Prefetched instructions are placed into an 8-entry instruction buffer, with each entry capable of holding a single 32-bit instruction or a pair of 16-bit instructions.

Branch target addresses are calculated in parallel with branch instruction decode. Conditional branches that are not taken execute in a single clock cycle. Branches with successful BTB target prefetching have an effective execution time of one clock cycle if correctly predicted. All other taken branches have an execution time of two clock cycles.

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 contains a dedicated effective address adder to allow effective address generation to be optimized. There is a single load-to-use bubble for load instructions.

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

Vectored and autovectored 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 or SPE2 category supports vector instructions operating on 8-, 16-, and 32-bit fixed-point data types, as well as 32-bit IEEE Std. 754™ single-precision floating-point formats. It supports single-precision floating-point operations in a pipelined fashion.

The 64-bit general-purpose register file is used for source and destination operands, and there is a unified storage model for single-precision floating-point data types of 32-bits and the normal integer type. Low latency fixed-point and floating-point add, subtract, multiply, multiply-add, multiply-sub, divide, compare, and conversion operations are provided. Most operations can be pipelined.

Chapter 2
Register Model

This section describes the registers implemented in the e200z4 core. It includes an overview of registers defined by the Power ISA embedded category architecture and highlights differences in how these registers are implemented in the e200z4 core. This section also provides a detailed description of e200-specific registers. Full descriptions of the architecture-defined register set are provided in the *EREF*.

The 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.

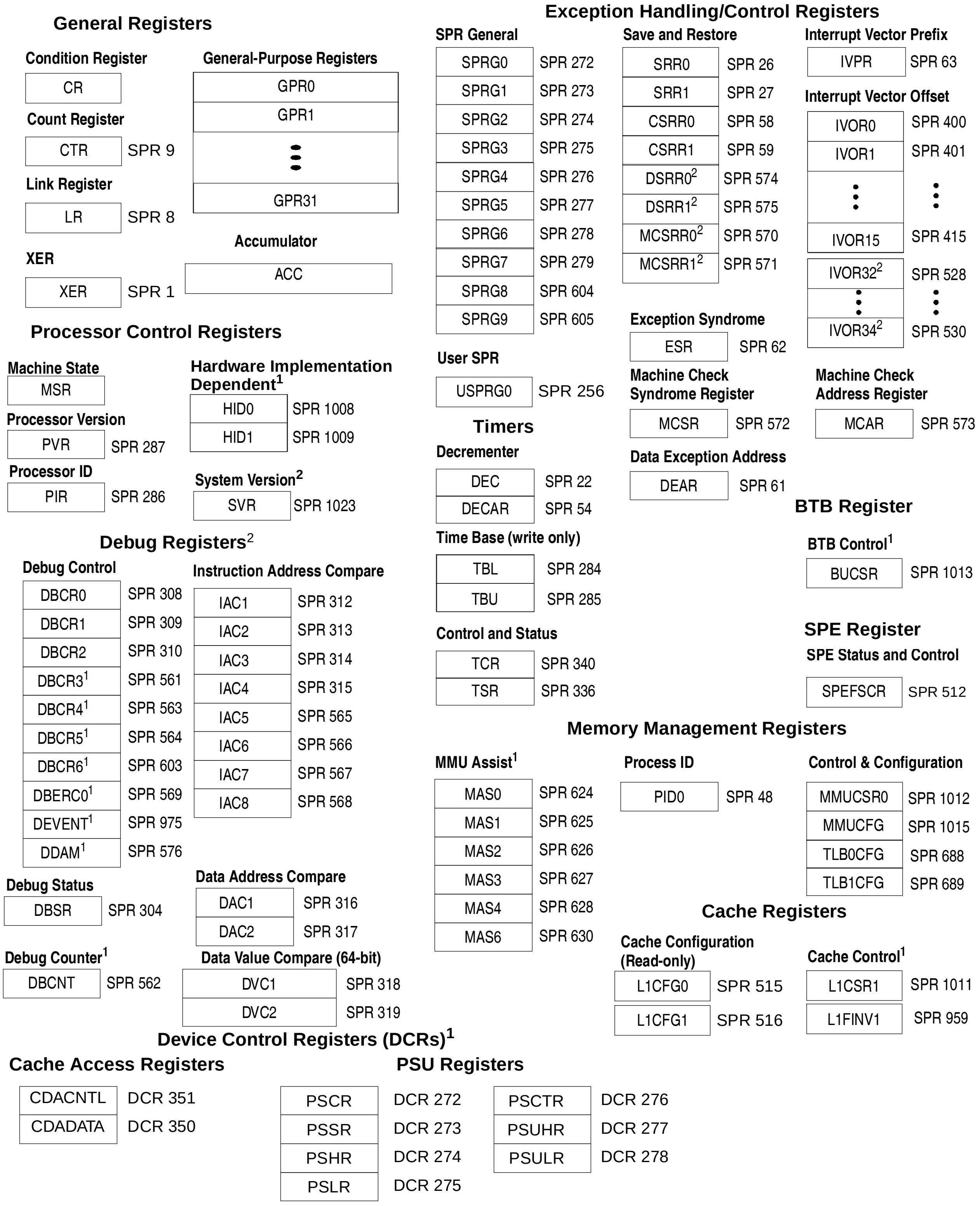
The e200z446n3 extends the general-purpose registers to 64-bits for supporting SPE operations. Power ISA embedded category instructions operate on the lower 32 bits of the GPRs only, and the upper 32 bits are unaffected by these instructions. SPE vector instructions operate on the entire 64-bit register. The SPE defines load and store instructions for transferring 64-bit values to/from memory.

The following figures show the complete e200 register set including the sets of the registers that are accessible while in supervisor mode, and the set of registers that are accessible while in user mode. 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.

NOTE

> The e200z4 is a 32-bit implementation of the Power ISA embedded category. In this document, register bits are sometimes numbered from bit 0 (most significant bit) to 31 (least significant bit), rather than the Book E numbering scheme of 32–63; thus register bit numbers for some registers in Book E are 32 higher.
>
> Where appropriate, the Book E defined bit numbers are shown in parentheses.

Figure 2-1 shows the supervisor-mode special-purpose registers.



General Registers

Condition Register

Register	SPR
CR	

Count Register

Register	SPR
CTR	SPR 9

Link Register

Register	SPR
LR	SPR 8

XER

Register	SPR
XER	SPR 1

General-Purpose Registers

Register
GPR0
GPR1
⋮
GPR31

Accumulator

Register
ACC

Processor Control Registers

Machine State

Register	SPR
MSR	

Processor Version

Register	SPR
PVR	SPR 287

Processor ID

Register	SPR
PIR	SPR 286

Hardware Implementation Dependent[1]

Register	SPR
HID0	SPR 1008
HID1	SPR 1009

System Version[2]

Register	SPR
SVR	SPR 1023

Debug Registers[2]

Debug Control

Register	SPR
DBCR0	SPR 308
DBCR1	SPR 309
DBCR2	SPR 310
DBCR3[1]	SPR 561
DBCR4[1]	SPR 563
DBCR5[1]	SPR 564
DBCR6[1]	SPR 603
DBERC0[1]	SPR 569
DEVENT[1]	SPR 975
DDAM[1]	SPR 576

Instruction Address Compare

Register	SPR
IAC1	SPR 312
IAC2	SPR 313
IAC3	SPR 314
IAC4	SPR 315
IAC5	SPR 565
IAC6	SPR 566
IAC7	SPR 567
IAC8	SPR 568

Debug Status

Register	SPR
DBSR	SPR 304

Data Address Compare

Register	SPR
DAC1	SPR 316
DAC2	SPR 317

Debug Counter[1]

Register	SPR
DBCNT	SPR 562

Data Value Compare (64-bit)

Register	SPR
DVC1	SPR 318
DVC2	SPR 319

Exception Handling/Control Registers

SPR General

Register	SPR
SPRG0	SPR 272
SPRG1	SPR 273
SPRG2	SPR 274
SPRG3	SPR 275
SPRG4	SPR 276
SPRG5	SPR 277
SPRG6	SPR 278
SPRG7	SPR 279
SPRG8	SPR 604
SPRG9	SPR 605

Save and Restore

Register	SPR
SRR0	SPR 26
SRR1	SPR 27
CSRR0	SPR 58
CSRR1	SPR 59
DSRR0[2]	SPR 574
DSRR1[2]	SPR 575
MCSRR0[2]	SPR 570
MCSRR1[2]	SPR 571

Interrupt Vector Prefix

Register	SPR
IVPR	SPR 63

Interrupt Vector Offset

Register	SPR
IVOR0	SPR 400
IVOR1	SPR 401
⋮	⋮
IVOR15	SPR 415
IVOR32[2]	SPR 528
⋮	⋮
IVOR34[2]	SPR 530

User SPR

Register	SPR
USPRG0	SPR 256

Exception Syndrome

Register	SPR
ESR	SPR 62

Machine Check Syndrome Register

Register	SPR
MCSR	SPR 572

Machine Check Address Register

Register	SPR
MCAR	SPR 573

Data Exception Address

Register	SPR
DEAR	SPR 61

Timers

Decrementer

Register	SPR
DEC	SPR 22
DECAR	SPR 54

Time Base (write only)

Register	SPR
TBL	SPR 284
TBU	SPR 285

Control and Status

Register	SPR
TCR	SPR 340
TSR	SPR 336

BTB Register

BTB Control[1]

Register	SPR
BUCSR	SPR 1013

SPE Register

SPE Status and Control

Register	SPR
SPEFSCR	SPR 512

Memory Management Registers

MMU Assist[1]

Register	SPR
MAS0	SPR 624
MAS1	SPR 625
MAS2	SPR 626
MAS3	SPR 627
MAS4	SPR 628
MAS6	SPR 630

Process ID

Register	SPR
PID0	SPR 48

Control & Configuration

Register	SPR
MMUCSR0	SPR 1012
MMUCFG	SPR 1015
TLB0CFG	SPR 688
TLB1CFG	SPR 689

Cache Registers

Cache Configuration (Read-only)

Register	SPR
L1CFG0	SPR 515
L1CFG1	SPR 516

Cache Control[1]

Register	SPR
L1CSR1	SPR 1011
L1FINV1	SPR 959

Device Control Registers (DCRs)[1]

Cache Access Registers

Register	DCR
CDACNTL	DCR 351
CDADATA	DCR 350

PSU Registers

Register	DCR
PSCR	DCR 272
PSSR	DCR 273
PSHR	DCR 274
PSLR	DCR 275
PSCTR	DCR 276
PSUHR	DCR 277
PSULR	DCR 278

1 - These e200-specific registers may not be supported by other processors built on Power Architecture technology
2 - Optional registers defined by the Power ISA embedded architecture
3 - Read-only registers

Figure 2-1. e200z446n3 Supervisor Mode Programmer's Model SPRs

Figure 2-2 shows the user-mode special-purpose registers.

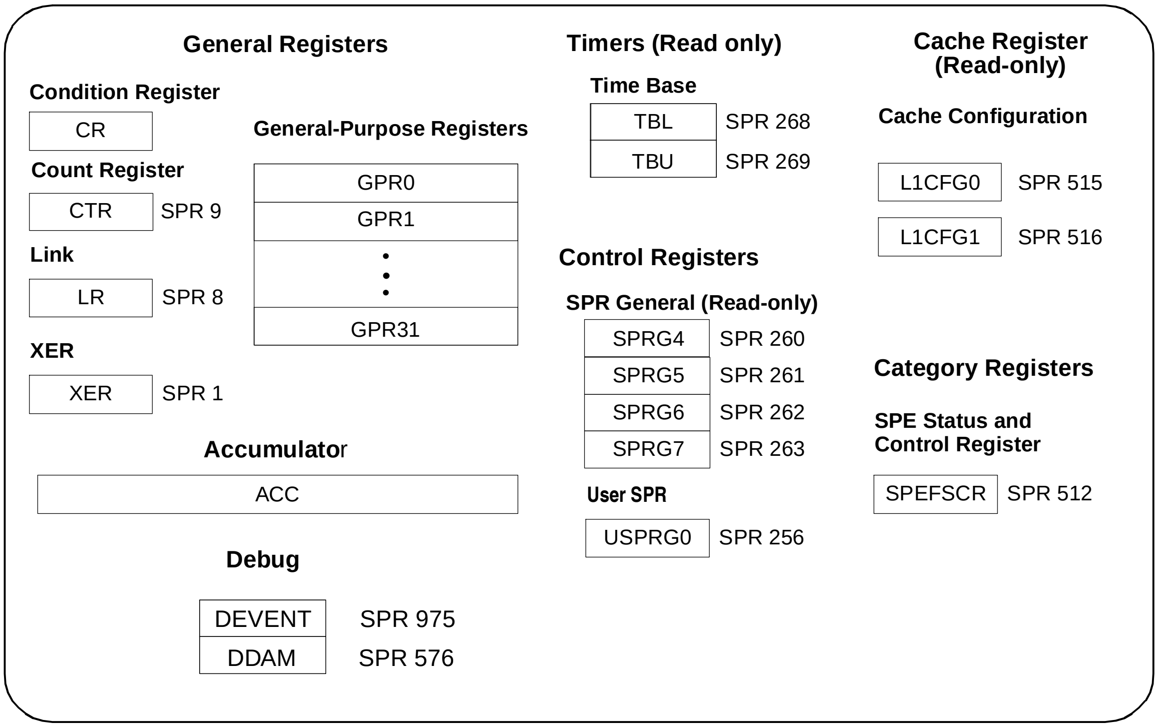


Figure 2-2. e200z446n3 User-Mode Programmer's Model SPRs

General-purpose registers (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 core supports most of the registers defined by Power ISA embedded category architecture. Notable exceptions are the floating-point registers FPR0–FPR31 and FPSCR. The e200z4 does not support the Power ISA floating-point architecture in hardware. The general-purpose registers have been extended to 64-bits. e200-specific registers are described in Section 2.2, "e200-Specific Special Purpose Registers," and the Power ISA embedded registers are described in the following sections. For complete descriptions, see the *EREF*.

2.1.1 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 EFP instructions are provided that 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. See "Condition Register (CR)," in Chapter 3, "Branch and Condition Register Operations of the *EREF*.

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. See "XER Register (XER)," in Chapter 4, "Integer Operations" of the *EREF* for more information.
- Link register (LR).
  - — The LR provides the branch target address for the branch [conditional] to link register instructions (**bclr, bclrl, se_blr, se_blrl**). It holds the address of the instruction that follows a branch and link instruction, typically used for linking to subroutines. See "Link Register (LR)", in Chapter 3, "Branch and Condition Register Operations" of the *EREF*.
- 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 instructions (**bcctr, bcctrl, se_bctr, se_bctrl**). See "Count Register (CTR)", in Chapter 3, "Branch and Condition Register Operations" of the *EREF*.
- Time base upper (TBU) and time base lower (TBL)
  - — The time base facility (TB) consists of two 32-bit registers. These two registers are accessible in a read-only fashion to user-level software. See "Time Base", in Chapter 8, "Timer Facilities" of the *EREF*.
- SPRG4—SPRG7
  - — The Power ISA embedded category architecture defines software-use special purpose registers (SPRGs). SPRG4 through SPRG7 are accessible in a read-only fashion by user-level software. 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, which is accessible in a read-write fashion by user-level software.

2.1.2 Supervisor-level Registers

Supervisor-level software has access to additional control and status registers used for configuration, exception handling, and other operating system functions in addition to the registers accessible in user-mode. 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 instruction (**mtmsr**), system call instructions (**sc, se_sc)**, and return from exception instructions (**rfi, rfci, rfdi, rfmci, se_rfi, se_rfci, se_rfdi, se_rfmci)**. It can be read by the move from machine state register instruction (**mfmsr)** . 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 modules for ownership trace message generation. The Power ISA allows multiple PIDs; the e200z4 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 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-IVOR34)

    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 non-critical interrupt. It contains the address of the instruction at which execution resumes when an **rfi** or **se_rfi** instruction is executed at the end of a non-critical class interrupt handler routine.

  - Critical save/restore register 0 (CSRR0)

    The CSRR0 register is used to save machine state on a critical interrupt. It 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 non-critical 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 (DAC1–2)

    These registers contain addresses and/or masks which are used to specify data address compare debug events.
  - Data value compare registers (DVC1–2)

    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: TBU and TBL. The time base registers can be written to 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 explained in the following sections.

2.2.1 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 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 the IEEE Std. 754 standard. See “SPE Status and Control Register (SPEFSCR),” in Chapter 7, “Signal Processing Extension Unit.”
- The L1 cache configuration registers (L1CFG0, L1CGF1)

  These read-only registers allows software to query the configuration of the L1 cache structure.

2.2.2 Supervisor-Level Registers

In addition to the Power ISA embedded category registers described above, the following supervisor-level registers are defined in the e200:

- 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, 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 register 3–6 (DBCR3, DBCR4, DBCR5, 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 PowerPC Book E architecture.
  - Debug counter register (DBCNT)

    This register provides counter capability for debug functions.
  - Branch unit control and status register (BUCSR)

    This register controls operation of the BTB
- Cache Registers
  - L1 cache configuration registers (L1CFG0, L1CFG1)

    These are read-only registers that allow software to query the configuration of the L1 Cache.
  - L1 cache control and status registers (L1CSR0, L1CSR1)

    These registers control operations of the L1 Cache, such as cache enabling, cache invalidation, and cache locking.
  - L1 cache flush and invalidate register (L1FINV1)

    This register controls software flushing and invalidation of the L1 Caches.
- Memory management unit registers
  - MMU configuration register (MMUCFG)

    This 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 Memory Management Unit.
  - MMU control and status register (MMUCSR0)

    This register controls invalidation of the MMU.
  - TLB configuration registers (TLB0CFG, TLB1CFG)

    These 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 that includes the e200 processor.

Note that 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 core 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 11.9, “Parallel Signature Unit.”

2.4 Special Purpose Register Descriptions

The following sections provide a register figure and accompanying field descriptions table for each of the SPRs in the core.

2.4.1 Machine State Register (MSR)

The machine state register defines the state of the processor. Chapter 5, “Interrupts and Exceptions,” of this document describes how interrupts affect the MSR, and the *EREF* contains a complete description.

Figure 2-3 shows the e200 MSR.

0					UCLE	SPE	0						WE	CE	0	EE	PR	FP	ME	FE0	0	DE	FE1	0		IS	DS	0		RI	0
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

Read/ Write; Reset - 0x0

Figure 2-3. Machine State Register (MSR)

Table 2-1 defines the MSR bits.

Table 2-1. MSR Field Descriptions

Bit(s)	Name	Description
0–4 (32–36)	—	Reserved[1]
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 Available 0 Execution of SPE APU vector instructions is disabled; SPE Unavailable exception taken instead, and SPE bit is set in ESR. 1 Execution of SPE APU vector instructions is enabled.

Table 2-1. MSR Field Descriptions (Continued)

Bit(s)	Name	Description
7–12 (39–44)	—	Reserved[1]
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)	—	Preserved[1]
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 (e.g. 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. (A 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:** For the e200, the floating point unit is not supported in hardware, and an unimplemented operation exception will be 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 the e200)
21 (53)	—	Reserved[1]
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)
24 (56)	—	Reserved[1]
25 (57)	—	Preserved[1]

Table 2-1. MSR Field Descriptions (Continued)

Bit(s)	Name	Description
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).
28–29 (60–61)	—	Reserved[1]
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)	—	Preserved[1]

[1] These bits are not implemented, will be read as zero, and writes are ignored.

2.4.2 Processor ID Register (PIR)

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

ID
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

SPR - 286; Read/Write; Reset: - bits 24:31 updated to reflect the values on **p_cpuid[0–7]**, bits 0–23 reset to 0

Figure 2-4. 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. These bits are writable by software.
24–31		These bit are reset to the values provided on the **p_cpuid[0–7]** input signals. These bits are writable by software.

2.4.3 Processor Version Register (PVR)

The processor version register (PVR) contains the processor version number for the CPU core.

1	0	0	0	0	0	0	1	0	1	0	1	Version				MBG Reserved				Minor Rev				Major Rev				MBG ID			
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

SPR - 287; Read-only

Figure 2-5. Processor Version Register (PVR)

This register contains fields to specify a particular implementation of an e200 family member as well as allocating fields to be used by a particular business unit at their discretion. This register is read-only. Interface signals **p_pvrin[16–31]** provide the contents of a portion of this register.

Table 2-3 shows the PVR field descriptions.

Table 2-3. PVR Field Descriptions

Bits	Name	Description
0–3	Manuf. ID	These bits identify the Manufacturer ID. Freescale is 4`b1000.
4–5	—	These bits are reserved (00)
6–11	Type	These bits identify the processor type. e200z4 is 6`b010101.
12–15	Version	These bits identify the version of the processor and inclusion of optional elements. For the e200z446n3, these are tied to 4`b0101.
16–19	MBG Use	These bits are allocated for use by Freescale Business Groups 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 Business Group responsible for a particular mask set, and are provided by the **p_pvrin[28–31]** input signals. MBG value of 4`b0000 is reserved.

2.4.4 System Version Register (SVR)

The system version register (SVR) contains system version information for an e200-based SoC.

System Version																															
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

SPR - 1023; Read-only

Figure 2-6. System Version Register (SVR)

This register is used to specify a particular implementation of an e200-based system by a particular business unit at their discretion. This register is read-only.

Table 2-4 shows the SVR field descriptions.

Table 2-4. SVR Field Descriptions

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

2.4.5 Integer Exception Register (XER)

The *EREF* contains a complete description of the integer exception register (XER).

The XER bit assignments are shown in Figure 2-7.

SO	OV	CA	0																						Bytecnt						
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

SPR - 1; Read/Write; Reset - 0x0

Figure 2-7. 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 the Power ISA embedded category)
1 (33)	OV	Overflow (per the Power ISA embedded category)
2 (34)	CA	Carry (per the Power ISA embedded category)
3–24 (35–56)	—	Reserved[1]
25–31 (57–63)	Bytecnt[2]	Preserved for **lswi**, **lswx**, **stswi**, **stswx** string instructions

[1] These bits are not implemented, will be read as zero, and writes are ignored.

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

2.4.6 Exception Syndrome Register

The exception syndrome register (ESR) provides a syndrome to differentiate between exceptions that can generate the same interrupt type. The *EREF* contains a complete description of the ESR. The e200 adds some implementation specific bits to this register, as seen in Figure 2-8.

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				PIL	PPR	PTR	FP	ST	0	DLK	ILK	AP	PUO	BO	PIE	0								SPE	0	VLEMI	0			MIF	0

SPR - 62; Read/Write; Reset - 0x0

Figure 2-8. 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)	—	Allocated[1]	—
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[2]	—
10 (42)	DLK	Data Cache Locking[3]	Data Storage
11 (43)	ILK	Instruction Cache Locking	Data Storage
12 (44)	AP	Auxiliary Processor operation (Currently unused in the e200)	Alignment Data Storage Data TLB Program
13 (45)	PUO	Unimplemented Operation exception	Program

Table 2-6. ESR Field Descriptions (Continued)

Bit(s)	Name	Description	Associated Interrupt Type
14 (46)	BO	Byte Ordering exception Mismatched Instruction Storage exception	Data Storage Instruction Storage
15 (47)	PIE	Program Imprecise exception (Reserved)	Currently unused in the e200

Table 2-6. ESR Field Descriptions (Continued)

Bit(s)	Name	Description	Associated Interrupt Type
16–23 (48–55)	—	Reserved[2]	—
24 (56)	SPE	SPE Operation	SPE Unavailable SPE Floating-point Data Exception SPE Floating-point Round Exception Alignment Data Storage Data TLB
25 (57)	—	Allocated[1]	—
26 (58)	VLEMI	VLE Mode Instruction	SPE Unavailable SPE Floating-point Data Exception SPE Floating-point Round Exception Data Storage Data TLB Instruction Storage Alignment Program System Call
27:29 (59–61)	—	Allocated[1]	—
30 (62)	MIF	Misaligned Instruction Fetch	Instruction Storage Instruction TLB
31 (63)	—	Allocated[1]	—

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

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

[3] This bit is implemented, but not set by hardware

2.4.6.1 Power ISA VLE Mode Instruction Syndrome

The ESR[VLEMI] bit 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] bit is provided to indicate that an Instruction Storage Interrupt was caused by an attempt to fetch an instruction from a Power ISA page that 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 that is not marked as a Power ISA VLE page.

The ESR[MIF] bit 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 points to the first half of the instruction, which resides 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. The MCSR is shown in Figure 2-9.

MCP	IC_DPERR	0		EXCP_ERR	IC_TPERR	0	IC_LKERR	0			NMI	MAV	MEA	0	IF	LD	ST	G	0								BUS_IRERR	BUS_DRERR	BUS_WRERR	0	
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

SPR - 572; Read/Clear; Reset - 0x0

Figure 2-9. 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 1 to clear.” Therefore, software must write ones into those bit positions it wishes to clear, typically by writing back what was originally read. See Section 5.7.2, “Machine Check Interrupt (IVOR1) for more details of the MCSR settings.

Table 2-7. 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
2–3 (34–35)	—	Reserved, should be cleared.	—	—
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

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

Bit	Name	Description	Exception Type[1]	Recoverable
6 (38)	—	Reserved, should be cleared.	—	—
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	—
8–10 (40–42)	—	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. 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 Load or Store 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 guarded access encountered an error on the external bus.	Error Report	Precise
19–26 (51–58)	—	Reserved, should be cleared.	—	—
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	Async Mchk	Precise

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

Bit	Name	Description	Exception Type[1]	Recoverable
29 (61)	BUS_WRERR	Write bus error on store	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

- “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 will 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 an asynchronous machine check exception. These bits will 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 will remain 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 will 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. The *EREF* contains a complete description of the TCR. 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, TCR[WPEXT] and TCR[FPEXT].

The TCR is shown in Figure 2-10.

WP		WRC		WIE	DIE	FP		FIE	ARE	0	WPEXT				FPEXT				0												
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

SPR - 340; Read/Write; Reset - 0x0

Figure 2-10. 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 one of 64 bit locations of the time base used to signal a watchdog timer exception on a transition from 0 to 1. TCRwpext[0–3],TCRwp[0–1] == 6’b000000 selects TBU[0] TCRwpext[0–3],TCRwp[0–1] == 6’b111111 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 time-out of Watchdog Timer 10 Assert watchdog reset status output 0 (**p_wrs[0]**) on second time-out of Watchdog Timer 11 Assert watchdog reset status outputs 0 and 1 (**p_wrs[0], p_wrs[1]**) on second time-out 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 one of 64 bit locations of the time base used to signal a fixed-interval timer exception on a transition from 0 to 1. TCR_{fpext}[0–3],TCR_{fp}[0–1] == 6’b000000 selects TBU[0] TCR_{fpext}[0–3],TCR_{fp}[0–1] == 6’b111111 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 the one of the 64 Time Base bits used to signal a Watchdog Timer exception. $tb_{0:63} \leftarrow TBU_{0:31} \parallel TBL_{0:31}$ $wp \leftarrow TCR_{WPEXT} \parallel TCR_{WP}$ $tb\_wp\_bit \leftarrow tb_{wp}$
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 the one of the 64 Time Base bits used to signal a Fixed-Interval Timer exception. $tb_{0:63} \leftarrow TBU_{0:31} \parallel TBL_{0:31}$ $fp \leftarrow TCR_{FPEXT} \parallel TCR_{FP}$ $tb\_fp\_bit \leftarrow tb_{fp}$
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 is in the *EREF*. TSR[WRS] is defined to be implementation-dependent and is described below.

The TSR is shown in Figure 2-11.

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
ENW	WIS	WRS		DIS	FIS	0																									

SPR - 336; Read/Clear; Reset - 0x0

Figure 2-11. 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 time-out of Watchdog Timer has occurred 10 Assertion of watchdog reset status output 0 (**p_wrs[0]**) on second time-out of Watchdog Timer has occurred 11 Assertion of watchdog reset status outputs 0 and 1 (**p_wrs[0], p_wrs[1]**) on second time-out of Watchdog Timer has occurred
4 (36)	DIS	Decrementer interrupt status
5 (37)	FIS	Fixed-Interval Timer interrupt status
6–31 (38–63)	—	Reserved[1]

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

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 bits in GPR[RS] can be cleared using **mtspr TSR,RS**.

2.4.10 Debug Registers

The debug facility registers are described in Chapter 11, “Debug Support.”

2.4.11 Hardware Implementation Dependent Register 0 (HID0)

The HID0 register is an e200 implementation-dependent register used for various configuration and control functions, as shown in Figure 2-12.

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
EMCP	0							DOZE	NAP	SLEEP	0			ICR	NHR	0	TBEN	SEL_TBCLK	DCLREE	DCLRCE	CICLRDE	MCCLRDE	DAPUEN	0							NOPTI

SPR - 1008; Read/Write; Reset - 0x0

Figure 2-12. 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. The primary purpose of this bit is to mask out further machine check exceptions caused by assertion of **p_mcp_b**.
1–7 [33–39]	—	Reserved[1]
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[1]

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

Bits	Name	Description
14 [46]	ICR	Interrupt Inputs Clear Reservation 0 External Input, Critical Input, and Non-Maskable Interrupts do not affect reservation status 1 External Input, Critical Input, and Non-Maskable 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.
16 [48]	—	Reserved[1]
17 [49]	TBEN	TimeBase Enable 0 TimeBase is disabled 1 TimeBase is enabled
18 [50]	SEL_TBCLK	Select TimeBase Clock 0 TimeBase is based on processor clock 1 TimeBase is based on **p_tbclk** input This bit controls the clock source for the TimeBase. 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 TimeBase clock source.
19 [51]	DCLREE	Debug Interrupt Clears $\mathrm{MSR}_{\mathrm{EE}}$ 0 $\mathrm{MSR}_{\mathrm{EE}}$unaffected by Debug Interrupt 1 $\mathrm{MSR}_{\mathrm{EE}}$ 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 $\mathrm{MSR}_{\mathrm{CE}}$ 0 $\mathrm{MSR}_{\mathrm{CE}}$ unaffected by Debug Interrupt 1 $\mathrm{MSR}_{\mathrm{CE}}$ 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 $\mathrm{MSR}_{\mathrm{DE}}$ 0 $\mathrm{MSR}_{\mathrm{DE}}$ unaffected by Critical class interrupt 1 $\mathrm{MSR}_{\mathrm{DE}}$ 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:** If Critical Interrupt Debug events are enabled ($\mathrm{DBCR0}_{\mathrm{CIRPT}}$ set, which should only be done when the Debug functionality is enabled), and $\mathrm{MSR}_{\mathrm{DE}}$ 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, $\mathrm{DSRR0}_{\mathrm{DE}}$ will have been cleared, such that after returning from the debug handler, the Critical interrupt handler will not be run with $\mathrm{MSR}_{\mathrm{DE}}$ enabled.

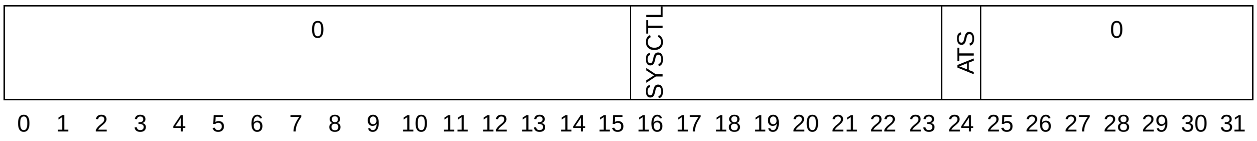
Table 2-10. Hardware Implementation Dependent Register 0 (Continued)

Bits	Name	Description
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.
23 [55]	DAPUEN	Debug APU enable 0 Debug APU disabled 1 Debug APU enabled This bit controls whether the Debug APU 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 ‘1’s Read and write access to DSRR0/DSRR1 by means of the **mfspr** and **mtspr** instructions is not affected by this bit.
24 [56]	—	Reserved[1]
25–30 [58–62]	—	Reserved[1]
31 [63]	NOPTI	No-op Touch Instructions 0 **icbt** instruction operates normally 1 **icbt** instruction is no-oped This bit only affects the **icbt** instruction.

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

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 2-13.



SPR - 1009; Read/Write; Reset - 0x0

Figure 2-13. Hardware Implementation Dependent Register 1 (HID1)

The HID1 fields are defined in Table 2-11.

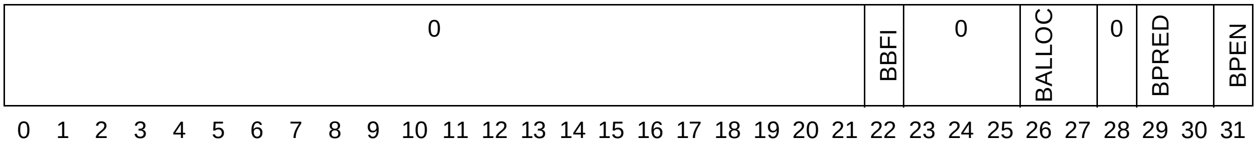
Bits	Name	Description
0–15 [32–47]	—	Reserved[1]
16–23 [48–56]	SYSCTL	System Control These bits are reflected on the outputs of the p_hid1_sysctl[0–7] output signals for use in controlling the system. They may need external synchronization.
24 [56]	ATS	Atomic status (read-only) Indicates state of the reservation bit in the load/store unit. See Section 3.6, “Memory Synchronization and Reservation Instructions for more detail.
25–31 [57–63]	—	Reserved[1]

Table 2-11. Hardware Implementation Dependent Register 1

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

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), as shown in Figure 2-14.



SPR - 1013; Read/Write; Reset - 0x0

Figure 2-14. Branch Unit Control and Status Register (BUCSR)

The BUCSR fields are defined in Table 2-12.

Bits	Name	Description
0–21 [32–53]	—	Reserved[1]
22 [54]	BBFI	Branch target buffer flash invalidate. When written to a ‘1’, 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 0.
23–25 [55–57]	—	Reserved[1]
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[1]
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.

Table 2-12. Branch Unit Control and Status Register

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

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 cache. 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 Register (L1FINV1)

The L1FINV0 and L1FINV1 registers provide software-based flush and invalidation control for the L1 instruction cache supplied with this version of the e200 CPU core. A description of the L1FINV1 register 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. 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 lists the system response to an invalid SPR.

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 category. The *EREF* contains a complete description of synchronization requirements.

Software requirements for synchronization before or 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	
mfspr				
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	
L1FINV1	L1 cache flush and invalidate control register 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	Debug Control Register 0	none	CSI	
DBCR1	Debug Control Register 1	none	CSI	
DBCR2	Debug Control Register 2	none	CSI	
DBCR3	Debug control register 3	none	CSI	
DBCR4	Debug control register 4	none	CSI	

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

Context Altering Event or Instruction		Required Before	Required After	Notes
DBCR5	Debug control register 5	none	CSI	
DBCR6	Debug control register 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	
L1FINV1	L1 cache flush and invalidate control register 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

Table 2-15. Special Purpose Registers (Continued)

Mnemonic	Name	SPR Number	Access	Privileged	e200 Specific
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
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

Table 2-15. Special Purpose Registers (Continued)

Mnemonic	Name	SPR Number	Access	Privileged	e200 Specific
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
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
L1FINV1	L1 cache flush and invalidate control register 0	959	R/W	Yes	Yes
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

Table 2-15. Special Purpose Registers (Continued)

Mnemonic	Name	SPR Number	Access	Privileged	e200 Specific
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 APU 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
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

Table 2-15. Special Purpose Registers (Continued)

Mnemonic	Name	SPR Number	Access	Privileged	e200 Specific
USPRG0	User SPR General 0	256	R/W	No	No
XER	Integer Exception Register	1	R/W	No	No

Notes:

[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 architected 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] || 2’b00
GPRs	Unaffected[1]
CR	Unaffected[1]
BUCSR	0x0000_0000
CSRR0	Unaffected[1]
CSRR1	Unaffected[1]
CTR	Unaffected[1]
DAC1	0x0000_0000[2]
DAC2	0x0000_0000[2]
DBCNT	Unaffected[1]
DBCR0	0x0000_0000[2]
DBCR1	0x0000_0000[2]
DBCR2	0x0000_0000[2]
DBCR3	0x0000_0000[2]
DBCR4	0x0000_0000[2]
DBCR5	0x0000_0000[2]
DBCR6	0x0000_0000[2]

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

Resource	system reset setting
DBSR	0x1000_0000[2]
DDAM	0x0000_0000[2]
DEAR	Unaffected[1]
DEC	Unaffected[1]
DECAR	Unaffected[1]
DEVENT	0x0000_0002
DSRR0	Unaffected[1]
DSRR1	Unaffected[1]
DVC1	Unaffected[1]
DVC2	Unaffected[1]
ESR	0x0000_0000
HID0	0x0000_0000
HID1	0x0000_0000
IAC1	0x0000_0000[2]
IAC2	0x0000_0000[2]
IAC3	0x0000_0000[2]
IAC4	0x0000_0000[2]
IAC5	0x0000_0000[2]
IAC6	0x0000_0000[2]
IAC7	0x0000_0000[2]
IAC8	0x0000_0000[2]
IVOR0	Unaffected[1]
IVOR1	Unaffected[1]
IVOR2	Unaffected[1]
IVOR3	Unaffected[1]
IVOR4	Unaffected[1]
IVOR5	Unaffected[1]
IVOR6	Unaffected[1]
IVOR7	Unaffected[1]
IVOR8	Unaffected[1]
IVOR9	Unaffected[1]
IVOR10	Unaffected[1]
IVOR11	Unaffected[1]

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

Resource	system reset setting
IVOR12	Unaffected[1]
IVOR13	Unaffected[1]
IVOR14	Unaffected[1]
IVOR15	Unaffected[1]
IVPR	Unaffected[1]
LR	Unaffected[1]
L1CFG0, L1CFG1[3]	—
L1CSR0, 1	0x0000_0000
L1FINV1	0x0000_0000
MAS0	Unaffected[1]
MAS1	Unaffected[1]
MAS2	Unaffected[1]
MAS3	Unaffected[1]
MAS4	Unaffected[1]
MAS6	Unaffected[1]
MCAR	Unaffected[1]
MCSR	0x0000_0000
MCSRR0	Unaffected[1]
MCSRR1	Unaffected[1]
MMUCFG[3]	—
MSR	0x0000_0000
PID0	0x0000_0000
PIR	0x0000_00 || p_cpuid[0–7]
PVR[3]	—
SPEFSCR	0x0000_0000
SPRG0	Unaffected[1]
SPRG1	Unaffected[1]
SPRG2	Unaffected[1]
SPRG3	Unaffected[1]
SPRG4	Unaffected[1]
SPRG5	Unaffected[1]
SPRG6	Unaffected[1]
SPRG7	Unaffected[1]

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

Resource	system reset setting
SPRG8	Unaffected[1]
SPRG9	Unaffected[1]
SRR0	Unaffected[1]
SRR1	Unaffected[1]
SVR[3]	—
TBL	Unaffected[1]
TBU	Unaffected[1]
TCR	0x0000_0000
TSR	0x0000_0000
TLB0CFG[3]	—
TLB1CFG[3]	—
USPRG0	Unaffected[1]
XER	0x0000_0000

[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 Power Architecture technology as it relates specifically to the e200z4.

The e200z4 is a 32-bit implementation of Power Architecture technology as defined in the Power ISA. This architecture specification includes a recognition that different processor implementations may require clarifications, extensions, or deviations from the architectural descriptions.

3.1 Unsupported Instructions and Instruction Forms

Because the e200z4 is a 32-bit Power ISA embedded 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.

The e200z4 core does not support the instructions listed in Table 3-1. An unimplemented instruction or FP-unavailable exception is generated if the processor attempts to execute one of these instructions.

Table 3-1. List of Unsupported Instructions

Type/Name	Mnemonics
String Instructions	**lswi, lswx, stswi, stswx**
Floating Point Instructions	**fxxxx, lfxxx, sfxxxx, mcrfs, mffs, mtfxxx**
Device control register and Move from APID	**mfapidi**, **mfdcrx, mtdcrx**

3.2 Optionally Supported Instructions and Instruction Forms

e200 cores optionally support the instructions listed in Table 3-2 if a cache and/or TLB is present. An instruction exception may be generated if the processor attempts to execute one of these instructions and the related functional block is not present. The specific instruction may also be treated as a no-op.

Table 3-2. List of Optionally Supported Instructions

Type / Name	Mnemonics	Unit
Cache Management Instructions[1]	**dcba, dcbf, dcbi, dcbt, dcbtst, dcbst, dcbz**	Data Cache
Cache Management Instructions[2]	**icbi, icbt**	Instruction Cache
Cache Locking Instructions[3]	**dcbtls, dcbtstls, dcblc**	Data Cache
Cache Locking Instructions[2]	**icbtls, icblc**	Instruction Cache
TLB Management Instructions[2]	**tlbivax, tlbre, tlbsx, tlbsync, tlbwe**	TLB
DCR Management [2]	**mfdcr, mtdcr**	DCR

[1] These instructions are not supported and are treated as no-ops, with the exception of **dcbz** which results in an Alignment Interrupt, and **dcbi**, which is treated as a privileged no-op.

[2] These instructions are supported by e200z446n3

[3] These instructions are not supported and are treated as no-ops.

3.3 Implementation Specific Instructions

Several instructions defined in the Power ISA are implementation specific. Table 3-3 summarizes the e200 implementation-specific instructions.

Table 3-3. Implementation-specific Instruction Summary

Mnemonic	Implementation Details
mfapidi	Unimplemented instructions
mfdcrx	
mtdcrx	

Table 3-3. Implementation-specific Instruction Summary

Mnemonic	Implementation Details
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 takes an illegal instruction exception for unsupported DCR values

3.4 Power ISA Instruction Extensions

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

3.5 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; 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.

3.6 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.

On the e200 core, the **mbar** instruction with MO = 0, 1, or 2 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 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 Freescale EIS enhanced reservation instruction set. 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 for a reservation store-type instruction to succeed, the EA of the store-type instruction must be to the same reservation granule as the EA of a preceding reservation load-type instruction. 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 it 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 non-maskable interrupt is signaled and the HID0[ICR] bit 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, then the store-type reservation instruction is treated as a nop. No exceptions will be taken, and no access is performed, thus no data breakpoint will occur, regardless of matching the data breakpoint attributes.

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 will cause 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 will be 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 multi-tasking applications.

3.7 Branch Prediction

The e200z4 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.

3.8 Interruption of Instructions by Interrupt Requests

In general, the core samples pending non-maskable 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]**), floating square root (**efssqrt**, **evfssqrt**), 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 instructions may also be interrupted prior to completion. When interrupted prior to completion, the value saved in SRR0/CSRR0/MCSRR0 will be the address of the interrupted instruction. The instruction will be restarted from the beginning after returning to it from the interrupt handler.

3.9 New e200z4 Categories

The e200z4 core implements the following Freescale EIS categories that extend the Power ISA:

- The ISEL category, which is described in Section 3.10, "ISEL Instruction."
- The Enhanced Debug category and the Debug Notify Halt Instruction, which are described in Section 3.11, "Enhanced Debug."
- The Machine Check category ,which is described in Section 3.12, "Machine Check."
- The WAIT category, which is described in Section 3.13, "WAIT Instruction."
- The Volatile Context Save/Restore category, which is described in Section 3.15, "Volatile Context Save/Restore."
- The Embedded Floating-Point category version 2, described along with supporting instructions in Chapter 6, "Embedded Floating-Point Unit, Version 2."
- The Signal Processing Extension (SPE) category version 1.1, described along with supporting instructions in Chapter 7, "Signal Processing Extension Unit."
- The Cache Line-Locking category, which is described in Section 9.10, "Cache Line Locking/Unlocking."
- The Enhanced Reservations category, which is described in Section 3.14, "Enhanced Reservations."

3.10 ISEL Instruction

The **isel** instruction provides a way 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

3.11 Enhanced Debug

The e200z4 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 instruction set, 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]|| 1’b0. 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 functionality 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 either 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[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 functionality is disabled (HID0[DAPUEN] = 0), this instruction is treated as an illegal instruction.

3.11.1 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. **dnh** provides the external debugger with bits reserved in the instruction itself to pass additional information. When the e200z4 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. Prior to exiting the debug halted state, the external debugger should update the CPUSCR to point past the **dnh**, **e_dnh**, or **se_dnh** instruction.

Note that the **dnh** instruction is only available in the 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 the EDBCR[DNH_EN] bit. 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].

Whether the processor is in IDM or EDM mode has no effect on the execution of the **dnh** instruction.

Other registers altered:

- None

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 the EDBCR[DNH_EN] bit. 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; it executes the same regardless of the state of MSR[PR].

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 EDBCRDNH_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 the EDBCR[DNH_EN] bit.

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; it executes the same regardless of the state of MSR[PR].

Whether the processor is in IDM or EDM mode has no effect on the execution of the **se_dnh** instruction.

Other registers altered:

- None

3.12 Machine Check

The e200z4 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 instruction set, 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 either 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]|| 1’b0. 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 (i.e. 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 placed into the machine state register. If the new machine state register value does not enable any pending exceptions, the next instruction is fetched under control of the new machine state register value from the address MCSRR0[0–30]|| 1’b0. 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.

3.13 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**). Non-maskable interrupts (**p_nmi_b**) will 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.

3.14 Enhanced Reservations

The e200 implements the Freescale EIS enhanced reservations functionality, which 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.

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	/
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 0s concatenated with bits 32:63 of the sum of the contents of GPR[RA], or 64 0s 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 0.

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

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 0.

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 two. If it is not, either an alignment interrupt is invoked or the results are boundedly undefined.

Special registers altered:

None

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

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 two. If it is not, either an alignment interrupt is invoked or the results are boundedly undefined.

Special registers altered:

CR0

3.15 Volatile Context Save/Restore

The e200 implements the Power ISA embedded category volatile context save/restore instruction set 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 has been defined as part of the volatile context save/restore instruction set, as shown in Table 3-4.

Table 3-4. Volatile Context Save/Restore Instruction Set

Instruction	Definition
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.

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 contents of GPR[RA], or 0 if RA = 0 and 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 four. If it is not, either an alignment interrupt is invoked or the results are boundedly undefined.

Special registers affected:

None

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 contents of GPR[RA], or 0 if RA = 0 and 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 four. If it is not, either an alignment interrupt is invoked or the results are boundedly undefined.

Special registers affected:

None

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 contents of GPR[RA], or 0 if RA = 0 and 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 four. If it is not, either an alignment interrupt is invoked or the results are boundedly undefined.

Special registers affected:

CR, LR, CTR, XER

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 contents of GPR[RA], or 0 if RA=0 and 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 affected:

None

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 contents of GPR[RA], or 0 if RA = 0 and 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 four. If it is not, either an alignment interrupt is invoked or the results are boundedly undefined.

Special registers affected:

SRR0, SRR1

Store Multiple Volatile SRR Word

e_stmvsrrw D8(RA) (D8-mode)

0 0 0 1 1 0	0 0 1 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) ← $SRR0_{32:63}$
EA ← (EA+4)
MEM(EA,4) ← $SRR1_{32:63}$
```

Let the effective address (EA) be the sum of the contents of GPR[RA], or 0 if RA = 0 and 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 four. If it is not, either an alignment interrupt is invoked or the results are boundedly undefined.

Special registers affected:

None

Load Multiple Volatile CSRR Word

e_lmvcsrrw D8(RA) (D8-mode)

0 0 0 1 1 0	0 0 1 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))

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

Let the effective address (EA) be the sum of the contents of GPR[RA], or 0 if RA = 0 and 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 four. If it is not, either an alignment interrupt is invoked or the results are boundedly undefined.

Special registers affected:

CSRR0, CSRR1

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 contents of GPR[RA], or 0 if RA = 0 and 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 four. If it is not, either an alignment interrupt is invoked or the results are boundedly undefined.

Special registers affected:

None

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 contents of GPR[RA], or 0 if RA = 0 and 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 four. If it is not, either an alignment interrupt is invoked or the results are boundedly undefined.

Special registers affected:

DSRR0, DSRR1

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 contents of GPR[RA], or 0 if RA = 0 and 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 four. If it is not, either an Alignment interrupt is invoked or the results are boundedly undefined.

Special registers affected:

None

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 contents of GPR[RA], or 0 if RA = 0 and 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 four. If it is not, either an alignment interrupt is invoked or the results are boundedly undefined.

Special registers affected:

MCSRR0, MCSRR1

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 contents of GPR(RA), or 0 if RA=0, and 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 four. If it is not, either an alignment interrupt is invoked or the results are boundedly undefined.

Special registers affected:

None

3.16 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.

3.17 Invalid Forms of Instructions

This section discusses the following invalid forms of instructions:

- Section 3.17.1, “Load and Store with Update Instructions”
- Section 3.17.2, “Load Multiple Word (lmw, e_lmw) Instruction”
- Section 3.17.3, “Branch Conditional To Count Register Instructions”
- Section 3.17.4, “Instructions With Reserved Fields Non-Zero”

3.17.1 Load and Store with Update Instructions

Power ISA 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).

3.17.2 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 that existed when beginning execution of this **lmw** or **e_lmw** instruction. RA is 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 is done substituting the value of 0 for the RA operand.

3.17.3 Branch Conditional To Count Register Instructions

The Power ISA embedded category defines as invalid any **bcctr** or **bcctrl** instruction that specifies the decrement and test CTR (BO[2] = 0) option. For these invalid forms of instructions, the core executes 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.

3.17.4 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, the 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.

3.18 Instruction Summary

Table 3-5 and Table 3-6 list all 32-bit instructions in the Power ISA embedded category, as well as certain e200-specific instructions, sorted by mnemonic. Format, opcode, mnemonic, instruction name, and page number in the *EREF* are included in the table. For e200-specific instructions, page number is not shown. Instructions not listed here, but which are part of the Power ISA embedded category will either signal an illegal unimplemented or FP unavailable exception. Implementation-dependent instructions are noted with a footnote. Instructions which 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
- Enhanced reservation functionality
- Volatile context save/restore functionality

3.18.1 Instruction Index Sorted by Mnemonic

Table 3-5 shows the instructions sorted by mnemonic.

Table 3-5. Instructions Sorted by Mnemonic

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary ($\text{Inst}_{0:5}$)	Extended ($\text{Inst}_{21:31}$)			
X	011111	01000 01010 0	add	Add	223
X	011111	01000 01010 1	add.	Add & record CR	223
X	011111	00000 01010 0	addc	Add Carrying	224
X	011111	00000 01010 1	addc.	Add Carrying & record CR	224
X	011111	10000 01010 0	addco	Add Carrying & record OV	224
X	011111	10000 01010 1	addco.	Add Carrying & record OV & CR	224
X	011111	00100 01010 0	adde	Add Extended with CA	225
X	011111	00100 01010 1	adde.	Add Extended with CA & record CR	225
X	011111	10100 01010 0	addeo	Add Extended with CA & record OV	225

Legend:
- - 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’ Manual for the implementation

Table 3-5. Instructions Sorted by Mnemonic

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
X	011111	10100 01010 1	addeo.	Add Extended with CA & record OV & CR	225
D	001110	----- ----- -	addi	Add Immediate	226
D	001100	----- ----- -	addic	Add Immediate Carrying	227
D	001101	----- ----- -	addic.	Add Immediate Carrying & record CR	227
D	001111	----- ----- -	addis	Add Immediate Shifted	226
X	011111	00111 01010 0	addme	Add to Minus One Extended with CA	228
X	011111	00111 01010 1	addme.	Add to Minus One Extended with CA & record CR	228
X	011111	10111 01010 0	addmeo	Add to Minus One Extended with CA & record OV	228
X	011111	10111 01010 1	addmeo.	Add to Minus One Extended with CA & record OV & CR	228
X	011111	11000 01010 0	addo	Add & record OV	223
X	011111	11000 01010 1	addo.	Add & record OV & CR	223
X	011111	00110 01010 0	addze	Add to Zero Extended with CA	229
X	011111	00110 01010 1	addze.	Add to Zero Extended with CA & record CR	229
X	011111	10110 01010 0	addzeo	Add to Zero Extended with CA & record OV	229
X	011111	10110 01010 1	addzeo.	Add to Zero Extended with CA & record OV & CR	229
X	011111	00000 11100 0	and	AND	230
X	011111	00000 11100 1	and.	AND & record CR	230
X	011111	00001 11100 0	andc	AND with Complement	230
X	011111	00001 11100 1	andc.	AND with Complement & record CR	230
D	011100	----- ----- -	andi.	AND Immediate & record CR	230
D	011101	----- ----- -	andis.	AND Immediate Shifted & record CR	230
I	010010	----- ----0 0	b	Branch	231
I	010010	----- ----1 0	ba	Branch Absolute	231
B	010000	----- ----0 0	bc	Branch Conditional	232
B	010000	----- ----1 0	bca	Branch Conditional Absolute	232
XL	010011	10000 10000 0	bcctr	Branch Conditional to Count Register	233

Legend:

- 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’ Manual for the implementation

Table 3-5. Instructions Sorted by Mnemonic

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
XL	010011	10000 10000 1	bcctrl	Branch Conditional to Count Register & Link	233
B	010000	----- ----0 1	bcl	Branch Conditional & Link	232
B	010000	----- ----1 1	bcla	Branch Conditional & Link Absolute	232
XL	010011	00000 10000 0	bclr	Branch Conditional to Link Register	234
XL	010011	00000 10000 1	bclrl	Branch Conditional to Link Register & Link	234
I	010010	----- ----0 1	bl	Branch & Link	231
I	010010	----- ----1 1	bla	Branch & Link Absolute	231
X	011111	00000 00000 /	cmp	Compare	235
D	001011	----- ----- -	cmpi	Compare Immediate	235
X	011111	00001 00000 /	cmpl	Compare Logical	236
D	001010	----- ----- -	cmpli	Compare Logical Immediate	236
X	011111	00000 11010 0	cntlzw	Count Leading Zeros Word	237
X	011111	00000 11010 1	cntlzw.	Count Leading Zeros Word & record CR	237
XL	010011	01000 00001 /	crand	Condition Register AND	238
XL	010011	00100 00001 /	crandc	Condition Register AND with Complement	238
XL	010011	01001 00001 /	creqv	Condition Register Equivalent	238
XL	010011	00111 00001 /	crnand	Condition Register NAND	239
XL	010011	00001 00001 /	crnor	Condition Register NOR	239
XL	010011	01110 00001 /	cror	Condition Register OR	239
XL	010011	01101 00001 /	crorc	Condition Register OR with Complement	240
XL	010011	00110 00001 /	crxor	Condition Register XOR	240
X	011111	10111 10110 /	dcba	Data Cache Block Allocate	241
X	011111	00010 10110 /	dcbf	Data Cache Block Flush	242
X	011111	01110 10110 /	dcbi	Data Cache Block Invalidate	243
X	011111	01100 00110 /	dcblc[1]	Data Cache Block Lock Clear	----
X	011111	00001 10110 /	dcbst	Data Cache Block Store	245

Legend:

- 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’ Manual for the implementation

Table 3-5. Instructions Sorted by Mnemonic

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary (Inst$_{0:5}$)	Extended (Inst$_{21:31}$)			
X	011111	01000 10110 /	dcbt	Data Cache Block Touch	246
X	011111	00101 00110 /	dcbtls[1]	Data Cache Block Touch and Lock Set	----
X	011111	00111 10110 /	dcbtst	Data Cache Block Touch for Store	247
X	011111	00100 00110 /	dcbtstls[1]	Data Cache Block Touch for Store and Lock Set	----
X	011111	11111 10110 /	dcbz	Data Cache Block set to Zero	248
X	011111	01111 01011 0	divw	Divide Word	251
X	011111	01111 01011 1	divw.	Divide Word & record CR	251
X	011111	11111 01011 0	divwo	Divide Word & record OV	251
X	011111	11111 01011 1	divwo.	Divide Word & record OV & CR	251
X	011111	01110 01011 0	divwu	Divide Word Unsigned	252
X	011111	01110 01011 1	divwu.	Divide Word Unsigned & record CR	252
X	011111	11110 01011 0	divwuo	Divide Word Unsigned & record OV	252
X	011111	11110 01011 1	divwuo.	Divide Word Unsigned & record OV & CR	252
X	011111	01000 11100 0	eqv	Equivalent	253
X	011111	01000 11100 1	eqv.	Equivalent & record CR	253
X	011111	11101 11010 0	extsb	Extend Sign Byte	254
X	011111	11101 11010 1	extsb.	Extend Sign Byte & record CR	254
X	011111	11100 11010 0	extsh	Extend Sign Half Word	254
X	011111	11100 11010 1	extsh.	Extend Sign Half Word and record CR	254
X	011111	11110 10110 /	icbi	Instruction Cache Block Invalidate	280
X	011111	00111 00110 /	icblc[1]	Instruction Cache Block Lock Clear	----
X	011111	00000 10110 /	icbt	Instruction Cache Block Touch	281
X	011111	01111 00110 /	icbtls[1]	Instruction Cache Block Touch and Lock Set	----
??	011111	----- 01111 /	isel[2]	Integer Select	----
XL	010011	00100 10110 /	isync	Instruction Synchronize	282
D	100010	----- ----- -	lbz	Load Byte & Zero	283

Legend:

- 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’ Manual for the implementation

Table 3-5. Instructions Sorted by Mnemonic

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
D	100011	----- ----- -	lbzu	Load Byte & Zero with Update	283
X	011111	00011 10111 /	lbzux	Load Byte & Zero with Update Indexed	283
X	011111	00010 10111 /	lbzx	Load Byte & Zero Indexed	283
D	101010	----- ----- -	lha	Load Half Word Algebraic	288
D	101011	----- ----- -	lhau	Load Half Word Algebraic with Update	288
X	011111	01011 10111 /	lhaux	Load Half Word Algebraic with Update Indexed	288
X	011111	01010 10111 /	lhax	Load Half Word Algebraic Indexed	288
X	011111	11000 10110 /	lhbrx	Load Half Word Byte-Reverse Indexed	289
D	101000	----- ----- -	lhz	Load Half Word and Zero	290
D	101001	----- ----- -	lhzu	Load Half Word & Zero with Update	290
X	011111	01001 10111 /	lhzux	Load Half Word & Zero with Update Indexed	290
X	011111	01000 10111 /	lhzx	Load Half Word & Zero Indexed	290
D	101110	----- ----- -	lmw	Load Multiple Word	291
X	011111	00000 10100 /	lwarx[3]	Load Word & Reserve Indexed	294
X	011111	10000 10110 /	lwbrx	Load Word Byte-Reverse Indexed	296
D	100000	----- ----- -	lwz	Load Word & Zero	297
D	100001	----- ----- -	lwzu	Load Word & Zero with Update	297
X	011111	00001 10111 /	lwzux	Load Word & Zero with Update Indexed	297
X	011111	00000 10111 /	lwzx	Load Word & Zero Indexed	297
X	011111	11010 10110 /	mbar[3]	Memory Barrier	298
XL	010011	00000 00000 /	mcrf	Move Condition Register Field	299
X	011111	10000 00000 /	mcrxr	Move to Condition Register from XER	300
X	011111	00000 10011 /	mfcr	Move From Condition Register	301
XFX	011111	01010 00011 /	mfdcr	Move From Device Control Register	302
X	011111	00010 10011 /	mfmsr	Move From Machine State Register	303
XFX	011111	01010 10011 /	mfspr	Move From Special Purpose Register	304

Legend:
- 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’ Manual for the implementation

Table 3-5. Instructions Sorted by Mnemonic

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary (Inst$_{0:5}$)	Extended (Inst$_{21:31}$)			
X	011111	10010 10110 /	msync[3]	Memory Synchronize	305
XFX	011111	00100 10000 /	mtcrf	Move To Condition Register Fields	306
XFX	011111	01110 00011 /	mtdcr	Move To Device Control Register	307
X	011111	00100 10010 /	mtmsr	Move To Machine State Register	311
XFX	011111	01110 10011 /	mtspr	Move To Special Purpose Register	312
X	011111	/0010 01011 0	mulhw	Multiply High Word	314
X	011111	/0010 01011 1	mulhw.	Multiply High Word & record CR	314
X	011111	/0000 01011 0	mulhwu	Multiply High Word Unsigned	314
X	011111	/0000 01011 1	mulhwu.	Multiply High Word Unsigned & record CR	314
D	000111	----- ----- -	mulli	Multiply Low Immediate	315
X	011111	00111 01011 0	mullw	Multiply Low Word	316
X	011111	00111 01011 1	mullw.	Multiply Low Word & record CR	316
X	011111	10111 01011 0	mullwo	Multiply Low Word & record OV	316
X	011111	10111 01011 1	mullwo.	Multiply Low Word & record OV & CR	316
X	011111	01110 11100 0	nand	NAND	317
X	011111	01110 11100 1	nand.	NAND & record CR	317
X	011111	00011 01000 0	neg	Negate	318
X	011111	00011 01000 1	neg.	Negate & record CR	318
X	011111	10011 01000 0	nego	Negate & record OV	318
X	011111	10011 01000 1	nego.	Negate & record OV & record CR	318
X	011111	00011 11100 0	nor	NOR	319
X	011111	00011 11100 1	nor.	NOR & record CR	319
X	011111	01101 11100 0	or	OR	320
X	011111	01101 11100 1	or.	OR & record CR	320
X	011111	01100 11100 0	orc	OR with Complement	320
X	011111	01100 11100 1	orc.	OR with Complement & record CR	320

Legend:
- 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’ Manual for the implementation

Table 3-5. Instructions Sorted by Mnemonic

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary (Inst$_{0:5}$)	**Extended (Inst$_{21:31}$)**			
D	011000	----- ----- -	ori	OR Immediate	320
D	011001	----- ----- -	oris	OR Immediate Shifted	320
XL	010011	00001 10011 /	rfci	Return From Critical Interrupt	321
XL	010011	00001 00111 /	rfdi[4]	Return From Debug Interrupt	----
XL	010011	00001 10010 /	rfi	Return From Interrupt	322
XL	010011	00001 00110 /	rfmci[5]	Return From Machine Check Interrupt	----
M	010100	----- ----- 0	rlwimi	Rotate Left Word Immediate then Mask Insert	327
M	010100	----- ----- 1	rlwimi.	Rotate Left Word Immediate then Mask Insert & record CR	327
M	010101	----- ----- 0	rlwinm	Rotate Left Word Immediate then AND with Mask	328
M	010101	----- ----- 1	rlwinm.	Rotate Left Word Immediate then AND with Mask & record CR	328
M	010111	----- ----- 0	rlwnm	Rotate Left Word then AND with Mask	328
M	010111	----- ----- 1	rlwnm.	Rotate Left Word then AND with Mask & record CR	328
SC	010001	///// ////1 /	sc	System Call	330
X	011111	00000 11000 0	slw	Shift Left Word	332
X	011111	00000 11000 1	slw.	Shift Left Word & record CR	332
X	011111	11000 11000 0	sraw	Shift Right Algebraic Word	334
X	011111	11000 11000 1	sraw.	Shift Right Algebraic Word & record CR	334
X	011111	11001 11000 0	srawi	Shift Right Algebraic Word Immediate	334
X	011111	11001 11000 1	srawi.	Shift Right Algebraic Word Immediate & record CR	334
X	011111	10000 11000 0	srw	Shift Right Word	336
X	011111	10000 11000 1	srw.	Shift Right Word & record CR	336
D	100110	----- ----- -	stb	Store Byte	337
D	100111	----- ----- -	stbu	Store Byte with Update	337
X	011111	00111 10111 /	stbux	Store Byte with Update Indexed	337
X	011111	00110 10111 /	stbx	Store Byte Indexed	337
D	101100	----- ----- -	sth	Store Half Word	343

Legend:

- - 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’ Manual for the implementation

Table 3-5. Instructions Sorted by Mnemonic

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary (Inst$_{0:5}$)	Extended (Inst$_{21:31}$)			
X	011111	11100 10110 /	sthbrx	Store Half Word Byte-Reverse Indexed	344
D	101101	----- ----- -	sthu	Store Half Word with Update	343
X	011111	01101 10111 /	sthux	Store Half Word with Update Indexed	343
X	011111	01100 10111 /	sthx	Store Half Word Indexed	343
D	101111	----- ----- -	stmw	Store Multiple Word	345
D	100100	----- ----- -	stw	Store Word	347
X	011111	10100 10110 /	stwbrx	Store Word Byte-Reverse Indexed	348
X	011111	00100 10110 1	stwcx.[3]	Store Word Conditional Indexed & record CR	349
D	100101	----- ----- -	stwu	Store Word with Update	347
X	011111	00101 10111 /	stwux	Store Word with Update Indexed	347
X	011111	00100 10111 /	stwx	Store Word Indexed	347
X	011111	00001 01000 0	subf	Subtract From	351
X	011111	00001 01000 1	subf.	Subtract From & record CR	351
X	011111	00000 01000 0	subfc	Subtract From Carrying	352
X	011111	00000 01000 1	subfc.	Subtract From Carrying & record CR	352
X	011111	10000 01000 0	subfco	Subtract From Carrying & record OV	352
X	011111	10000 01000 1	subfco.	Subtract From Carrying & record OV & CR	352
X	011111	00100 01000 0	subfe	Subtract From Extended with CA	353
X	011111	00100 01000 1	subfe.	Subtract From Extended with CA & record CR	353
X	011111	10100 01000 0	subfeo	Subtract From Extended with CA & record OV	353
X	011111	10100 01000 1	subfeo.	Subtract From Extended with CA & record OV & CR	353
D	001000	----- ----- -	subfic	Subtract From Immediate Carrying	354
X	011111	00111 01000 0	subfme	Subtract From Minus One Extended with CA	355
X	011111	00111 01000 1	subfme.	Subtract From Minus One Extended with CA & record CR	355
X	011111	10111 01000 0	subfmeo	Subtract From Minus One Extended with CA & record OV	355
X	011111	10111 01000 1	subfmeo.	Subtract From Minus One Extended with CA & record OV & CR	355

Legend:

- 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’ Manual for the implementation

Table 3-5. Instructions Sorted by Mnemonic

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
X	011111	10001 01000 0	subfo	Subtract From & record OV	351
X	011111	10001 01000 1	subfo.	Subtract From & record OV & CR	351
X	011111	00110 01000 0	subfze	Subtract From Zero Extended with CA	356
X	011111	00110 01000 1	subfze.	Subtract From Zero Extended with CA & record CR	356
X	011111	10110 01000 0	subfzeo	Subtract From Zero Extended with CA & record OV	356
X	011111	10110 01000 1	subfzeo.	Subtract From Zero Extended with CA & record OV & CR	356
X	011111	11000 10010 /	tlbivax	TLB Invalidate Virtual Address Indexed	358
X	011111	11101 10010 /	tlbre	TLB Read Entry	359
X	011111	11100 10010 ?	tlbsx	TLB Search Indexed	360
X	011111	10001 10110 /	tlbsync	TLB Synchronize	361
X	011111	11110 10010 /	tlbwe	TLB Write Entry	362
X	011111	00000 00100 /	tw	Trap Word	363
D	000011	----- ----- -	twi	Trap Word Immediate	363
X	011111	00100 00011 /	wrtee	Write External Enable	364
X	011111	00101 00011 /	wrteei	Write External Enable Immediate	364
X	011111	01001 11100 0	xor	XOR	365
X	011111	01001 11100 1	xor.	XOR & record CR	365
D	011010	----- ----- -	xori	XOR Immediate	365
D	011011	----- ----- -	xoris	XOR Immediate Shifted	365

Legend:

- 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’ Manual for the implementation

[1] Motorola Book E cache locking APU, refer to Section 9.10, “Cache Line Locking/Unlocking” on page 9-19

[2] Motorola Book E **isel** APU, refer to Section 3.10, “ISEL Instruction” on page 3-6

[3] See Section 3.6, “Memory Synchronization and Reservation Instructions” on page 3-3

[4] See Section 3.11, “Enhanced Debug” on page 3-6

[5] See Section 3.12, “Machine Check” on page 3-10

3.18.2 Instruction Index Sorted by Opcode

Table 3-6 lists instructions sorted by opcode.

Table 3-6. Instructions Sorted by Opcode

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
D	000011	----- ----- -	twi	Trap Word Immediate	363
D	000111	----- ----- -	mulli	Multiply Low Immediate	315
D	001000	----- ----- -	subfic	Subtract From Immediate Carrying	354
D	001010	----- ----- -	cmpli	Compare Logical Immediate	236
D	001011	----- ----- -	cmpi	Compare Immediate	235
D	001100	----- ----- -	addic	Add Immediate Carrying	227
D	001101	----- ----- -	addic.	Add Immediate Carrying & record CR	227
D	001110	----- ----- -	addi	Add Immediate	226
D	001111	----- ----- -	addis	Add Immediate Shifted	226
B	010000	----- ----0 0	bc	Branch Conditional	232
B	010000	----- ----0 1	bcl	Branch Conditional & Link	232
B	010000	----- ----1 0	bca	Branch Conditional Absolute	232
B	010000	----- ----1 1	bcla	Branch Conditional & Link Absolute	232
SC	010001	///// ////1 /	sc	System Call	330
I	010010	----- ----0 0	b	Branch	231
I	010010	----- ----0 1	bl	Branch & Link	231
I	010010	----- ----1 0	ba	Branch Absolute	231
I	010010	----- ----1 1	bla	Branch & Link Absolute	231
XL	010011	00000 00000 /	mcrf	Move Condition Register Field	299
XL	010011	00000 10000 0	bclr	Branch Conditional to Link Register	234
XL	010011	00000 10000 1	bclrl	Branch Conditional to Link Register & Link	234
XL	010011	00001 00001 /	crnor	Condition Register NOR	239
XL	010011	00001 00110 /	rfmci	Return From Machine Check Interrupt	----
XL	010011	00001 00111 /	rfdi	Return From Debug Interrupt	----

Legend:
- 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' Manual for the implementation

Table 3-6. Instructions Sorted by Opcode (Continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary ($\text{Inst}_{0:5}$)	Extended ($\text{Inst}_{21:31}$)			
XL	010011	00001 10010 /	rfi	Return From Interrupt	322
XL	010011	00001 10011 /	rfci	Return From Critical Interrupt	321
XL	010011	00100 00001 /	crandc	Condition Register AND with Complement	238
XL	010011	00100 10110 /	isync	Instruction Synchronize	282
XL	010011	00110 00001 /	crxor	Condition Register XOR	240
XL	010011	00111 00001 /	crnand	Condition Register NAND	239
XL	010011	01000 00001 /	crand	Condition Register AND	238
XL	010011	01001 00001 /	creqv	Condition Register Equivalent	238
XL	010011	01101 00001 /	crorc	Condition Register OR with Complement	240
XL	010011	01110 00001 /	cror	Condition Register OR	239
XL	010011	10000 10000 0	bcctr	Branch Conditional to Count Register	233
XL	010011	10000 10000 1	bcctrl	Branch Conditional to Count Register & Link	233
M	010100	----- ----- 0	rlwimi	Rotate Left Word Immediate then Mask Insert	327
M	010100	----- ----- 1	rlwimi.	Rotate Left Word Immediate then Mask Insert & record CR	327
M	010101	----- ----- 0	rlwinm	Rotate Left Word Immediate then AND with Mask	328
M	010101	----- ----- 1	rlwinm.	Rotate Left Word Immediate then AND with Mask & record CR	328
M	010111	----- ----- 0	rlwnm	Rotate Left Word then AND with Mask	328
M	010111	----- ----- 1	rlwnm.	Rotate Left Word then AND with Mask & record CR	328
D	011000	----- ----- -	ori	OR Immediate	320
D	011001	----- ----- -	oris	OR Immediate Shifted	320
D	011010	----- ----- -	xori	XOR Immediate	365
D	011011	----- ----- -	xoris	XOR Immediate Shifted	365
D	011100	----- ----- -	andi.	AND Immediate & record CR	230
D	011101	----- ----- -	andis.	AND Immediate Shifted & record CR	230
??	011111	----- 01111 /	isel	Integer Select	----
X	011111	00000 00000 /	cmp	Compare	235

Legend:

- 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’ Manual for the implementation

Table 3-6. Instructions Sorted by Opcode (Continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary (Inst$_{0:5}$)	Extended (Inst$_{21:31}$)			
X	011111	00000 00100 /	tw	Trap Word	363
X	011111	00000 01000 0	subfc	Subtract From Carrying	352
X	011111	00000 01000 1	subfc.	Subtract From Carrying & record CR	352
X	011111	00000 01010 0	addc	Add Carrying	224
X	011111	00000 01010 1	addc.	Add Carrying & record CR	224
X	011111	/0000 01011 0	mulhwu	Multiply High Word Unsigned	314
X	011111	/0000 01011 1	mulhwu.	Multiply High Word Unsigned & record CR	314
X	011111	00000 10011 /	mfcr	Move From Condition Register	301
X	011111	00000 10100 /	lwarx	Load Word & Reserve Indexed	294
X	011111	00000 10110 /	icbt	Instruction Cache Block Touch	281
X	011111	00000 10111 /	lwzx	Load Word & Zero Indexed	297
X	011111	00000 11000 0	slw	Shift Left Word	332
X	011111	00000 11000 1	slw.	Shift Left Word & record CR	332
X	011111	00000 11010 0	cntlzw	Count Leading Zeros Word	237
X	011111	00000 11010 1	cntlzw.	Count Leading Zeros Word & record CR	237
X	011111	00000 11100 0	and	AND	230
X	011111	00000 11100 1	and.	AND & record CR	230
X	011111	00001 00000 /	cmpl	Compare Logical	236
X	011111	00001 01000 0	subf	Subtract From	351
X	011111	00001 01000 1	subf.	Subtract From & record CR	351
X	011111	00001 10110 /	dcbst	Data Cache Block Store	245
X	011111	00001 10111 /	lwzux	Load Word & Zero with Update Indexed	297
X	011111	00001 11100 0	andc	AND with Complement	230
X	011111	00001 11100 1	andc.	AND with Complement & record CR	230
X	011111	/0010 01011 0	mulhw	Multiply High Word	314
X	011111	/0010 01011 1	mulhw.	Multiply High Word & record CR	314

Legend:

- 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’ Manual for the implementation

Table 3-6. Instructions Sorted by Opcode (Continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary (Inst$_{0:5}$)	Extended (Inst$_{21:31}$)			
X	011111	00010 10011 /	mfmsr	Move From Machine State Register	303
X	011111	00010 10110 /	dcbf	Data Cache Block Flush	242
X	011111	00010 10111 /	lbzx	Load Byte & Zero Indexed	283
X	011111	00011 01000 0	neg	Negate	318
X	011111	00011 01000 1	neg.	Negate & record CR	318
X	011111	00011 10111 /	lbzux	Load Byte & Zero with Update Indexed	283
X	011111	00011 11100 0	nor	NOR	319
X	011111	00011 11100 1	nor.	NOR & record CR	319
X	011111	00100 00011 /	wrtee	Write External Enable	364
X	011111	00100 00110 /	dcbtstls[1]	Data Cache Block Touch for Store and Lock Set	----
X	011111	00100 01000 0	subfe	Subtract From Extended with CA	353
X	011111	00100 01000 1	subfe.	Subtract From Extended with CA & record CR	353
X	011111	00100 01010 0	adde	Add Extended with CA	225
X	011111	00100 01010 1	adde.	Add Extended with CA & record CR	225
XFX	011111	00100 10000 /	mtcrf	Move To Condition Register Fields	306
X	011111	00100 10010 /	mtmsr	Move To Machine State Register	311
X	011111	00100 10110 1	stwcx.	Store Word Conditional Indexed & record CR	349
X	011111	00100 10111 /	stwx	Store Word Indexed	347
X	011111	00101 00011 /	wrteei	Write External Enable Immediate	364
X	011111	00101 00110 /	dcbtls[1]	Data Cache Block Touch and Lock Set	----
X	011111	00101 10111 /	stwux	Store Word with Update Indexed	347
X	011111	00110 01000 0	subfze	Subtract From Zero Extended with CA	356
X	011111	00110 01000 1	subfze.	Subtract From Zero Extended with CA & record CR	356
X	011111	00110 01010 0	addze	Add to Zero Extended with CA	229
X	011111	00110 01010 1	addze.	Add to Zero Extended with CA & record CR	229
X	011111	00110 10111 /	stbx	Store Byte Indexed	337

Legend:
- 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’ Manual for the implementation

Table 3-6. Instructions Sorted by Opcode (Continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary (Inst$_{0:5}$)	Extended (Inst$_{21:31}$)			
X	011111	00111 00110 /	icblc[1]	Instruction Cache Block Lock Clear	----
X	011111	00111 01000 0	subfme	Subtract From Minus One Extended with CA	355
X	011111	00111 01000 1	subfme.	Subtract From Minus One Extended with CA & record CR	355
X	011111	00111 01010 0	addme	Add to Minus One Extended with CA	228
X	011111	00111 01010 1	addme.	Add to Minus One Extended with CA & record CR	228
X	011111	00111 01011 0	mullw	Multiply Low Word	316
X	011111	00111 01011 1	mullw.	Multiply Low Word & record CR	316
X	011111	00111 10110 /	dcbtst	Data Cache Block Touch for Store	247
X	011111	00111 10111 /	stbux	Store Byte with Update Indexed	337
X	011111	01000 01010 0	add	Add	223
X	011111	01000 01010 1	add.	Add & record CR	223
X	011111	01000 10110 /	dcbt	Data Cache Block Touch	246
X	011111	01000 10111 /	lhzx	Load Half Word & Zero Indexed	290
X	011111	01000 11100 0	eqv	Equivalent	253
X	011111	01000 11100 1	eqv.	Equivalent & record CR	253
X	011111	01001 10111 /	lhzux	Load Half Word & Zero with Update Indexed	290
X	011111	01001 11100 0	xor	XOR	365
X	011111	01001 11100 1	xor.	XOR & record CR	365
XFX	011111	01010 00011 /	mfdcr	Move From Device Control Register	302
XFX	011111	01010 10011 /	mfspr	Move From Special Purpose Register	304
X	011111	01010 10111 /	lhax	Load Half Word Algebraic Indexed	288
X	011111	01011 10111 /	lhaux	Load Half Word Algebraic with Update Indexed	288
X	011111	01100 00110 /	dcblc[1]	Data Cache Block Lock Clear	----
X	011111	01100 10111 /	sthx	Store Half Word Indexed	343
X	011111	01100 11100 0	orc	OR with Complement	320
X	011111	01100 11100 1	orc.	OR with Complement & record CR	320

Legend:

- 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’ Manual for the implementation

Table 3-6. Instructions Sorted by Opcode (Continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
X	011111	01101 10111 /	sthux	Store Half Word with Update Indexed	343
X	011111	01101 11100 0	or	OR	320
X	011111	01101 11100 1	or.	OR & record CR	320
XFX	011111	01110 00011 /	mtdcr	Move To Device Control Register	307
X	011111	01110 01011 0	divwu	Divide Word Unsigned	252
X	011111	01110 01011 1	divwu.	Divide Word Unsigned & record CR	252
XFX	011111	01110 10011 /	mtspr	Move To Special Purpose Register	312
X	011111	01110 10110 /	dcbi	Data Cache Block Invalidate	243
X	011111	01110 11100 0	nand	NAND	317
X	011111	01110 11100 1	nand.	NAND & record CR	317
X	011111	01111 00110 /	icbtls[1]	Instruction Cache Block Touch and Lock Set	----
X	011111	01111 01011 0	divw	Divide Word	251
X	011111	01111 01011 1	divw.	Divide Word & record CR	251
X	011111	10000 00000 /	mcrxr	Move to Condition Register from XER	300
X	011111	10000 01000 0	subfco	Subtract From Carrying & record OV	352
X	011111	10000 01000 1	subfco.	Subtract From Carrying & record OV & CR	352
X	011111	10000 01010 0	addco	Add Carrying & record OV	224
X	011111	10000 01010 1	addco.	Add Carrying & record OV & CR	224
X	011111	10000 10110 /	lwbrx	Load Word Byte-Reverse Indexed	296
X	011111	10000 11000 0	srw	Shift Right Word	336
X	011111	10000 11000 1	srw.	Shift Right Word & record CR	336
X	011111	10001 01000 0	subfo	Subtract From & record OV	351
X	011111	10001 01000 1	subfo.	Subtract From & record OV & CR	351
X	011111	10001 10110 /	tlbsync	TLB Synchronize	361
X	011111	10010 10110 /	msync	Memory Synchronize	305
X	011111	10011 01000 0	nego	Negate & record OV	318

Legend:

- 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’ Manual for the implementation

Table 3-6. Instructions Sorted by Opcode (Continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
X	011111	10011 01000 1	nego.	Negate & record OV & record CR	318
X	011111	10100 01000 0	subfeo	Subtract From Extended with CA & record OV	353
X	011111	10100 01000 1	subfeo.	Subtract From Extended with CA & record OV & CR	353
X	011111	10100 01010 0	addeo	Add Extended with CA & record OV	225
X	011111	10100 01010 1	addeo.	Add Extended with CA & record OV & CR	225
X	011111	10100 10110 /	stwbrx	Store Word Byte-Reverse Indexed	348
X	011111	10110 01000 0	subfzeo	Subtract From Zero Extended with CA & record OV	356
X	011111	10110 01000 1	subfzeo.	Subtract From Zero Extended with CA & record OV & CR	356
X	011111	10110 01010 0	addzeo	Add to Zero Extended with CA & record OV	229
X	011111	10110 01010 1	addzeo.	Add to Zero Extended with CA & record OV & CR	229
X	011111	10111 01000 0	subfmeo	Subtract From Minus One Extended with CA & record OV	355
X	011111	10111 01000 1	subfmeo.	Subtract From Minus One Extended with CA & record OV & CR	355
X	011111	10111 01010 0	addmeo	Add to Minus One Extended with CA & record OV	228
X	011111	10111 01010 1	addmeo.	Add to Minus One Extended with CA & record OV & CR	228
X	011111	10111 01011 0	mullwo	Multiply Low Word & record OV	316
X	011111	10111 01011 1	mullwo.	Multiply Low Word & record OV & CR	316
X	011111	10111 10110 /	dcba	Data Cache Block Allocate	241
X	011111	11000 01010 0	addo	Add & record OV	223
X	011111	11000 01010 1	addo.	Add & record OV & CR	223
X	011111	11000 10010 /	tlbivax	TLB Invalidate Virtual Address Indexed	358
X	011111	11000 10110 /	lhbrx	Load Half Word Byte-Reverse Indexed	289
X	011111	11000 11000 0	sraw	Shift Right Algebraic Word	334
X	011111	11000 11000 1	sraw.	Shift Right Algebraic Word & record CR	334
X	011111	11001 11000 0	srawi	Shift Right Algebraic Word Immediate	334
X	011111	11001 11000 1	srawi.	Shift Right Algebraic Word Immediate & record CR	334
X	011111	11010 10110 /	mbar	Memory Barrier	298

Legend:

- 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’ Manual for the implementation

Table 3-6. Instructions Sorted by Opcode (Continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary (Inst$_{0:5}$)	Extended (Inst$_{21:31}$)			
X	011111	11100 10010 ?	tlbsx	TLB Search Indexed	360
X	011111	11100 10110 /	sthbrx	Store Half Word Byte-Reverse Indexed	344
X	011111	11100 11010 0	extsh	Extend Sign Half Word	254
X	011111	11100 11010 1	extsh.	Extend Sign Half Word & record CR	254
X	011111	11101 10010 /	tlbre	TLB Read Entry	359
X	011111	11101 11010 0	extsb	Extend Sign Byte	254
X	011111	11101 11010 1	extsb.	Extend Sign Byte & record CR	254
X	011111	11110 01011 0	divwuo	Divide Word Unsigned & record OV	252
X	011111	11110 01011 1	divwuo.	Divide Word Unsigned & record OV & CR	252
X	011111	11110 10010 /	tlbwe	TLB Write Entry	362
X	011111	11110 10110 /	icbi	Instruction Cache Block Invalidate	280
X	011111	11111 01011 0	divwo	Divide Word & record OV	251
X	011111	11111 01011 1	divwo.	Divide Word & record OV & CR	251
X	011111	11111 10110 /	dcbz	Data Cache Block set to Zero	248
D	100000	----- ----- -	lwz	Load Word & Zero	297
D	100001	----- ----- -	lwzu	Load Word & Zero with Update	297
D	100010	----- ----- -	lbz	Load Byte & Zero	283
D	100011	----- ----- -	lbzu	Load Byte & Zero with Update	283
D	100100	----- ----- -	stw	Store Word	347
D	100101	----- ----- -	stwu	Store Word with Update	347
D	100110	----- ----- -	stb	Store Byte	337
D	100111	----- ----- -	stbu	Store Byte with Update	337
D	101000	----- ----- -	lhz	Load Half Word & Zero	290
D	101001	----- ----- -	lhzu	Load Half Word & Zero with Update	290
D	101010	----- ----- -	lha	Load Half Word Algebraic	288
D	101011	----- ----- -	lhau	Load Half Word Algebraic with Update	288

Legend:

-	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’ Manual for the implementation

Table 3-6. Instructions Sorted by Opcode (Continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary (Inst$_{0:5}$)	Extended (Inst$_{21:31}$)			
D	101100	----- ----- -	sth	Store Half Word	343
D	101101	----- ----- -	sthu	Store Half Word with Update	343
D	101110	----- ----- -	lmw	Load Multiple Word	291
D	101111	----- ----- -	stmw	Store Multiple Word	345

[1] Freescale Power ISA cache locking category, refer to Section 9.10, “Cache Line Locking/Unlocking.”

Legend:

- 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’ Manual for the implementation

Chapter 4
Instruction Pipeline and Execution Timing

This section describes the e200z4 instruction pipeline and instruction timing information. The core is partitioned into the following subsystems:

- Instruction unit
- Control unit
- Branch unit
- Instruction decode unit
- Exception handling unit
- Execution units
- Core interface

4.1 Overview of Operation

A block diagram of the e200z446n3 core is shown in Figure 4-1. 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 GPRs or from the operand feed-forward muxes. Data or resource hazards may create stall conditions that 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 writeback 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.

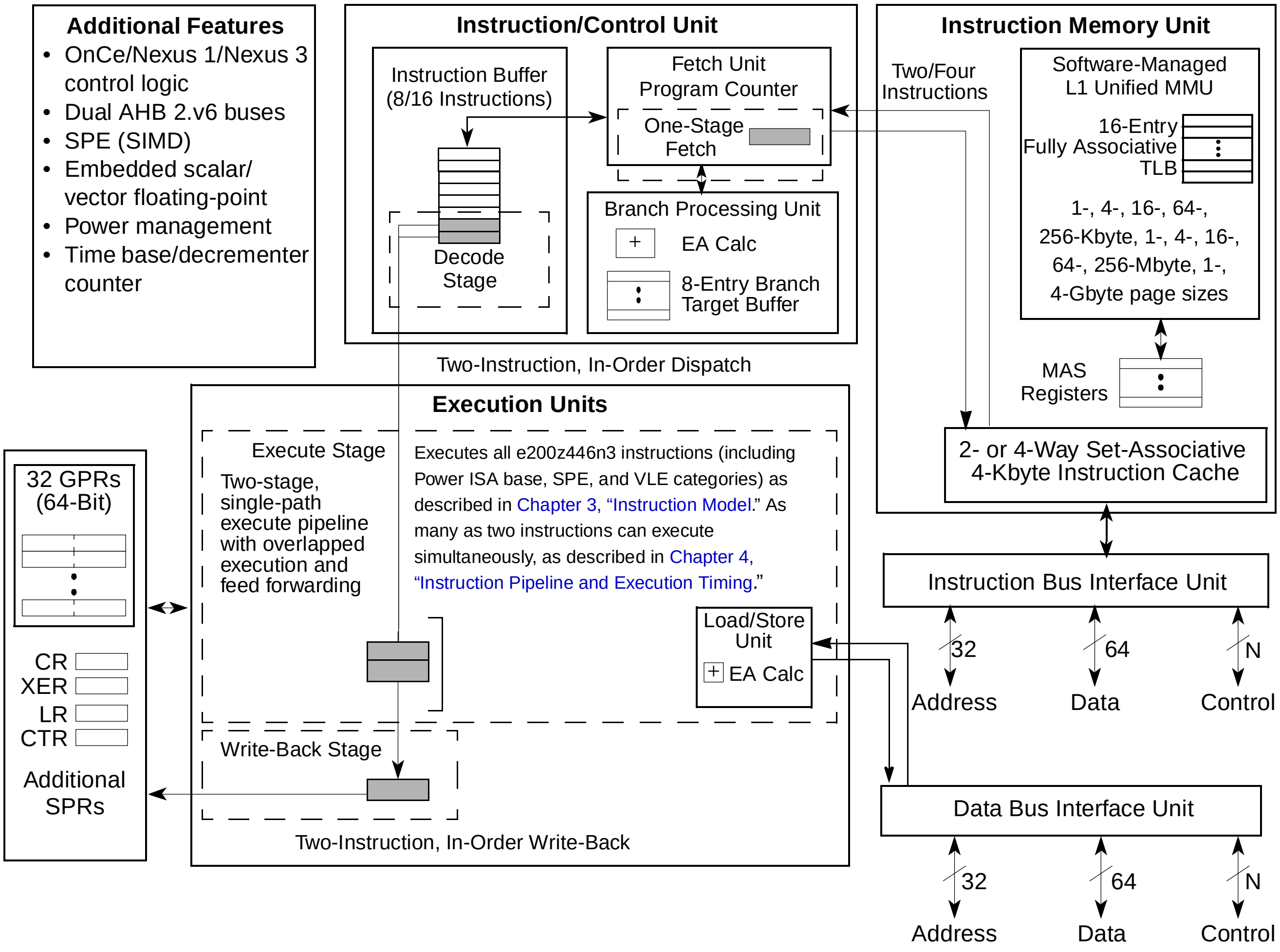

Additional Features
• OnCe/Nexus 1/Nexus 3 control logic
• Dual AHB 2.v6 buses
• SPE (SIMD)
• Embedded scalar/ vector floating-point
• Power management
• Time base/decrementer counter
Instruction/Control Unit
Instruction Buffer (8/16 Instructions)
Decode Stage
Fetch Unit
Program Counter
One-Stage Fetch
Branch Processing Unit
EA Calc
8-Entry Branch Target Buffer
Instruction Memory Unit
Two/Four Instructions
Software-Managed L1 Unified MMU
16-Entry Fully Associative TLB
1-, 4-, 16-, 64-, 256-Kbyte, 1-, 4-, 16-, 64-, 256-Mbyte, 1-, 4-Gbyte page sizes
MAS Registers
2- or 4-Way Set-Associative 4-Kbyte Instruction Cache
Two-Instruction, In-Order Dispatch
Execution Units
Execute Stage
Two-stage, single-path execute pipeline with overlapped execution and feed forwarding
Executes all e200z446n3 instructions (including Power ISA base, SPE, and VLE categories) as described in Chapter 3, “Instruction Model.” As many as two instructions can execute simultaneously, as described in Chapter 4, “Instruction Pipeline and Execution Timing.”
Load/Store Unit
EA Calc
Write-Back Stage
Two-Instruction, In-Order Write-Back
32 GPRs (64-Bit)
CR
XER
LR
CTR
Additional SPRs
Instruction Bus Interface Unit
32
Address
64
Data
N
Control
Data Bus Interface Unit
32
Address
64
Data
N
Control

Figure 4-1. e200z4 Block Diagram

Table 4-1 shows the e200z446n3 concurrent instruction issue capabilities. Note that data dependencies between instructions 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	—	4	4	4	4	4	—
Load/store	4	—	4	4	4	4	—
Scalar Integer	4	4	4[1]	4	4[2]	4	—
Scalar Float	4	4	4	4	4	—	—
Vector Integer	4	4	4[2]	4	4[3]	4	—
Vector Float	4	4	4	—	4	—	—
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

4.2 Core Subsystems

This section provides a brief overview of the core subsystems.

4.2.1 Control Unit

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

4.2.2 Instruction Unit

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

4.2.3 Branch Unit

The branch unit executes branch instructions, predicts conditional branches, and provides branch target addresses for instruction fetches. It contains an 8-entry branch target buffer to accelerate execution of branch instructions.

4.2.4 Instruction Decode Unit

The decode unit includes the instruction buffers. A pair of instructions can be decoded each clock 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.

4.2.5 Exception Handling

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

4.3 Execution Units

The core data execution units consist of the integer units, SPE floating-point units, and the load/store unit. Included in the execution units section are the 32 general-purpose registers (GPRs). Instructions with data dependencies begin execution when all such dependencies are resolved.

4.3.1 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.

Multiply instructions have a latency of 2 cycles with a maximum throughput of 1 per cycle.

Divide instructions have a variable latency (4–14 cycles) depending upon the operand data. The worst case integer divide will take 14 cycles. While the divide is running, the rest of the pipeline is unavailable for additional instructions (blocking divide).

4.3.2 Load/Store Unit

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

4.3.3 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 2 cycles with a maximum throughput of 1 per cycle. SPE floating-point divide instructions have a latency of 13 cycles. While the divide is running, the rest of the pipeline is unavailable for additional instructions (blocking divide).

4.4 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.

Table 4-2 explains the five pipeline stages.

Table 4-2. Pipeline Stages

STAGE	Description
IFETCH	Instruction Fetch From Memory
DECODE/RF READ/FF/ MEM EA	Instruction Decode/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
WB	Write Back to Registers

Figure 4-2 shows the pipeline diagram.

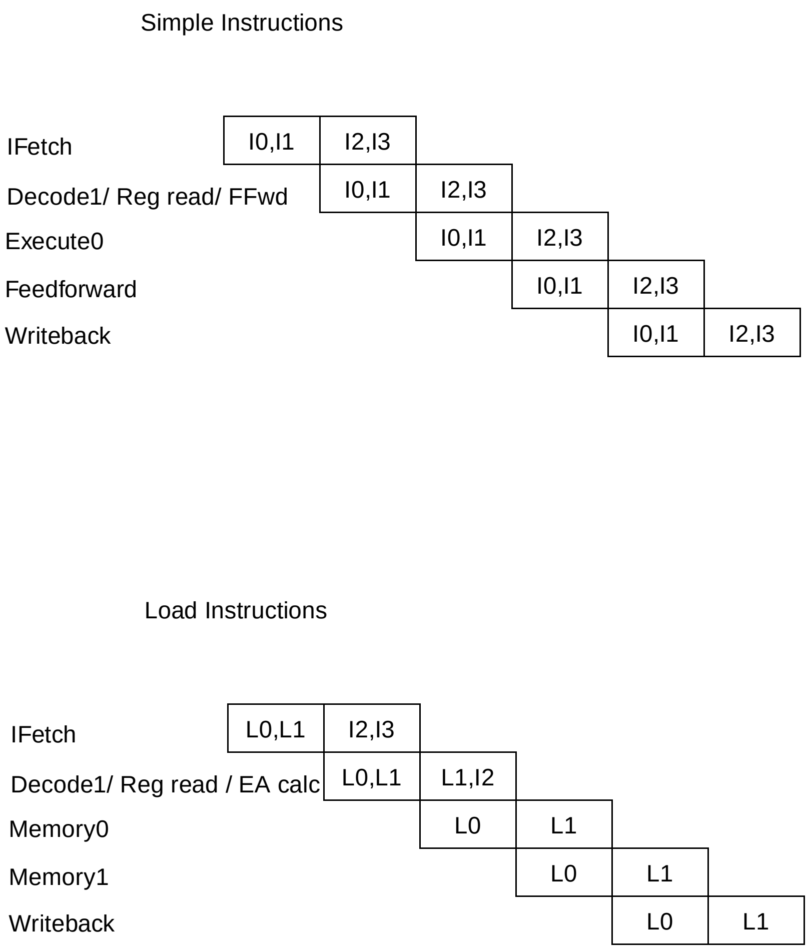


Figure 4-2. Pipeline Diagram

4.4.1 Description of Pipeline Stages

The IFetch pipeline stage retrieves instructions from the memory system and determines 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 stage decodes instructions, reads operands from the register file, and performs dependency checking.

Execution occurs in one or both of the execute pipeline stages in each execution unit (perhaps over multiple cycles). Execution of most load/store instructions is pipelined. The load/store unit has three pipeline stages: effective address calculation (EA Calc), initial memory access (MEM0), and final memory access, data format, and forward (MEM1).

Simple integer instructions complete execution in the Execute 0 stage of the pipeline. Multiply instructions require both the Execute 0 and Execute 1 stages but may be pipelined as well. Most condition-setting instructions complete in the Execute 0 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 does not wait until the completion of the result write-back. Feed forward hardware is supplied to allow bypassing of completed instructions from both execute stages into the first execution stage for a subsequent data-dependent instruction.

4.4.2 Instruction Prefetch Buffers and Branch Target Buffer

The e200z4 contains an eight-entry instruction prefetch buffer, which 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.

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. 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, 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 buffers.

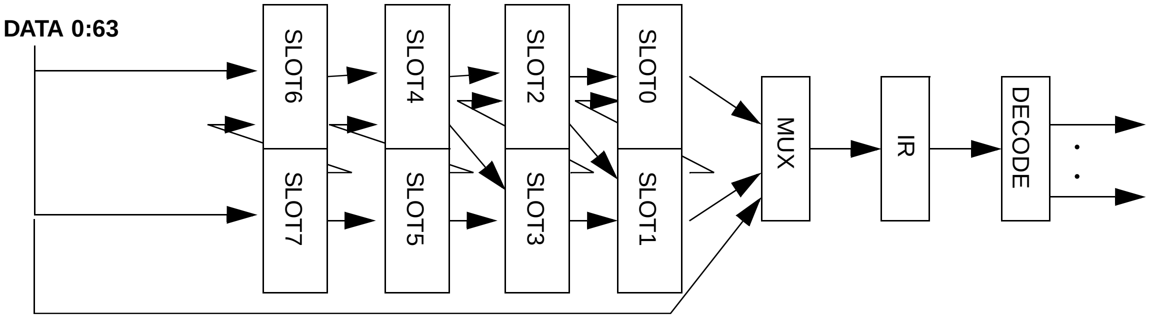


Figure 4-3. e200z4 Instruction Prefetch Buffers

To resolve branch instructions and improve the accuracy of branch predictions, the e200z4 implements a dynamic branch prediction mechanism using an 8-entry branch target buffer (BTB).

An entry is allocated in the BTB whenever a branch resolves as taken and the BTB is enabled. 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 will be 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 e200z4 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 the BUCSR[BALLOC] field 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 e200z4 reverts to a static prediction mechanism using the BUCSR[BPRED] field 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 e200z4 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 7

IS = Instruction Space

Figure 4-4. e200z4 Branch Target Buffer

4.4.3 Single-Cycle Instruction Pipeline Operation

Sequences of single-cycle execution instructions follow the flow in Figure 4-5. Instructions are issued and completed in program order. Most arithmetic and logical instructions fall into this category. Instructions may feed-forward results of execution at the end of the E0 or FF stage.

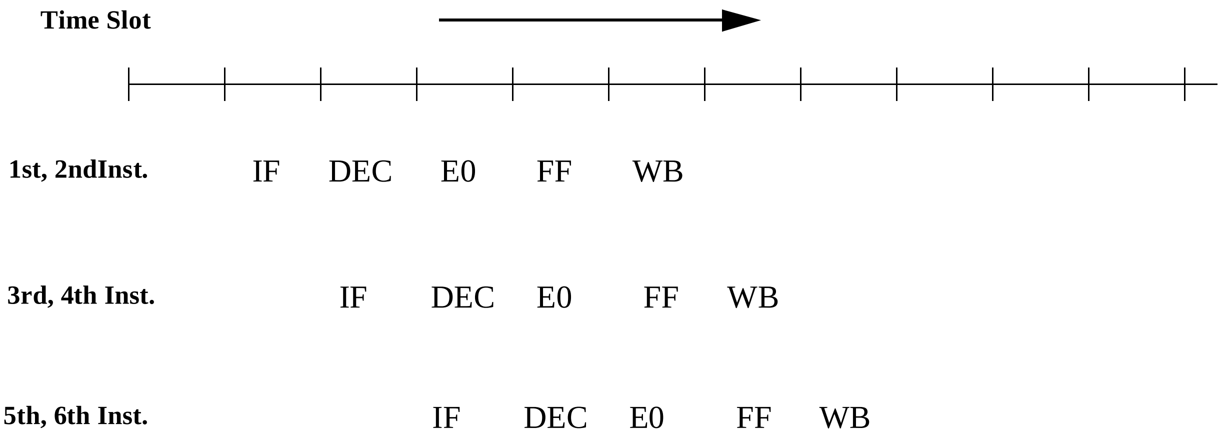


Figure 4-5. Basic Pipe Line Flow, Single Cycle Instructions

4.4.4 Basic Load and Store Instruction Pipeline Operation

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 MEM1, and the result is available at the end of MEM1 for the following instruction. If the instruction has a data dependency on the result of a load, there is a single stall cycle. Data will be fed-forward from the preceding load at the end of the MEM1 stage.

Figure 4-6 shows the basic pipe line flow for the load/store instructions.

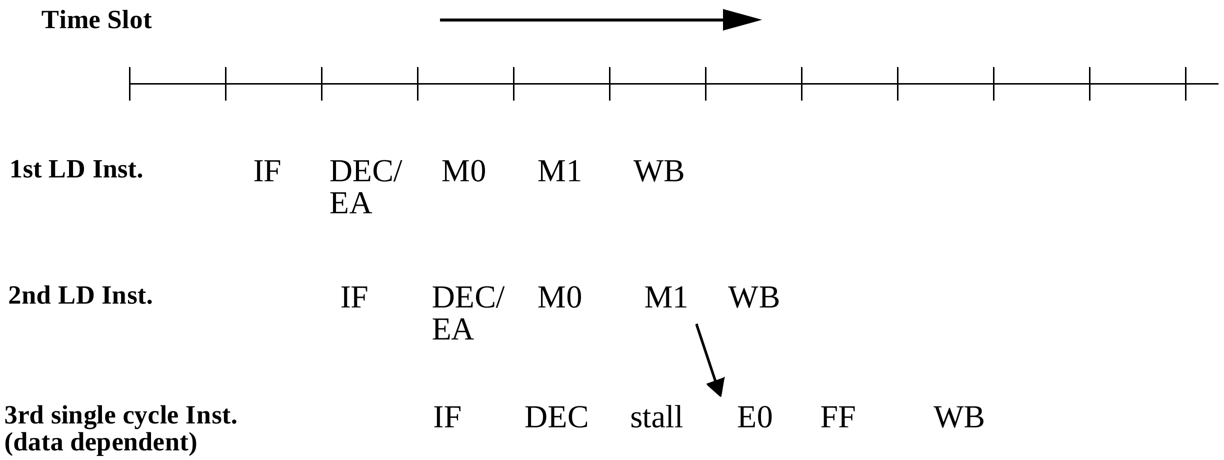


Figure 4-6. Basic Pipe Line Flow, Load/Store Instructions

4.4.5 Change-of-Flow Instruction Pipeline Operation

Simple change of flow instructions require 2 clock cycles to refill the pipeline with the target instruction for taken branches and branch and link instructions with no BTB hit and correct branch prediction.

Figure 4-7 shows the basic pipe line flow for the change of flow instructions.

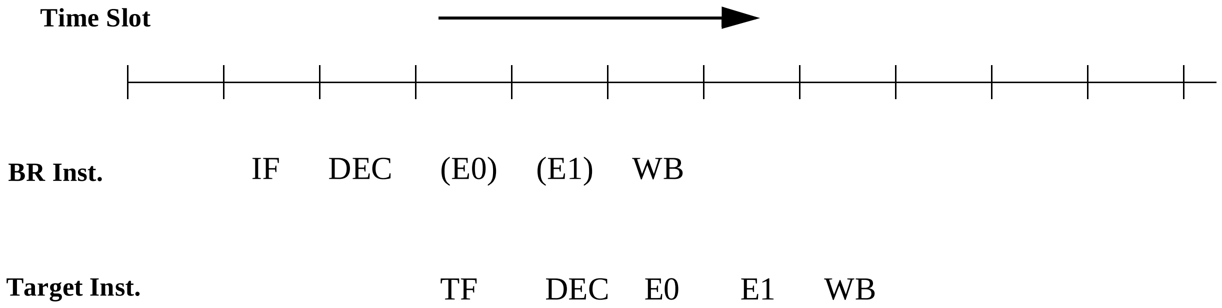


Figure 4-7. Basic Pipe Line Flow, Branch Instructions (BTB Miss, Correct Prediction, Branch Taken)

This 2 cycle timing may be reduced for branch type instructions 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 is reduced to a single clock when the target fetch is initiated early enough and the branch is correctly predicted.

Figure 4-8 shows the basic pipe line flow for the reduced timing.

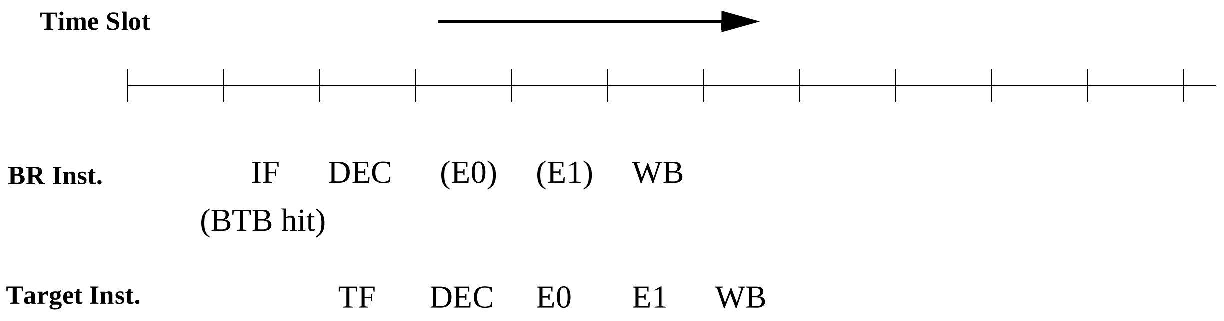


Figure 4-8. Basic Pipe Line Flow, Branch Instructions (BTB Hit, Correct Prediction, Branch Taken)

For certain cases where the branch is incorrectly predicted, 3 cycles are required for the not-taken branch, which must correct the misprediction outcome. Figure 4-9 shows one example.

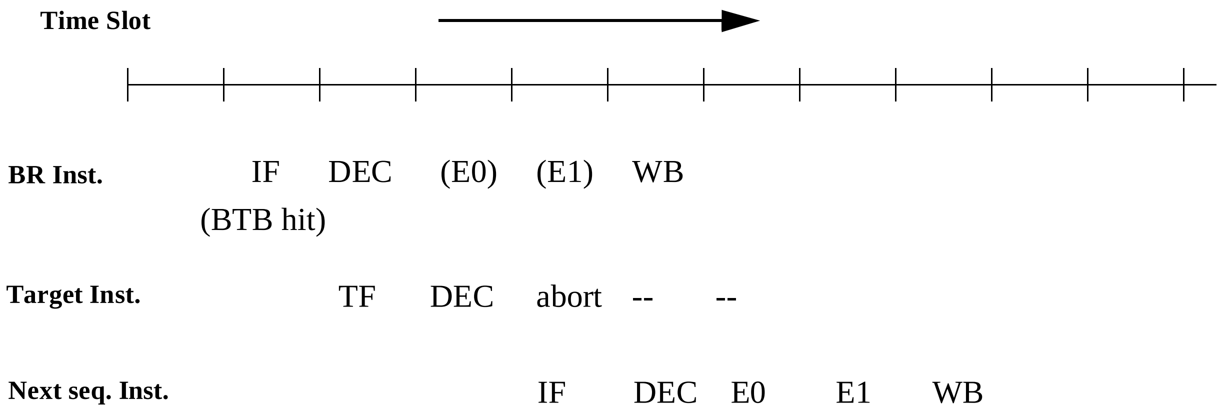
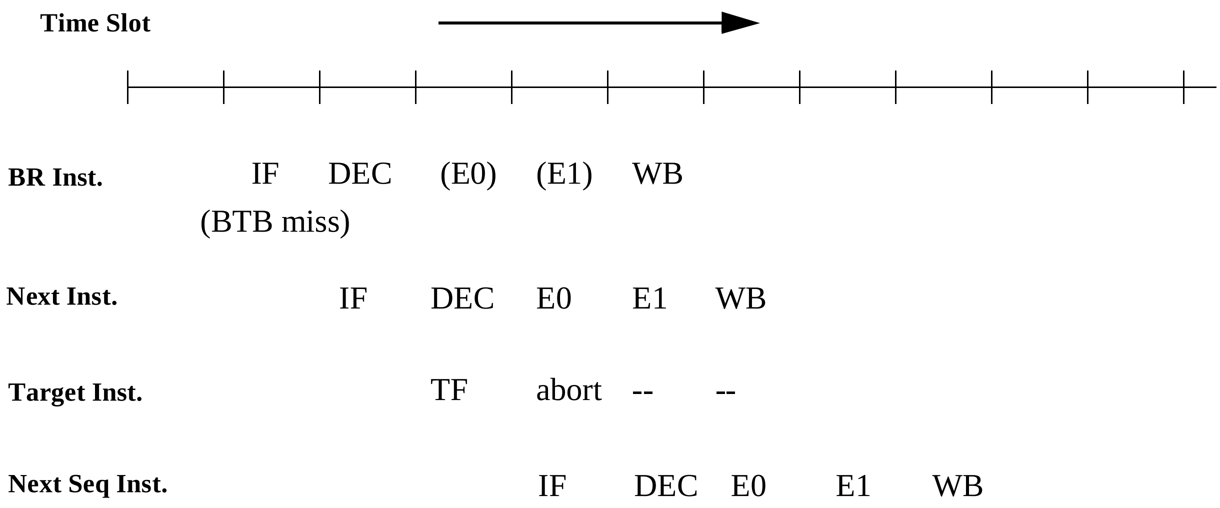


Figure 4-9. Basic Pipe Line Flow, Branch Instruction (BTB Hit, Predict Taken, Incorrect Prediction)

For certain other cases where the branch is incorrectly predicted as taken, a stall cycle is required to correct the misprediction outcome and begin refilling the instruction buffer. Figure 4-10 shows one example.



Time Slot										
BR Inst.		IF (BTB miss)	DEC	(E0)	(E1)	WB				
Next Inst.			IF	DEC	E0	E1	WB			
Target Inst.				TF	abort	--	--			
Next Seq Inst.					IF	DEC	E0	E1	WB	

Figure 4-10. Basic Pipe Line Flow, Branch Instructions (BTB Miss, Predict Taken, Incorrect Prediction, Instruction Buffer Empty)

4.4.6 Basic Multi-Cycle Instruction Pipeline Operation

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

Figure 4-11 shows the basic pipe line flow for multi-cycle instruction.

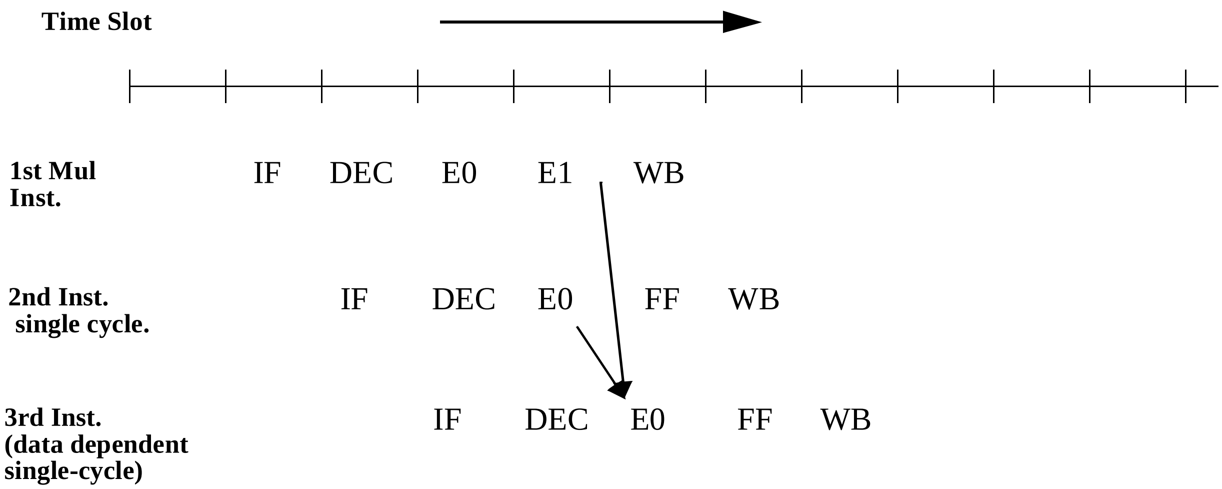


Figure 4-11. Basic Pipe Line Flow, Multiply Class Instructions

Since load and store instructions calculate the effective address in the DEC stage, any dependency on a previous instruction for EA calculation may stall the load or store in DEC until the result is available. Figure 4-12 shows the infrequent case of a load instruction dependent on a multiply instruction.

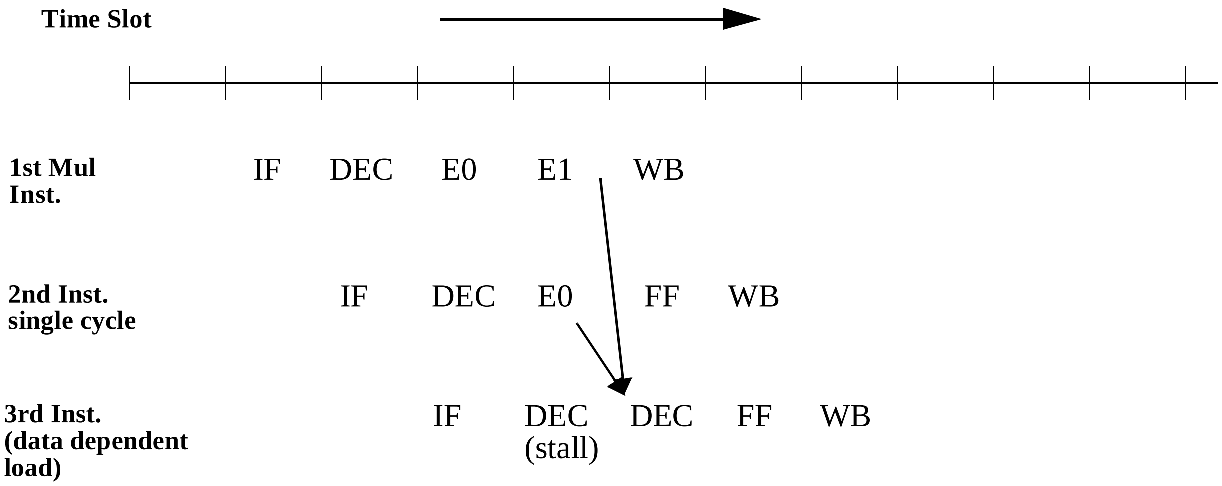


Figure 4-12. Pipe Line Flow, Multiply with Data Dependent Load Instruction

The divide and load and store multiple instructions require multiple cycles in the execute stage as shown in Figure 4-13.

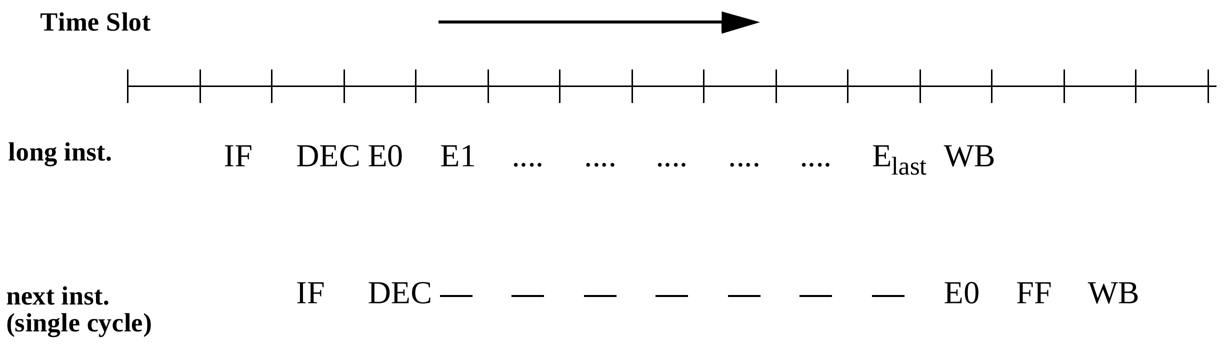


Figure 4-13. Basic Pipe Line Flow, long instruction

4.4.7 Additional Examples of Instruction Pipeline Operation for Load and Store

Figure 4-14 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, even during the MEM0 or MEM1 stage. The **add** in this example will not stall, even though a data dependency exists on the updated base register of the load with update.

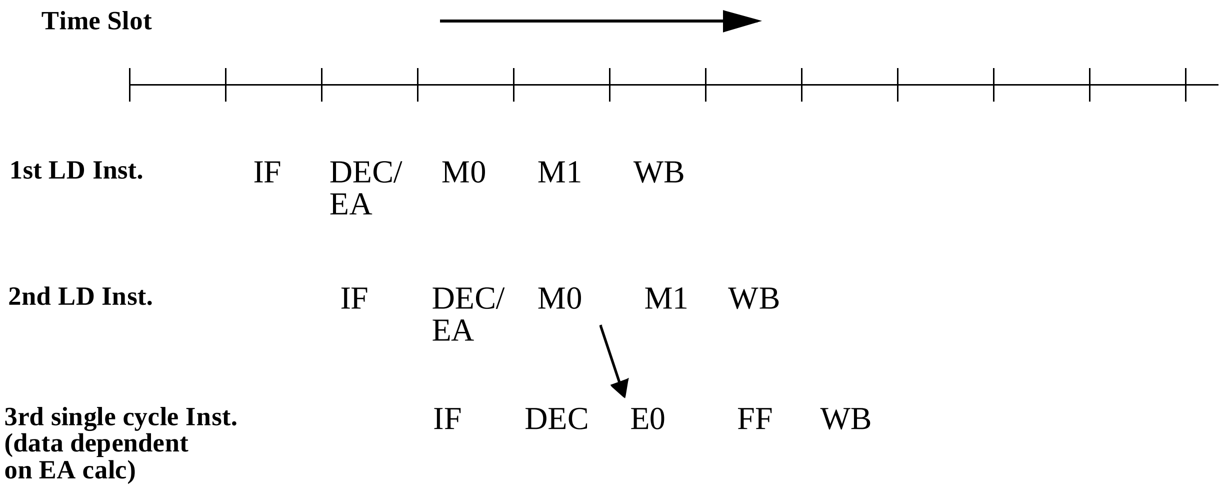


Figure 4-14. Pipe Line Flow, Load/Store Instructions with Base Register Update

Figure 4-15 shows an example of pipelining a data-dependent store instruction following a load instruction. The **store** in this example will stall, due to the store data dependency existing on the load data of the load instruction.

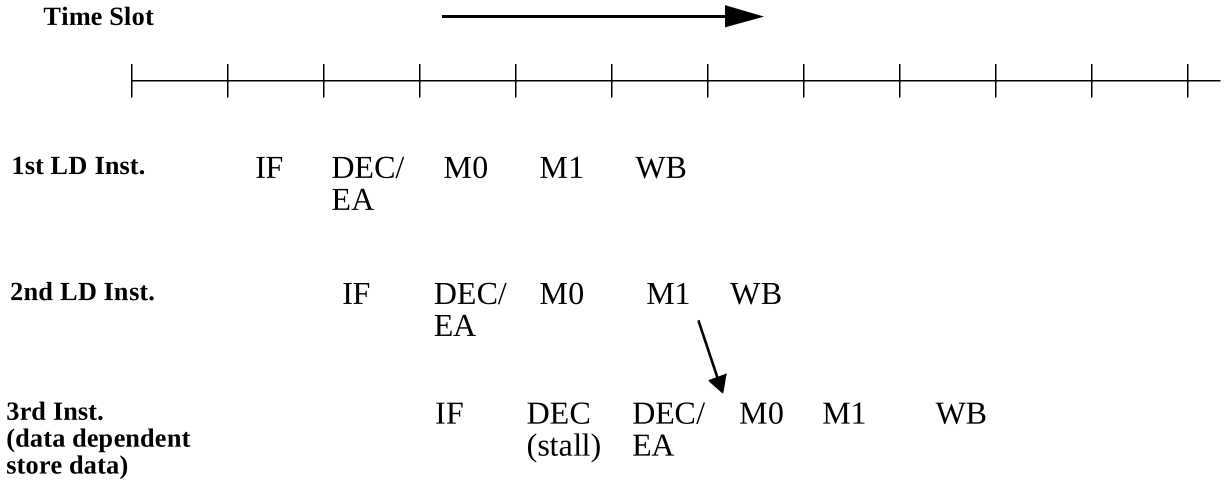


Figure 4-15. Pipelined Store Instruction with Store Data Dependency

4.4.8 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. The following SPRs are exceptions and do cause stalls:

- MSR
- Debug SPRs
- The SPE unit
- Cache/MMU SPRs

Figure 4-16–Figure 4-18 show examples of **mtspr** and **mfspr** instruction timing.

Figure 4-16 applies to the debug SPRs and 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.

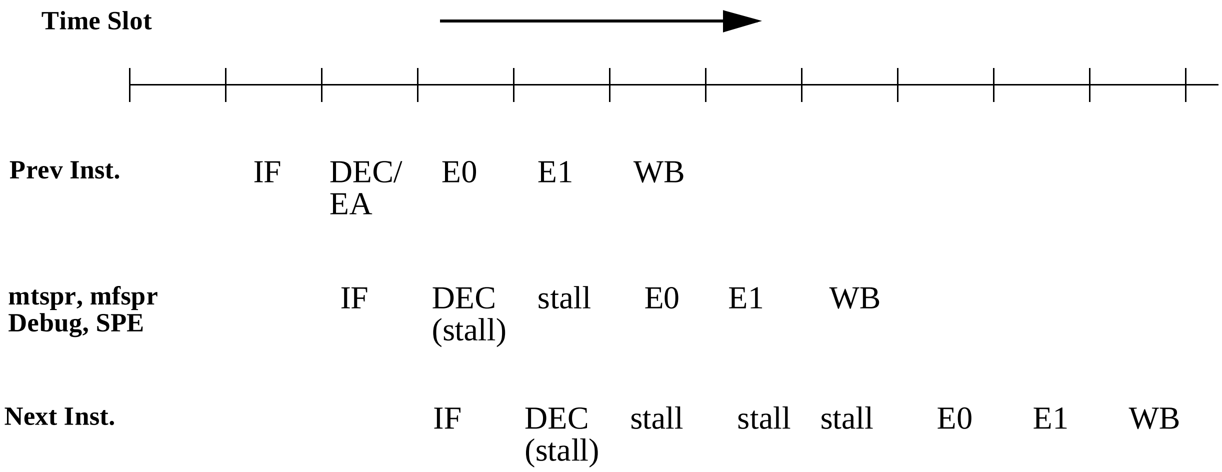


Figure 4-16. mtspr, mfspr **Instruction Execution, Debug and SPE SPRs**

Figure 4-17 applies to the **mtmsr** instruction and the **wrtee** and **wrteei** instructions. Execution of subsequent instructions is stalled until the cycle after these instructions write-back.

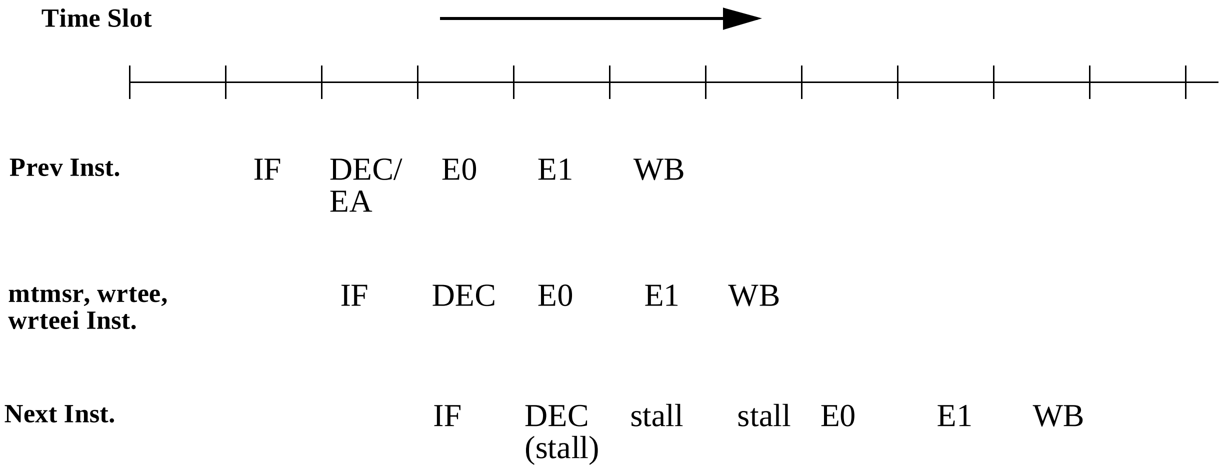


Figure 4-17. mtmsr, wrtee[i] **Instruction Execution**

Access to cache and MMU SPRs are stalled until all outstanding bus accesses have completed on both interfaces and the Cache and MMU are idle (**p_[d,i]_cmbusy** negated) to allow an access window where no translations or cache cycles are required. 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 (i.e. during the WB cycle). This same protocol applies to cache and MMU management instructions (e.g. **icbi**, **tlbre**, **tlbwe**, etc.) as well as the DCRs.

Figure 4-18 shows an example where an outstanding bus access causes **mtspr**/**mfspr** execution to be delayed until the bus becomes idle.

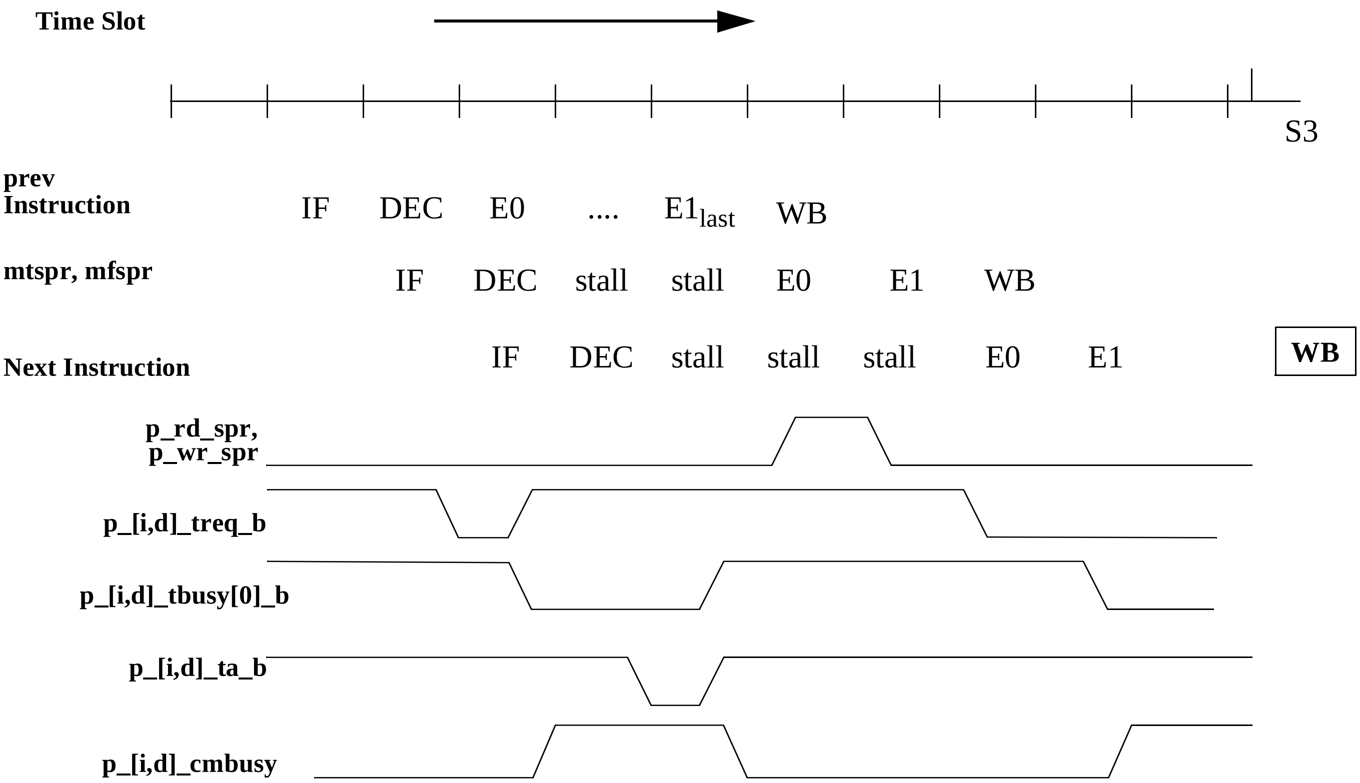


Figure 4-18. Cache/DCR, MMU mtspr, mfspr **and MMU Management Instruction Execution**

4.5 Control Hazards

Internal control hazards exist in the e200z4 that can cause certain instruction sequences to incur one or more stall cycles. One such hazard is an **mfspr** instruction preceded by a **mtspr** instruction. This causes issue stalls until the **mtspr** completes.

4.6 Instruction Serialization

There are three types of serialization required by the core:

- Completion
- Dispatch (Decode/Issue)
- Refetch

4.6.1 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:

- Instructions that access or modify system control or status registers, such as **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, such as **isync**, **se_isync**, **msync**, **rfi**, **rfci**, **rfdi**, **rfmci**, **se_rfi**, **se_rfci**, **se_rfdi**, **se_rfmci**, **sc**, **se_sc.**
- **wait**

4.6.2 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, complete. Instructions that 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.

4.6.3 Refetch Serialization

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

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

4.7 Interrupt Recognition and Exception Processing

Figure 4-19 shows timing for interrupt recognition and exception processing overhead. This example shows best-case response timing when an interrupt is received and processed during execution of a sequence of single-cycle instructions. The handler is present in the cache and proceeds with no bubbles.

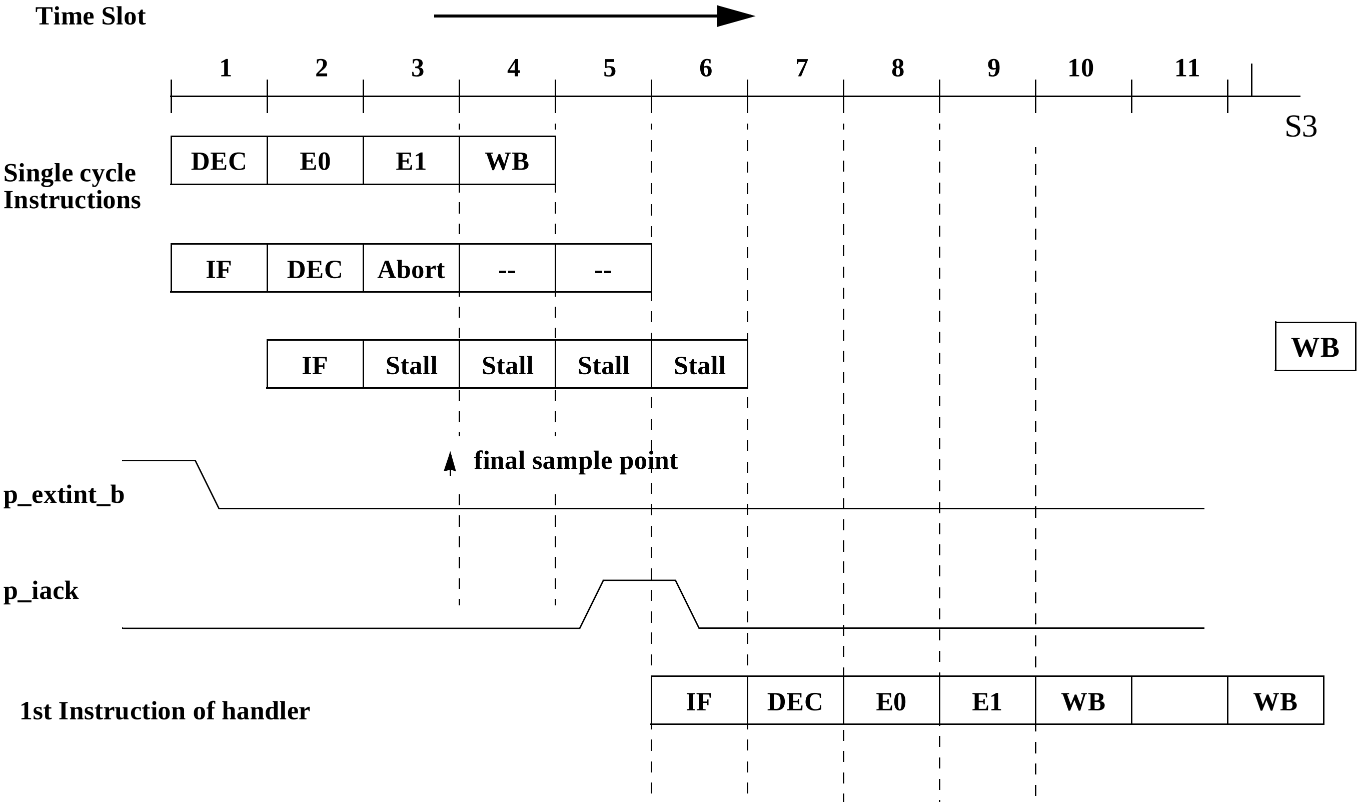


Figure 4-19. Interrupt Recognition and Handler Instruction Execution

Figure 4-20 shows timing for interrupt recognition and exception processing overhead. This example shows best-case response timing when an interrupt is received and processed during execution of a load or store instruction. The fetch for the handler is delayed until completion of any outstanding load or store, regardless of the number of wait-states.

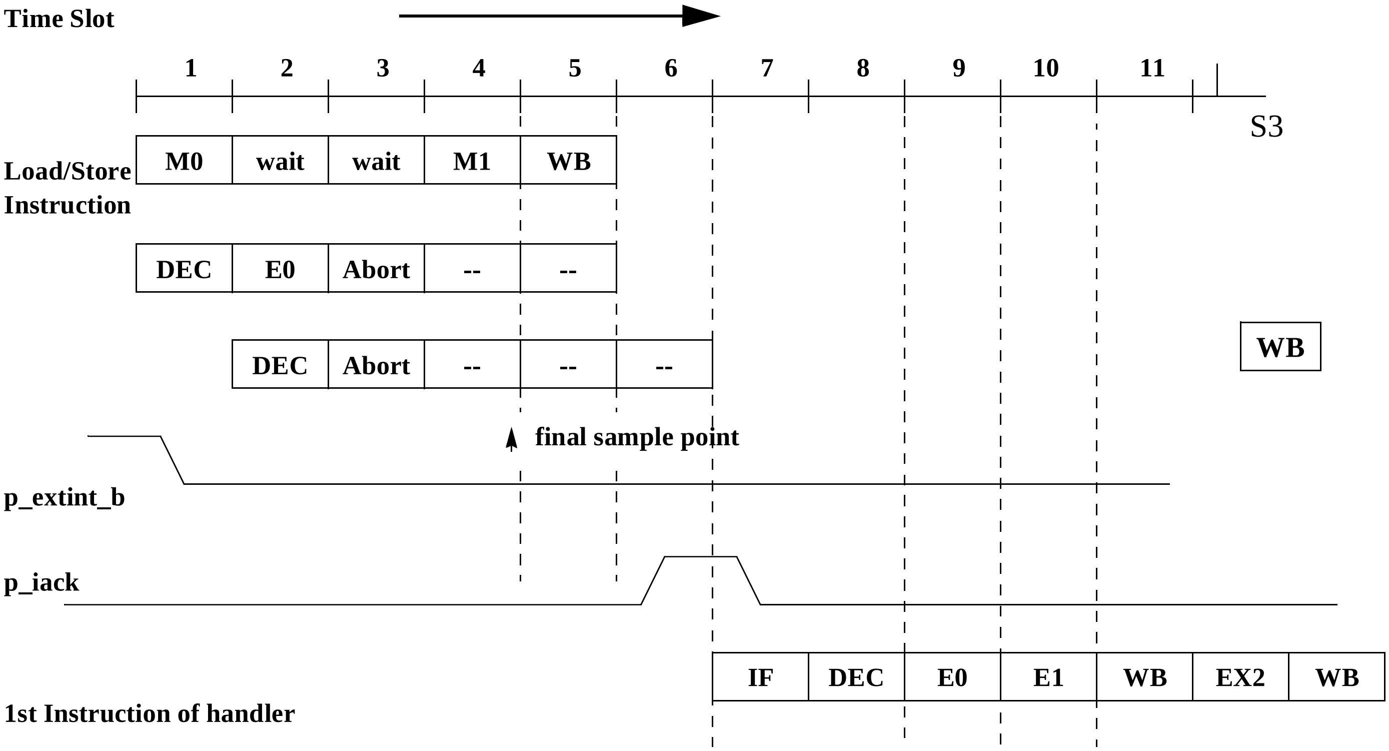


Figure 4-20. Interrupt Recognition and Handler Instruction Execution—Load/Store in Progress

Figure 4-21 shows timing for interrupt recognition and exception processing overhead. This example shows best-case response timing when an interrupt is received and processed during execution of a multicycle interruptible instructions. The handler is present in the cache and proceeds with no bubbles.

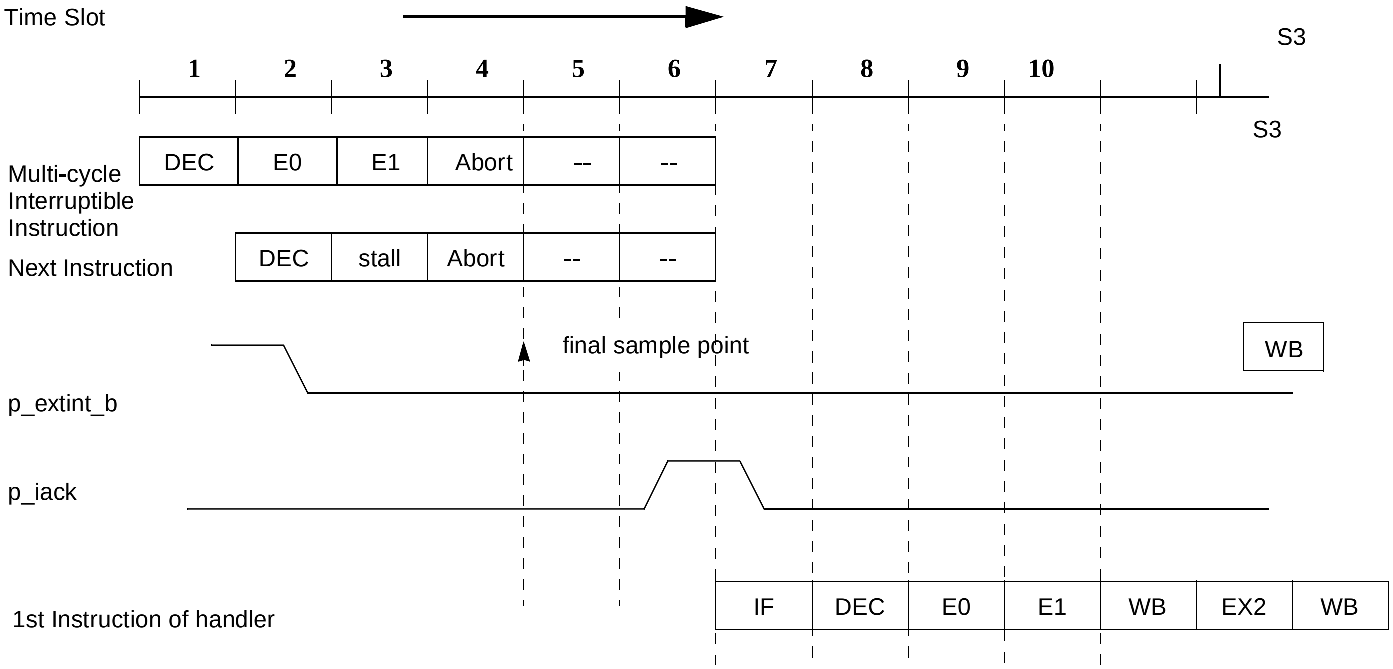


Figure 4-21. Interrupt Recognition and Handler Instruction Execution—Multi-Cycle Instruction Abort

4.8 Concurrent Instruction Execution

The core effectively has several 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.

The following instruction timing section 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.

4.9 Instruction Timings

Table 4-3 shows instruction timing in number of processor clock cycles for various instruction classes. 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 7.6, “SPE Instruction Timing.” Timing for EFPU2 instructions is detailed in Section 6.5, “EFPU 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 a ‘&’ require a 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	3/2/1	3/2/1	Correct branch lookahead allows single cycle execution worst-case mispredicted branch is 3 cycles
multiply	2	1	—
divide	4–14	4–14	data dependent timing
CR logical	1	1	—
loads (non-multiple)	2	1	—
load multiple	2 + n ÷ 2 (max)	1 + n ÷ 2 (max)	Actual timing depends on n and address alignment.
stores (non-multiple)	2	1	—
store multiple	2 + n ÷ 2 (max)	1 + n ÷ 2 (max)	Actual timing depends on n and address alignment.
mtmsr, **wrtee**, **wrteei**	3&	3	—
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**	3	—	—
sc	4	—	—
tw, **twi**	4	—	Trap taken timing

Detailed timing for each instruction mnemonic along with serialization requirements is shown in Table 4-4.

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]	3/2/1	none
bc[l][a]	3/2/1	none
bcctr[l]	3/2	none
bclr[l]	3/2	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][.]	4-14[1]	none
divwu[o][.]	4-14[1]	none

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

Mnemonic	Latency	Serialization
eqv[.]	1	none
extsb[.]	1	none
extsh[.]	1	none
isel	1	none
isync	3[2]	refetch
lbarx	2	none
lbz	2[3]	none
lbzu	2[3]	none
lbzux	2[3]	none
lbzx	2[3]	none
lha	2[3]	none
lharx	2[3]	none
lhau	2[3]	none
lhaux	2[3]	none
lhax	2[3]	none
lhbrx	2[3]	none
lhz	2[3]	none
lhzu	2[3]	none
lhzux	2[3]	none
lhzx	2[3]	none
lmw	2 +(n/2)	none
lwarx	2[3]	none
lwbrx	2[3]	none
lwz	2[3]	none
lwzu	2[3]	none
lwzux	2[3]	none
lwzx	2[3]	none
mbar	1[2]	pseudo-dispatch
mcrf	1	none
mcrxr	1	completion
mfcr	1	none
mfmsr	1	none

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

Mnemonic	Latency	Serialization
mfspr (except DEBUG)	1	none
mfspr (DEBUG)	4[2]	completion
msync	1[2]	completion
mtcrf	2	none
mtmsr	3[2]	completion
mtspr (DEBUG)	4[2]	completion
mtspr (except DEBUG, msr, hid0/1)	1	none
mulhw[.]	2	none
mulhwu[.]	2	none
mulli	2	none
mullw[o][.]	2	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	3	refetch
rfdi	3	refetch
rfi	3	refetch
rfmci	3	refetch
rlwimi[.]	1	none
rlwinm[.]	1	none
rlwnm[.]	1	none
sc	4	refetch
slw[.]	1	none
sraw[.]	1	none

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

Mnemonic	Latency	Serialization
srawi[.]	1	none
srw[.]	1	none
stb	2[3]	none
stbcx.	2	none
stbu	2[3]	none
stbux	2[3]	none
stbx	2[3]	none
sth	2[3]	none
sthbrx	2[3]	none
sthcx.	2[3]	none
sthu	2[3]	none
sthux	2[3]	none
sthx	2[3]	none
stmw	2 + (n ÷ 2)	none
stw	2[3]	none
stwbrx	2[3]	none
stwcx.	2[3]	none
stwu	2[3]	none
stwux	2[3]	none
stwx	2[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	3	completion
wrteei	3	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

4.10 Operand Placement On Performance

The placement (location and alignment) of operands in memory affects the relative performance of memory accesses, in some cases significantly. Table 4-5 indicates the effects for the e200z4 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 that 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	—	Protection Boundary
4 Byte	4 <4	Optimal Good	—	-- Good
2 Byte	2 <2	Optimal Good	—	-- Good
1 Byte	1	Optimal	—	--
lmw, stmw	4 <4	Good Poor	—	Good Poor
string	N/A	—	—	—

Note:

Optimal: One EA calculation occurs.

Good: Multiple EA calculations occur that may cause additional bus activities with multiple bus transfers.

Poor: Alignment interrupt occurs.

Chapter 5
Interrupts and Exceptions

The Power ISA embedded category architecture defines the mechanisms by which the e200 core implements interrupts and exceptions. The document uses the word ‘interrupt’ to mean the action in which the processor saves its old context and begins execution at a predetermined interrupt handler address. Exceptions are referred to as events that, when enabled, cause the processor to take an interrupt. This section uses the same terminology.

The Power ISA embedded category architecture exception mechanism allows the processor to change to supervisor state as a result of unusual conditions arising either in the execution of instructions or 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, DSRR0/DSRR1, or 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). The 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 non-critical interrupts, CSRR0 and CSRR1 for critical interrupts, DSRR0 and DSRR1 for debug interrupts (when the debug functionality is enabled), and MCSRR0 and MCSRR1 for machine check interrupts. Hardware supports the nesting of critical interrupts within non-critical interrupts, machine check interrupts within both critical and non-critical interrupts, and debug interrupts within both critical, non-critical, 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 a condition that can cause an exception is identified by the processor. This is also referred to as an exception event.

Taken — An interrupt is described as taken when control of instruction execution is passed to the interrupt handler; that is, the context is saved, 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 begins 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.

5.1 Interrupts

This section discusses interrupt classes and types.

5.1.1 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.

5.1.2 Interrupt Types

The e200z4 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 5-1 describes the interrupt types.

Table 5-1. 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 interrupt is not enabled, it is also 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 5-1. Interrupt Types

Category	Description	Programming Resources
Machine check interrupt	Provides a separate set of resources for the machine check interrupt. See Section 5.7.2, "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 5.7.16, "Debug Interrupt (IVOR15)."	Debug save and restore SPRs (DSRR0/DSRR1) and **rfdi**. Can be masked by the machine check enable bit, MSR[DE]. Includes the debug syndrome 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.

As specified by the Power ISA embedded category architecture, interrupts can be either precise or imprecise, synchronous or asynchronous, and critical or non-critical. A precise interrupt architecturally guarantees that no instruction beyond the instruction causing the exception has (visibly) executed. 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. Critical interrupts are provided with a separate save/restore register pair (CSRR0/CSRR1) to allow certain critical exceptions to be handled within a non-critical 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 non-critical or critical interrupt handler.

The types of interrupts handled are shown in Table 5-2. Refer to the interrupt chapter in the *EREF* for exact details of each interrupt type.

Table 5-2. Interrupt Classifications

Interrupt Types	Synchronous/Asynchronous	Precise/Imprecise	Critical/Non-critical/ Debug/Machine Check
System Reset	Asynchronous, non-maskable	Imprecise	—
Machine Check	—	—	Machine Check
Non-Maskable Input Interrupt	Asynchronous, non-maskable	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	Non-critical
Instruction-based Debug Interrupts	Synchronous	Precise	Critical/Debug

Table 5-2. Interrupt Classifications (Continued)

Interrupt Types	Synchronous/Asynchronous	Precise/Imprecise	Critical/Non-critical/ Debug/Machine Check
Debug Interrupt (UDE) Debug Imprecise Interrupt	Asynchronous	Imprecise	Critical/Debug
Data Storage/Alignment/TLB Interrupts Instruction Storage/TLB Interrupts	Synchronous	Precise	Non-critical

These classifications are discussed in greater detail in Section 5.7, "Interrupt Definitions." Interrupts implemented in the e200 and the exception conditions that cause them are listed in Table 5-3.

Table 5-3. Exceptions and Conditions

Interrupt Type	Interrupt Vector Offset Register	Causing Conditions
System reset	none, vector to [p_rstbase[0:29]] || 2'b00	Reset by assertion of **p_reset_b**.
Critical Input	IVOR 0[1]	**p_critint_b** is asserted and MSR_{CE} = 1.
Machine check	IVOR 1	• **p_mcp_b** transitions from negated to asserted • ISI, ITLB Error on first instruction fetch for an exception handler • Parity Error signaled on cache access • External bus error
Machine check (NMI)	IVOR 1	**p_nmi_b** transitions from negated to asserted.
Data Storage	IVOR 2	• Access control. • Byte ordering due to misaligned access across page boundary to pages with mismatched E bits • Cache locking exception
Instruction Storage	IVOR 3	• 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
External Input	IVOR 4[1]	**p_extint_b** is asserted and MSR_{EE}=1.
Alignment	IVOR 5	• **lmw**, **stmw** not word aligned • **lwarx** or **stwcx.** not word aligned, **lharx** or **sthcx.** not half-word aligned • **dcbz** with disabled cache, or to W or I storage • SPE ld and st instructions not properly aligned
Program	IVOR 6	Illegal, Privileged, Trap, FP enabled, AP enabled, Unimplemented Operation.
Floating-point unavailable	IVOR 7	MSR_{FP} = 0 and attempt to execute a Book E floating point operation

Table 5-3. Exceptions and Conditions (Continued)

Interrupt Type	Interrupt Vector Offset Register	Causing Conditions
System call	IVOR 8	Execution of the System Call (**sc, se_sc**) instruction
AP unavailable	IVOR 9	
Decrementer	IVOR 10	As specified in the *EREF*
Fixed Interval Timer	IVOR 11	As specified in the *EREF*
Watchdog Timer	IVOR 12	As specified in the *EREF*
Data TLB Error	IVOR 13	Data translation lookup did not match a valid entry in the TLB
Instruction TLB Error	IVOR 14	Instruction translation lookup did not match a valid entry in the TLB
Debug	IVOR 15	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	IVOR 16–31	—
SPE Unavailable Exception	IVOR 32	See Section 7.2.6.1, “SPE Unavailable Exception
EFP Data Exception	IVOR 33	See Section 6.2.5.2, “Embedded Floating-point Data Exception
EFP Round Exception	IVOR 34	See Section 6.2.5.3, “Embedded Floating-Point Round Exception

[1] Autovectored external and critical input interrupts use this IVOR. Vectored interrupts supply an interrupt vector offset directly.

5.2 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 5-1.

0				PIL	PPR	PTR	FP	ST	0	DLK	ILK	AP	PUO	BO	PIE	0								SPE	0	VLEMI	0			MIF	0
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

SPR - 62; Read/Write; Reset - 0x0

Figure 5-1. Exception Syndrome Register (ESR)

The ESR bits are defined in Table 5-4.

Table 5-4. ESR Bit Settings

Bit(s)	Name	Description	Associated Interrupt Type
0–3 (32–35)	—	Allocated[1]	—
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 (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[2]	—
10 (42)	DLK	Data Cache Locking[3]	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[2]	—
24 (56)	SPE	SPE APU Operation	SPE Unavailable SPE Floating-point Data Exception SPE Floating-point Round Exception Alignment Data Storage Data TLB

Table 5-4. ESR Bit Settings (Continued)

Bit(s)	Name	Description	Associated Interrupt Type
25 (57)	—	Allocated[1]	—
26 (58)	VLEMI	VLE Mode Instruction	SPE Unavailable SPE Floating-point Data Exception SPE Floating-point Round Exception Data Storage Data TLB Instruction Storage Alignment Program System Call
27–29 (59–61)	—	Allocated[1]	—
30 (62)	MIF	Misaligned Instruction Fetch	Instruction Storage Instruction TLB
31 (63)	—	Allocated[1]	—

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

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

[3] This bit is implemented, but not set by hardware

5.3 Machine State Register

The machine state register defines the state of the processor. It is shown in Figure 5-2.

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					UCLE	SPE	0						WE	CE	0	EE	PR	FP	ME	FE0	0	DE	FE1	0		IS	DS	0		RI	0

Read/ Write; Reset - 0x0

Figure 5-2. Machine State Register (MSR)

The MSR bits are defined in Table 5-5.

Table 5-5. MSR Bit Settings

Bit(s)	Name	Description
0–4 (32–36)	—	Reserved[1]
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 5-5. MSR Bit Settings (Continued)

Bit(s)	Name	Description
6 (38)	SPE	SPE Available 0 Execution of SPE and EFP APU vector instructions is disabled; SPE Unavailable exception taken instead, and SPE bit is set in ESR. 1 Execution of SPE and EFP APU vector instructions is enabled.
7–12 (39–44)	—	Reserved[1]
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[1]
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 (e.g. 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 e200, the floating point unit is not supported, and an Unimplemented Operation exception will be generated for attempted execution of 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[1]
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 5-5. MSR Bit Settings (Continued)

Bit(s)	Name	Description
24 (56)	—	Reserved[1]
25 (57)	—	Reserved[1]
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).
28–29 (60–61)	—	Reserved[1]
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[1]

[1] These bits are not implemented, will be read as zero, and writes are ignored.

5.4 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 is shown in Figure 5-3.

MCP	IC_DPERR	0		EXCP_ERR	IC_TPERR	0	IC_LKERR	0			NMI	MAV	MEA	0	IF	LD	ST	G	0								BUS_IRERR	BUS_DRERR	BUS_WRERR	0	
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

SPR - 572; Read/Clear; Reset - 0x0

Figure 5-3. Machine Check Syndrome Register (MCSR)

Table 5-6 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.

All bits in the MCSR are implemented as “write ‘1’ 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 will not clear a bit in the MCSR other than at reset. Software will typically sample MCSR early in the machine check handler, and will use the sampled value to clear those bits which 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 will cause a subsequent machine check interrupt once MSR_{ME}=1.

Table 5-6. 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
2–3 (34–35)	—	Reserved, should be cleared.	—	—
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)	—	Reserved, should be cleared.		—
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	—
8–10 (40–42)	—	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.		—

Table 5-6. Machine Check Syndrome Register (MCSR) (Continued)

Bit	Name	Description	Exception Type[1]	Recoverable
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.	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 Load or Store 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 guarded access encountered an error on the external bus.	Error Report	Precise
19–26 (51–58)	—	Reserved, should be cleared.		—
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	Async Mchk	Precise if data used
29 (61)	BUS_WRERR	Write bus error on store	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

“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 will 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 an asynchronous machine check exception. These bits will 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 will remain 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 will remain set until cleared by software writing a “1” to the bit position(s) to be cleared.

5.5 Interrupt Vector Prefix Registers (IVPR)

The interrupt vector prefix register (IVPR) 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 to form an instruction address from which execution is to begin. The format of IVPR is shown in Figure 5-4.

<table>
<tr><td colspan="16">Vector Base</td><td colspan="16">0</td></tr>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td><td>30</td><td>31</td></tr>
</table>

SPR 63; Read/Write

Figure 5-4. e200 Interrupt Vector Prefix Register (IVPR)

The IVPR fields are defined in Table 5-7.

Table 5-7. IVPR Register Fields

Bit(s)	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 64Kbyte boundary. This field provides the high-order 16 bits of the location of all interrupt handlers. The contents of the IVORxx 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[1]

[1] These bits are not implemented, will be read as zero, and writes are ignored.

5.6 Interrupt Vector Offset Registers (IVORxx)

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 IVPR to form an instruction address from which execution is to begin.

The format of an e200 IVOR is shown in Figure 5-5.

<table>
<tr><td colspan="16">0</td><td colspan="12">Vector Offset</td><td colspan="4">0</td></tr>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td><td>30</td><td>31</td></tr>
</table>

SPR 400–415, 528–530; Read/Write

Figure 5-5. e200 Interrupt Vector Offset Register (IVOR)

The IVOR fields are defined in Table 5-8.

Table 5-8. IVOR Register Fields

Bit(s)	Name	Description
0–15 (32–47)	—	Reserved[1]
16–27 (48–59)	Vector Offset	Vector Offset This field is used to provide a quadword index from the base address provided by the IVPR to locate an interrupt handler.
28–31 (60–63)	—	Reserved[1]

[1] These bits are not implemented, will be read as zero, and writes are ignored.

5.7 Interrupt Definitions

This section discusses IVOR0–IVOR32.

5.7.1 Critical Input Interrupt (IVOR0)

A critical input exception is signalled to the processor by the assertion of the critical interrupt pin (p_critint_b). When the e200 detects the exception and the exception is enabled by MSR[CE], it takes the critical input interrupt. The p_critint_b input is a level-sensitive signal expected to remain asserted until the the processor 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 5-9 lists register settings when a critical input interrupt is taken.

Table 5-9. 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 SPE0 WE0 CE0 EE0 PR0	FP0 ME— FE00 DE—/0[1]	FE10 IS 0 DS0 RI —
ESR	Unchanged		
MCSR	Unchanged		

Table 5-9. Critical Input Interrupt—Register Settings (Continued)

DEAR	Unchanged
Vector	$IVPR_{0–15}$ || $IVOR0_{16–27}$ || 4b0000 (autovectored) $IVPR_{0–15}$ || **p_voffset[0–11]** || 4b0000 (non-autovectored)

[1] DE is cleared when the Debug APU is disabled. Clearing of DE is optionally supported by control in HID0 when the Debug APU is enabled.

When the debug instructions set is enabled, the MSR[DE] bit 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 autovectored critical input interrupts to determine the interrupt handler location. The 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 autovectored (p_avec_b negated when p_critint_b asserted).

5.7.2 Machine Check Interrupt (IVOR1)

The e200z446n3 implements the machine check exception as defined in the Freescale EIS machine check instruction set except for automatic clearing of the MSR[DE] bit (see later paragraph). This behavior is different from the definition in the Power ISA embedded category architecture. The e200 initiates a machine check interrupt if any of the machine check sources listed in Table 5-3 is detected.

As defined in Freescale EIS machine check instruction set, 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. Processing of asynchronous type machine check sources (the sources reflected in the MCSR “async mchk” syndrome bits) is gated by MSR[ME].

The Freescale EIS machine check instruction set defines a separate set of save/restore registers (MCSRR0/1), 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 MCSRR0/1.

The MSR[RI] status bit is provided for software use in determining if multiple nested machine check exceptions have occurred. Software may interrogate the MCSRR1[RI] bit to determine if a machine check occurred during the initial portion of a machine check handler prior to handler code which sets MSR[RI] to ‘1’ to indicate that the handler can now tolerate another machine check condition without losing state necessary for recovery.

The MSR[DE] bit is not automatically cleared by a machine check exception, but can be configured to be cleared or left unchanged by the HID0 register (HID0[MCCLRDE]). Refer to Section 2.4.11, “Hardware Implementation Dependent Register 0 (HID0).”

5.7.2.1 Machine Check Causes

Machine check causes are divided into different types:

- Error report machine check conditions
- Non-maskable 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 e200z4 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.

5.7.2.1.1 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. The MSR[RI] and MCSRR1[RI] bits are provided to assist software in determining recoverability.

For error report machine check exceptions, the MCSR 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 asynchronous machine check condition may occur for the same error condition prior to the error report machine check, and the error report machine check may be discarded.

Depending on the type of error, the MCSR IF, LD, G, or ST bits are set by hardware to reflect the error being reported. Software is responsible for clearing these syndrome bits by writing a ‘1’ to the bits to be cleared. Hardware will not clear an error report bit once it is set.

- MCSR[IF] is set if the error occurred during an instruction fetch
- MCSR[LD] is set if the error occurred for a load instruction. If the error occurred for a guarded load and the error source was from the external bus, MCSR[G] will also be set.
- MCSR[ST] is set if the error occurred in the MMU (DTLB Error or DSI) for a store type instruction, if an external termination error was received on a cache-inhibited guarded store or on a store conditional instruction. If an external termination error occurred 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 e200z446n3, although this may not generally be the case for a multithreaded version.

Table 5-10 lists the error report machine check exceptions.

Table 5-10. 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) & $L1CSR1_{ICEA}$=’00’	IF	yes
	Icache uncorrectable tag array parity error $L1CSR1_{ICEA}$=’01’ & and locked line was invalidated		yes
	External termination error		yes
Load instruction	External termination error on load	LD, [G][2]	yes
Load and reserve instruction	External termination error on load	LD, [G][2]	yes
Store instruction	External termination error on unbuffered store[3]	ST, [G][5]	yes
	External termination error on CI+G store[4]	ST, G	yes
Store conditional instruction	External termination error on store conditional	ST, [G][5]	yes
icblc instruction	Icache tag array parity error & at least one line locked & $L1CSR1_{ICEA}$=‘00’	IF	yes
	Icache tag array uncorrectable parity error & $L1CSR1_{ICEA}$=‘01’ & locked line was invalidated		yes
icbtls instruction	Icache tag array parity error & $L1CSR1_{ICEA}$=‘00’	IF	yes
	Icache tag array uncorrectable parity error & $L1CSR1_{ICEA}$=‘01’ & locked line was invalidated		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] Store may be unbuffered if the store buffer is disabled

[4] Only reported if the store was a cache-inhibited guarded store

[5] Only reported if the store was a guarded store.

5.7.2.1.2 Non-Maskable Interrupt Machine Check Exceptions

Non-maskable interrupt exceptions are reported by means of the p_nmi_b input pin, which is transition sensitive. NMI exceptions are not gated by MSR[ME], thus they are not necessarily recoverable if an NMI exception occurs during the initial part of a machine check exception handler. The MSR[RI] and MCSRR1[RI] bits are provide to 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 ‘1’ to the bits to be cleared. Hardware will not clear an NMI bit once it is set.

The MCAR is not updated for NMI machine check exceptions.

5.7.2.1.3 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 that detected the condition. For many cases, the asynchronous condition will be 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 asynchronous machine check bit is set, the interrupt is postponed until MSR[ME] is later set to ‘1,’ 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 asynchronous machine check syndrome bit. Software is responsible for clearing these syndrome bits by writing a ‘1’ to the bit or bits to be cleared. Hardware will not clear an asynchronous machine check 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, then the MCAR and the MCSR[MEA] bit are updated and the MCSR[MAV] bit is set. If MCSR[MAV] was previously set, the MCAR and the MCSR[MEA] bit are not affected.

Table 5-11 details all asynchronous machine check sources.

Table 5-11. 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

Table 5-11. Asynchronous Machine Check Exceptions (Continued)

Asynchronous Machine Check Source	Transaction Source	Error Type	MCSR Update[1]		MCAR Update[2]
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
		L1CSR1[ICEA] = 01 and Auto-invalidation of locked line due to uncorrectable tag parity error		IC_TPERR, IC_LKERR	RA
	icblc	Tag array parity error and L1CSR1[ICEA] = 00 and at least one line locked		IC_TPERR	RA
	icbtls	Tag array parity error and L1CSR1[ICEA] = 00		IC_TPERR	RA
	icblc icbtls	L1CSR1[ICEA] = 01 and Auto-invalidation of locked line due to uncorrectable tag parity error		IC_TPERR, IC_LKERR	RA
BIU	store	Bus error on write	MAV	BUS_WRERR	RA
	load	Bus error	MAV	BUS_DRERR	RA
	icbtls CI or cache disabled Ifetch	Bus error on linefill Bus error on CI Ifetch Bus error on cache disabled Ifetch	MAV	BUS_IRERR	RA
	instruction fetch or icbtls	Bus error on locked line error recovery refill	MAV	BUS_IRERR, IC_LKERR	RA
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.

Table 5-12 details the priority of asynchronous machine check updates to the MCAR when multiple simultaneous asynchronous machine check conditions occur. Note that because a higher priority condition can occur after a lower priority condition occurs but before the machine check interrupt handler reads the

MCSR and MCAR, the interrupt handler may not necessarily see the higher priority MCAR value, even though multiple MCSR bits are set.

Table 5-12. 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	BIU	store	Bus error on write	BUS_WRERR
2	Instruction Cache	icblc icbtls instruction fetch	Uncorrectable tag array parity error & $L1CSR1_{ICEA}$=01 & locked line invalidated	IC_TPERR, IC_LKERR
3	BIU	instruction fetch	Bus error on refill of locked line with data parity error & $L1CSR1_{ICEA}$=01	BUS_IRERR, IC_LKERR
	Instruction Cache	icbtls	Tag array parity error & $L1CSR1_{ICEA}$=00	IC_TPERR
4		icblc	Tag array parity error & $L1CSR1_{ICEA}$=00 & at least one line is locked	IC_TPERR
5	BIU	load	Bus error on load	BUS_DRERR
6	BIU	icbtls CI or cache disabled Ifetch	Bus error on linefill Bus error on CI Ifetch Bus error on cache disabled Ifetch	BUS_IRERR
7	Instruction Cache	Instruction Fetch	Tag array parity error & $L1CSR1_{ICEA}$=00	IC_TPERR
8	Instruction Cache		Data array parity error & $L1CSR1_{ICEA}$=00	IC_DPERR

5.7.2.2 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 5-13.

Table 5-13. Machine Check Interrupt—Register Settings

<table>
<tr><th>Register</th><th colspan="3">Setting Description</th></tr>
<tr><td>MCSRR0</td><td colspan="3">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.</td></tr>
<tr><td>MCSRR1</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
SPE0
WE0
CE0
EE0
PR0</td><td>FP0
ME0
FE00
DE0/—[1]</td><td>FE10
IS 0
DS0
RI 0</td></tr>
<tr><td>ESR</td><td colspan="3">Unchanged</td></tr>
<tr><td>MCSR</td><td colspan="3">Updated to reflect the source(s) of a machine check. Hardware only sets appropriate bits, no previously set bits are cleared by hardware.</td></tr>
<tr><td>MCAR</td><td colspan="3">See Table 5-12</td></tr>
<tr><td>Vector</td><td colspan="3">IVPR$_{0–15}$ || IVOR1$_{16–27}$ || 4b0000</td></tr>
</table>

[1] DE is cleared when the debug functionality is disabled. Clearing of DE is optionally supported by control in HID0 when the debug functionality is enabled.

The machine check syndrome register is provided to identify the source(s) of a machine check, and in conjunction with MCSRR1[RI], may be used to identify recoverable events.

The MSR[RI] status bit is provided for software use in determining if multiple nested machine check exceptions have occurred. Software may interrogate the MCSRR1[RI] bit to determine whether a machine check occurred during the initial portion of a machine check handler prior to handler code, which sets MSR[RI] to ‘1’ 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 non-maskable 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.

5.7.2.3 Checkstop State

Machine checks no longer result in a checkstop, and there is no checkstop state implemented on e200z4 cores.

5.7.3 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
- Byte ordering
- Cache locking

Access control is defined as in the Power ISA embedded category architecture. 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 **icbtls** or **icblc** in user mode with MSR[UCLE] = 0.

Table 5-14 lists register settings when a DSI is taken.

Table 5-14. 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		
MSR	UCLE 0 SPE0 WE0 CE— EE0 PR0	FP0 ME— FE00 DE—	FE10 IS 0 DS0 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		
Vector	IVPR$_{0–15}$ || IVOR2$_{16–27}$ || 4b0000		

5.7.4 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 PowerISA embedded category architecture, 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 PowerPC VLE involve extending the definition of the instruction storage interrupt to include byte ordering exceptions for instruction accesses, and misaligned instruction fetch exceptions, and corresponding updates to the ESR.

Table 5-15 shows ISI exceptions and conditions.

Table 5-15. ISI Exceptions and Conditions

Interrupt Type	Interrupt Vector Offset Register	Causing Conditions
Instruction Storage	IVOR 3	• 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

Table 5-16 lists register settings when an ISI is taken.

Table 5-16. Instruction Storage 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 excepting instruction.</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
SPE0
WE0
CE—
EE0
PR0</td><td>FP0
ME—
FE00
DE—</td><td>FE10
IS 0
DS0
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">$\text{IVPR}_{0–15}$ || $\text{IVOR3}_{16–27}$ || 4b0000</td></tr>
</table>

5.7.5 External Input Interrupt (IVOR4)

An External Input exception is signalled 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 the 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 5-17 lists register settings when an external input interrupt is taken.

Table 5-17. External Input 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 SPE0 WE0 CE — EE 0 PR 0	FP 0 ME — FE 00 DE —	FE 10 IS 0 DS 0 RI —
ESR	Unchanged		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	IVPR$_{0–15}$ || IVOR4$_{16–27}$ || 4b0000 IVPR$_{0–15}$ || p_voffset[0:11] || 4b0000 (non-autovectored)		

IVOR4 is the vector offset register used by autovectored 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 an 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 autovectored (p_avec_b negated when p_extint_b asserted).

5.7.6 Alignment Interrupt (IVOR5)

The core implements the alignment interrupt as defined by the Power ISA. 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.
- Execution of an SPE load or store instruction which is not properly aligned.

Table 5-18 lists register settings when an alignment interrupt is taken.

Table 5-18. 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

Table 5-18. Alignment Interrupt—Register Settings (Continued)

MSR	UCLE 0 SPE 0 WE 0 CE — EE 0 PR 0	FP 0 ME — FE 00 DE —	FE 10 IS 0 DS 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$_{16–27}$ || 4b0000		

5.7.7 Program Interrupt (IVOR6)

The core implements the program interrupt as defined by the Power ISA. A program interrupt occurs when no higher priority exception exists and one or more of the following exception conditions defined in the Power ISA occur:

- Illegal Instruction
- Privileged Instruction
- Trap
- Unimplemented Operation

The core 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 core 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 which specify a SPRN value with SPRN[5] = 1 (even if the SPR is undefined).

The core invokes an 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 core invokes an unimplemented operation program exception on attempted execution of the instructions **lswi**, **lswx**, **stswi**, **stswx**, **mfapidi**, **mfdcrx**, **mtdcrx**, or on any Power ISA floating point instruction when MSR[FP] = 1. All other defined or allocated instructions that are not implemented by the core cause an illegal instruction program exception.

Table 5-19 lists register settings when a program interrupt is taken.

Table 5-19. 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 — FE 00 DE —	FE 10 IS 0 DS 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$_{0–15}$ || IVOR6$_{16–27}$ || 4b0000		

5.7.8 Floating-Point Unavailable Interrupt (IVOR7)

The floating-point unavailable exception is implemented as defined in the Power ISA. 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 5-20 lists register settings when a floating-point unavailable interrupt is taken.

Table 5-20. 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 — FE 00 DE —	FE 10 IS 0 DS 0 RI —
ESR	Unchanged		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	IVPR$_{0–15}$ || IVOR7$_{16–27}$ || 4b0000		

5.7.9 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 e200 for PowerPC VLE include modification of the system call interrupt definition to include updating the ESR.

Table 5-21 lists register settings when a system call interrupt is taken.

Table 5-21. 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 — FE 00 DE —	FE 10 IS 0 DS 0 RI —
ESR	[VLEMI] All other bits cleared.		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	IVPR$_{0–15}$ || IVOR8$_{16–27}$ || 4b0000		

5.7.10 Auxiliary Processor Unavailable Interrupt (IVOR9)

An auxiliary processor unavailable exception is defined by the Power ISA to occur when an attempt is made to execute an APU instruction which is implemented but configured as unavailable, and no higher priority exception condition exists.

The e200 does not utilize this interrupt.

5.7.11 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 5-22 lists register settings when a decrementer interrupt is taken.

Table 5-22. 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 — FE 00 DE —	FE 10 IS 0 DS 0 RI —
ESR	Unchanged		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	$IVPR_{0–15}$ || $IVOR10_{16–27}$ || 4b0000		

5.7.12 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 5-23 lists register settings when a FIT interrupt is taken.

Table 5-23. 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 — FE 00 DE —	FE 10 IS 0 DS 0 RI —
ESR	Unchanged		
MCSR	Unchanged		

Table 5-23. Fixed-Interval Timer Interrupt—Register Settings (Continued)

DEAR	Unchanged
Vector	IVPR$_{0–15}$ || IVOR11$_{16–27}$ || 4b0000

5.7.13 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 5-24 lists register settings when a watchdog timer interrupt is taken.

Table 5-24. 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 CE0 EE 0 PR 0	FP 0 ME — FE 00 DE 0/—[1]	FE 10 IS 0 DS 0 RI —
ESR	Unchanged		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	IVPR$_{0–15}$ || IVOR12$_{16–27}$ || 4b0000		

[1] DE is cleared when the debug functionality is disabled. Clearing of DE is optionally supported by control in HID0 when the debug functionality is enabled.

The MSR[DE] bit 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).”

5.7.14 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 5-25 lists register settings when a DTLB interrupt is taken.

Table 5-25. 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 — FE 00 DE —	FE1 0 IS 0 DS 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	$\text{IVPR}_{0–15}$ || $\text{IVOR13}_{16–27}$ || 4b0000		

5.7.15 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 PowerPC VLE involve extending the definition of the instruction TLB error interrupt to include updating the ESR.

Table 5-26 lists register settings when an ITLB interrupt is taken.

Table 5-26. Instruction TLB Error 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 — FE 00 DE —	FE 10 IS 0 DS 0 RI —
ESR	[MIF] All other bits cleared.		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	$\text{IVPR}_{0–15}$ || $\text{IVOR14}_{16–27}$ || 4b0000		

5.7.16 Debug Interrupt (IVOR15)

The e200 implements the debug interrupt as defined in the Power ISA embedded category architecture with the following changes:

- When the debug functionality 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 11, "Debug Support." With the debug functionality 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 non-critical interrupt handlers. In addition, the capability is provided to allow interrupts to 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 by the HID0 register (HID0[DCLREE, DCLRCE]). Refer to Section 2.4.11, "Hardware Implementation Dependent Register 0 (HID0)."

When the debug functionality is disabled, debug interrupts use the CSRR0 and CSRR1 registers to save machine state.

The following list describes the debug exception types. For additional details, refer to Section 11.2, "Software Debug Events and Exceptions."

- Instruction Address Compare (IAC)

  This 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.
- Branch Taken (BRT)

  This exception is signalled 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.
- Data Address Compare (DAC)

  This exception is signalled 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.

- IAC linked with DAC exceptions

  This results in a DAC exception only if one or more IAC conditions are also met. See Chapter 11, "Debug Support for more details.

- Trap (TRAP)

  This 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 occurs instead.

- Return (RET)

  This 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 there exists no higher priority exception that is enabled to cause an interrupt. CSRR0 (debug functionality disabled) or DSRR0 (debug functionality enabled) will be 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.

- Critical Return (CRET)

  This 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 will occur provided there exists no higher priority exception which is enabled to cause an interrupt. CSRR0 (debug functionality disabled) or DSRR0 (debug functionality 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 functionality is enabled to avoid corrupting CSRR0/1.

- Instruction Complete (ICMP)

  This exception is signalled following execution and completion of an instruction while this event is enabled.

- **mtmsr** or **mtdbcr0** causing both MSR[DE] and DBCR0[IDM] to end up set

  This enables precise debug mode, which may cause an imprecise (delayed) debug exception to be generated due to an earlier recorded event in the debug status register.

- Interrupt Taken (IRPT)

  This exception occurs when a non-critical interrupt context switch is detected. It is imprecise and unordered with respect to the program flow. Note that an IRPT Debug interrupt will only occur when detecting a non-critical interrupt on e200. The value saved in CSRR0/DSRR0 will be the address of the non-critical interrupt handler.

- Critical Interrupt Taken (CIRPT)

This 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 will only occur when detecting a critical interrupt on e200. The value saved in CSRR0/DSRR0 will be the address of the critical interrupt handler. Note that Critical Interrupt Taken debug events should not normally be enabled unless the Debug APU is enabled to avoid corruption of CSRR0/1.

- Unconditional Debug Event (UDE)

  This exception occurs when the unconditional debug event pin (p_ude) transitions to the asserted state.

- Debug Counter Debug

  These exceptions occur when enabled, and one of the debug counters decrements to zero.

- External Debug

  These 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 11, “Debug Support.”

Table 5-27 lists register settings when a debug interrupt is taken.

Table 5-27. Debug Interrupt—Register Settings

Register	Setting Description		
CSRR0/ DSRR0[1]	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.		
CSRR1/ DSRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 SPE 0 WE 0 CE —/0[2] EE —/0[2] PR 0	FP 0 ME — FE 00 DE 0	FE 10 IS 0 DS 0 RI —
DBSR[3]	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	UDE ICMP BRT IRPT CIRPT TRAP {IAC1, IAC2, IAC3, IAC4} {DAC1R, DAC1W, DAC2R, DAC2W} RET CRET {DCNT1, DCNT2} {DEVT1, DEVT2} {IDE}	

Table 5-27. Debug Interrupt—Register Settings (Continued)

ESR	Unchanged
MCSR	Unchanged
DEAR	Unchanged
Vector	$IVPR_{0:15}$ || $IVOR15_{16–27}$ || 4b0000

[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

5.7.17 System Reset Interrupt

The e200 implements the system reset interrupt as defined in the Power ISA embedded category architecture. The system reset exception is a non-maskable, asynchronous exception signalled 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 2’b00) 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. 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 5-28 shows the TSR register bits associated with watchdog timer reset status. Note that these bits will be cleared when a processor reset occurs; thus if the p_wrs[0:1] outputs are factored into p_reset_b, they will only be seen in the “00” state by software.

Table 5-28. TSR Watchdog Timer Reset Status

Bit(s)	Name	Function
2–3 (34–35)	WRS	00 No action performed by Watchdog Timer 01 Watchdog Timer second time-out caused **p_wrs[1]** to be asserted 10 Watchdog Timer second time-out caused **p_wrs[0]** to be asserted 11 Watchdog Timer second time-out caused **p_wrs[0]** and **p_wrs[1]** to be asserted

Table 5-29 shows the DBSR register bits associated with reset status.

Table 5-29. DBSR Most Recent Reset

Bit(s)	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 5-30 lists register settings when a system reset interrupt is taken.

Table 5-30. 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 FE 00 DE 0	FE 10 IS 0 DS 0 RI 0
ESR	Cleared		
DEAR	Undefined		
Vector	[p_rstbase[0:29]] || 2’b00		

5.7.18 SPE Unavailable Interrupt (IVOR32)

The SPE unavailable exception is taken if MSR[SPE] is cleared and execution of an SPE instruction other than the scalar floating-point instructions (**efs**$_{\mathbf{xxx}}$) or **brinc** is attempted, or execution of a EFPU **evfsxx** instruction is attempted. When the SPE APU Unavailable exception occurs, the processor suppresses execution of the instruction causing the exception. Table 5-31 lists register settings when a SPE unavailable interrupt is taken.

Table 5-31. SPE Unavailable Interrupt—Register Settings

Register	Setting Description		
SRR0	Set to the effective address of the excepting SPE or EFP 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 — FE 00 DE —	FE 10 IS 0 DS 0 RI —
ESR	SPE, [VLEMI]. All other bits cleared.		
MCSR	Unchanged		

Table 5-31. SPE Unavailable Interrupt—Register Settings (Continued)

DEAR	Unchanged
Vector	IVPR$_{0–15}$ || IVOR32$_{16–27}$ || 4b0000

5.7.19 EFP Floating-point Data Interrupt (IVOR33)

The EFP floating-point data interrupt is taken if no higher priority exception exists and an EFP floating-point data exception is generated. When a floating-point data exception occurs, the processor suppresses execution of the instruction causing the exception.

Table 5-32 lists register settings when an SPE floating-point data interrupt is taken.

Table 5-32. SPE Floating-point Data Interrupt—Register Settings

Register	Setting Description		
SRR0	Set to the effective address of the excepting EFP 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 — FE 00 DE —	FE 10 IS 0 DS 0 RI —
ESR	SPE, [VLEMI]. All other bits cleared.		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	IVPR$_{0–15}$ || IVOR33$_{16–27}$ || 4b0000		

5.7.20 EFP Floating-point Round Interrupt (IVOR34)

The EFP floating-point round interrupt is taken when an EFP floating-point instruction generates an inexact result and inexact exceptions are enabled.

Table 5-33 lists register settings when an EFP Floating-point Round interrupt is taken.

Table 5-33. SPE Floating-point Round Interrupt—Register Settings

Register	Setting Description		
SRR0	Set to the effective address of the instruction following the excepting EFP 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 — FE 00 DE —	FE 10 IS 0 DS 0 RI —

Table 5-33. SPE Floating-point Round Interrupt—Register Settings (Continued)

ESR	SPE, [VLEMI]. All other bits cleared.
MCSR	Unchanged
DEAR	Unchanged
Vector	IVPR$_{0–15}$ || IVOR34$_{16–27}$ || 4b0000

5.8 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. 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, non-maskable, non-recoverable:
    - 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, non-maskable, possibly non-recoverable:
    - Non-maskable 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/non-maskable, recoverable/non-recoverable:
    - 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, 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 cancelled 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 Unavailable, Data TLB, EFP Floating-point Data, EFP 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)

5.8.1 Exception Priorities

Exceptions are prioritized as described in Table 5-34. Some exceptions may be masked or imprecise which will affect their priority. Non-maskable 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 non-recoverable.

Table 5-34. e200 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	—	—	

Table 5-34. e200 Exception Priorities (Continued)

Priority	Exception	Cause	IVOR
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	—	—	
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	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, 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 Unavailable	Any SPE unavailable exception condition.	32
16	Program: Unimplemented	Attempted execution of an unimplemented instruction.	6

Table 5-34. e200 Exception Priorities (Continued)

Priority	Exception	Cause	IVOR
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
	EFP Floating-point Data	Denormalized, NaN, or Infinity data detected as input or output, or underflow, overflow, divide by zero, or invalid operation in the EFP APU.	33
	EFP Round	Inexact Result	34
19	Alignment	**lmw**, **stmw, lwarx,** or **stwcx.** not word aligned. **lharx,** or **sthcx.** not half word aligned. **dcbz**	5
20	Debug: Debug with concurrent DTLB or DSI exception: 1.DAC/IAC linked[2] 2.DAC unlinked[2]	Debug with concurrent DTLB or DSI exception. 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	1. Access control. 2. Byte ordering due to misaligned access across page boundary to pages with mismatched E bits. 3. Cache locking due to attempt to execute a **icbtls** or **icblc** in user mode with MSR[UCLE] = 0.	2
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

Table 5-34. e200 Exception Priorities (Continued)

Priority	Exception	Cause	IVOR
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 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 e200 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 points to the load or store class instruction, rather than to the next instruction.

5.9 Interrupt Processing

When an interrupt is taken, the processor uses SRR0/SRR1 for non-critical interrupts, CSRR0/CSRR1 for critical interrupts, MCSRR0/MCSRR1 for machine check interrupts, and 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 if appropriate:

- SRR1 is used to save machine state (selected MSR bits) on non-critical 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 functionality 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 5-35.

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 and an interrupt vector offset register specific for each type of interrupt (see Table 5-3).

Table 5-35 shows the MSR settings for different interrupt categories.

Table 5-35. MSR Setting Due to Interrupt

Bit(s)	MSR Definition	Reset Setting	Non-Critical 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
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
30 (62)	RI	0	—	—	—	0

Reserved and preserved bits are unimplemented and read as 0.

[1] Conditionally cleared based on control bits in HID0

5.9.1 Enabling and Disabling Exceptions

When a condition exists that may cause an exception to be generated, it must be determined whether the exception is enabled for that condition.

- 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 bits in the HID0 register.
- Asynchronous, maskable non-critical 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 non-critical 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 11, "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 the e200 instead).

5.9.2 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.*

5.10 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 6
Embedded Floating-Point Unit, Version 2

This chapter describes the instruction set architecture of the embedded floating-point unit, version 2 implemented on the e200z4. 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.

6.1 Nomenclature and Conventions

Several 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).

6.2 EFPU Programming Model

The e200z4 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 floating-point instructions operate on the lower half of GPRs. These single-precision 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 is 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.

6.2.1 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. Status and control bits are shared for vector floating-point operations, single-precision 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 6-1.

SOVH	OVH	FGH	FXH	FINVH	FDBZH	FUNFH	FOVFH	0		FINXS	FINVS	FDBZS	FUNFS	FOVFS	MODE	SOV	OV	FG	FX	FINV	FDBZ	FUNF	FOVF	0	FINXE	FINVE	FDBZE	FUNFE	FOVFE	FRMC	
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

SPR - 512; Read/Write; Reset - 0x0

Figure 6-1. SPE/EFPU Status and Control Register (SPEFSCR)

The SPEFSCR bits are defined in Table 6-1.

Table 6-1. 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 754 invalid operation (IEEE 754-1985 sec7.1) 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.
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.
8:9 (40:41)	—	Reserved

Table 6-1. SPE /EFPU Status and Control Register (Continued)

Bits	Name	Description
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 IEEE754 hardware results operating mode (not supported by Zen) 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 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.
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.

Table 6-1. SPE /EFPU Status and Control Register (Continued)

Bits	Name	Description
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 IEEE754 invalid operation (IEEE754-1985 sec7.1) 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 6-1. 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

6.2.2 GPRs and Power ISA Instructions

The e200z4 core implements the 32-bit forms of the Power ISA embedded category instructions. 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.

6.2.3 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 (**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.

6.2.4 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 the SPE APU instructions. 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 APU exceptions. Thus, SPE and embedded floating-point interrupts are indistinguishable in the ESR).

6.2.5 EFPU Exceptions

The architecture defines the following embedded floating-point category exceptions:

- SPE/EFP Unavailable exception
- EFP Floating-point Data exception
- EFP Floating-point Round exception

Three interrupt vector offset registers (IVORs)—IVOR32, IVOR33, and IVOR34—are used by the exception model. The SPR number for IVOR32 is 528, for IVOR33 it is 529, and for IVOR34 it is 530. These registers are privileged.

6.2.5.1 EFP Unavailable Exception

The EFP Unavailable exception is taken if MSR[SPE] is cleared and execution of an EFPU vector instruction (**evfs**$_{xxx}$) is attempted. When the EFP 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.
- The ESR[SPE]bit is set. All other ESR bits are cleared.

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

6.2.5.2 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.
- The ESR[SPE] bit 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.

6.2.5.3 Embedded Floating-Point Round Exception

The embedded floating-point round exception occurs if the SPEFSCR[FINXE] bit is set, no floating-point data exception is taken, and either the unrounded result of an operation is not exact, an overflow occurs and overflow exceptions are disabled (FOVF or FOVFH set with FOVFE cleared), or an underflow occurs and underflow exceptions are disabled (FUNF set with FUNFE cleared). The embedded floating-point round exception will 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 will be updated with the correctly rounded result. The FG and FX bits corresponding to the other exact element will both be ‘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.
- The ESR[SPE] bit is set. All other ESR bits are cleared.
- SPEFSCR[FGH, FG, FXH, FX] are set appropriately. SPEFSCR[FINXS] will be set.

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

6.2.6 Exception Priorities

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

1. EFP unavailable
2. EFP floating-point data
3. EFP floating-point round

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 embedded floating-point data exception.

6.3 Embedded Floating-Point Unit Operations

The e200z4 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 PowerPC ISA floating-point instructions are not implemented in the e200z4.

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 Std. 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 e200z4 hardware implements mode 0. Operation that conforms to IEEE Std. 754 standard is still available in mode 0 with assistance of a software envelope.

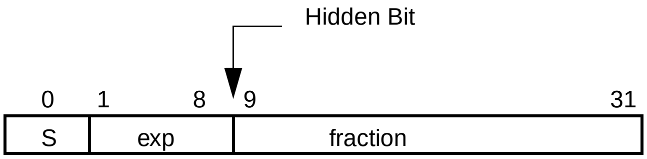
6.3.1 Floating-Point Data Formats

The EFPU supports single precision scalar and 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.

6.3.1.1 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 the IEEE 754 specification, single-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 6-2.



S—sign bit; 0—positive; 1—negative

exp—biased exponent field (excess 127 notation)

fraction—fractional portion of number

Figure 6-2. Single Precision Data Format

For normalized numbers, the biased exponent value '*e*' lies in the range of 1 to 254 and corresponds to an actual exponent value E in the range –126 to +127. The hidden bit is a '1,' and the value of the number is interpreted as:

$$(-1)^{S} \times 2^{E} \times (1.\text{fraction})$$

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 0x7F7F_FFFF, which is approximately 3.4E+38 (2^{128}), and the minimum positive normalized value (*pmin*) is represented by the encoding 0x0080_0000, which is approximately 1.2E-38 (2^{-126}).

Two specific values of the biased exponent, 0 and 255, are reserved for encoding special values of

$\pm 0,\ \pm\infty,\ \text{NaN},\ \text{and Denorm}$.

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 Std. 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 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.

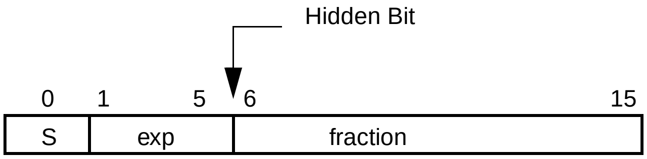
The EFPU hardware will not produce +Inf, –Inf, NaN, or a denormalized number. If the results of an instruction overflow and floating-point overflow exceptions are disabled (SPEFSCR[FOVFE] bit is cleared), then *pmax* or *nmax* is generated as the result of that instruction depending upon the sign of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

6.3.1.2 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 6-3.



S—sign bit; 0—positive; 1—negative

exp—biased exponent field (excess 15 notation)

fraction—fractional portion of number

Figure 6-3. Single 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' (for normalized numbers), and the value of the number is interpreted as:

$$(-1)^{S} \times 2^{E} \times (1.\text{fraction})$$

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, 0 and 31, are reserved for encoding special values of $\pm 0,\ \pm\infty,\ \mathrm{NaN},\ \text{and Denorm}$.

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

The EFPU hardware will not produce +Inf, –Inf, 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] bit 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] bit 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] bit is cleared), the conversion results in a properly signed max norm or zero default result.

6.3.2 Conformity to IEEE Std. 754 Standard

The Freescale EIS architecture specifies that the EFPU implements a single-precision floating-point system as defined in ANSI/IEEE Standard 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 +Infinity, –Infinity, Denormalized, NaN, or when the result of an operation produces an overflow or an underflow, an exception may be taken and the exception handler is responsible for delivering behavior that conforms to IEEE Std. 754 standard 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, to allow a software handler to conform to IEEE Std. 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 IEEE Std. 754 standard. All of these exceptions may be disabled, and the hardware will then deliver an appropriate default result.

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

6.3.3 Floating-Point Exceptions

See Section 6.2.5, “EFPU Exceptions.”

6.3.4 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	result value

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, the SPEFSCR[FINV] bit is set and FG and FX are cleared. FGH and FXH are cleared as well. I
- 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 **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 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). If RB is NaN or infinity, the result is either *pmax* (sb==0), or *nmax* (sb==1). If an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in RD. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or –0 (for rounding mode RM) is stored in RD.

Exceptions:

If any of the following exceptions are taken, the destination register is not updated.

- If the contents of RA or RB are Infinity, Denorm, or NaN, the SPEFSCR[FINV] bit is set. If SPEFSCR[FINVE] is set, an exception is taken.
- If an overflow occurs, the SPEFSCR[FOVF] bit is set. If an underflow occurs, the SPEFSCR[FUNF] bit is set. If either underflow or overflow exceptions are enabled and the corresponding bit is set, an exception is taken.

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 in the following cases.

- 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] bit will be set.
- If the floating-point inexact exception is enabled, an exception is taken using the floating-point round exception vector.

FGH, FXH, FG and FX will be 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 ¨ rB48:63

if (fexp = 0) & (ffrac = 0)) then
   result ¨ fsign || 310   // signed zero value
else if Isa16NaNorInfinity(f) then
   SPEFSCRFINV ¨ 1
   result ¨ fsign || 0b11111110 || 231   // max value
else if Isa16Denorm(f) then
   SPEFSCRFINV ¨ 1
   result ¨ fsign || 310
else
   resultsign ¨ fsign
   resultexp ¨ fexp - 15 + 127
   resultfrac ¨ ffrac || 130

rD32:63 = 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 because this conversion is always exact.

Exceptions:

SPEFSCR[FINV] is set if the source element of rB is Infinity, Denorm, or NaN. 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 **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 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 **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 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, 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, the SPEFSCR[FINV] bit 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.

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, 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, the SPEFSCR[FINV] bit 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.

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 = $RB[32:6]_3$
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, 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, the SPEFSCR[FINV] bit 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.

efscth efscth

Convert Floating-Point Single-Precision to Half-Precision

efscth **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 1 0 1

```
FP32format f;
FP16format result;

f ¨ rB[32:63]

if (fexp = 0) & (ffrac = 0)) then
    result ¨ fsign || 150   // signed zero value
else if Isa32NaNorInfinity(f) then
    SPEFSCRFINV ¨ 1
    result ¨ fsign || 0b11110 || 101   // max value
else if Isa32Denorm(f) then
    SPEFSCRFINV ¨ 1
    result ¨ fsign || 150
else
    unbias ¨ fexp - 127
    if unbias > 15 then
        result ¨ fsign || 0b11110 || 101    // max value
        SPEFSCRFOVF ¨ 1
    else if unbias < -14 && (result would not round up to bmin) then
        result ¨ fsign || 150   // like-signed zero value
        SPEFSCRFUNF ¨ 1
    else
        resultsign ¨ fsign
        resultexp ¨ unbias + 15
        resultfrac ¨ ffrac[0:9]
        guard ¨ ffrac[10]
        sticky ¨ (ffrac[11:22] ¼ 0)
        result ¨ Round16(result, LOWER, guard, sticky)
        SPEFSCRFG ¨ guard
        SPEFSCRFX ¨ sticky
        if guard | sticky then
            SPEFSCRFINXS ¨ 1

rD[32:63] = 160 || 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 any of the following interrupts are taken, the destination register is not updated:

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

In the following cases, 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.

- If the result of this instruction is inexact and no other interrupt is taken, SPEFSCR[FINXS] is set.
- 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.

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 **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 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 Na, or if an overflow occurs, the SPEFSCR[FINV] bit 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 **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	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, then the SPEFSCR[FINV] bit 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 **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 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, then the SPEFSCR[FINV] bit 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 efsctuf

Convert Floating-Point Single-Precision to Unsigned Fraction

efsctuf **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 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, then the SPEFSCR[FINV] bit 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 **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 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, then the SPEFSCR[FINV] bit 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 **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 1 0 0 0

```
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
```

Description:

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, then the SPEFSCR[FINV] bit 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 **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	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.
- 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).
- If an overflow occurs, then *pmax* or *nmax* (as appropriate) is stored in RD.
- If an underflow occurs, +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] bit 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, the SPEFSCR[FDBZ] bit is set.
- If floating-point divide by zero exceptions are enabled, an exception is taken.
- If an overflow occurs, then the SPEFSCRFOVF bit is set.
- If an underflow occurs, then the SPEFSCR[FUNF] bit 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.

In the following cases, 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.

- If the result of this instruction is inexact the SPEFSCR[FINXS$_{]}$ bit will be set
- If an overflow occurs but overflow exceptions are disabled and no other exception is taken, the SPEFSCR[FINXS$_{]}$ bit will be set.

If the floating-point inexact exception is enabled, an exception is taken using the floating-point round exception vector.

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

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

Description:

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, the following occurs:

- If RA or RB are either NaN or infinity and the intermediate product is either *pmax* (sa==sb) or *nmax* (sa!=sb), 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).
- If an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in RD.
- If an underflow occurs, +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, the SPEFSCR[FINV] bit is set. If SPEFSCR[FINVE] is set, an exception is taken and the destination register is not updated. Otherwise,

- If an overflow occurs, the SPEFSCR[FOVF] bit is set.
- If an underflow occurs, the SPEFSCR[FUNF] bit 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.

In the following cases, 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.

- If the result of this instruction is inexact, the SPEFSCR[FINXS] bit is set.
- If an overflow occurs on the add, but overflow exceptions are disabled and no other exception is taken, the SPEFSCR[FINXS] bit is set.

If the floating-point inexact exception is enabled, an exception is taken using the floating-point round exception vector.

FGH, FXH, FG, and FX are 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
```

Description:

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, SPEFSC[RFINV] 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*. 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
```

Description:

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

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

Description:

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, the SPEFSCR[FINV] bit is set. If SPEFSCR[FINVE] is set, an exception is taken, and the destination register is not updated. Otherwise,
- If an overflow occurs, the SPEFSCR[FOVF] bit is set.
- If an underflow occurs, the SPEFSCR[FUNF] bit 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] bit will be 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.

efsmul efsmul

Floating-Point Single-Precision Multiply

efsmul **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	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.
- If RA or RB are either NaN or infinity, the result is either *pmax* (sa==sb), or *nmax* (sa!=sb).
- 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, the SPEFSCR[FINV] bit is set. If SPEFSCR[FINVE] is set, an exception is taken and the destination register is not updated. Otherwise, the following occurs:

- If an overflow occurs, the SPEFSCR[FOVF] bit is set.
- If an underflow occurs, the SPEFSCR[FUNF] bit 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] bit will be 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, the SPEFSCR[FINV] bit 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.

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

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

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, the SPEFSCR[FINV] bit 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.

efsnmadd efsnmadd

Floating-Point Single-Precision Negative Multiply-Add

efsnmadd **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	0

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

Description:

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.
- 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).
- If an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in RD.
- If an underflow occurs, –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, the SPEFSCR[FINV] bit is set. If SPEFSCR[FINVE] is set, an exception is taken and the destination register is not updated. Otherwise, the following occurs:

- If an overflow occurs, the SPEFSCR[FOVF] bit is set.
- If an underflow occurs, the SPEFSCR[FUNF] bit 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] bit 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

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

Description:

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.
- 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).
- 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, the SPEFSCR[FINV] bit is set. If SPEFSCR[FINVE] is set, an exception is taken and the destination register is not updated.

If any of the following exceptions are taken, the destination register is not updated.

- If an overflow occurs, the SPEFSCR[FOVF] bit is set.
- If an underflow occurs, the SPEFSCR[FUNF] bit is set.
- If either underflow or overflow exceptions are enabled and the corresponding bit is set, an exception is taken.

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] bit will be 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.

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 are 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*.
- If the low element of rA is non-zero and has a negative sign, including –NaN or –infinity, the corresponding result is –0.
- 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).
- If RB is NaN or infinity, the result is either *nmax* (sb==0), or *pmax* (sb==1).
- 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, the SPEFSCR[FINV] bit is set. If SPEFSCR[FINVE] is set, an exception is taken and the destination register is not updated. If an underflow occurs, the SPEFSCR[FUNF] bit 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] bit 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.

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

```
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
```

Description:

The low element of RA is compared against the low element of RB. If RA is equal to RB, 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.

Exceptions:

No exceptions are generated during the execution of **efststeq** instruction. If strict conformity to the IEEE Std. 754 standard is required, the program should use the **efscmpeq** instruction.

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

```
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
```

Description:

The low element of RA is compared against the low element of RB. If RA is greater than RB, 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.

Exception:

No exceptions are generated during the execution of **efststgt** instruction. If strict conformity to IEEE Std. 754 standard is required, the program should use the **efscmpgt** instruction.

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

```
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
```

Description:

The low element of RA is compared against the low element of RB. If RA is less than RB, 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.

Exceptions:

No exceptions are generated during the execution of **efststlt** instruction. If strict conformity to IEEE Std. 754 standard is required, the program should use the **efscmplt** instruction.

NOTE:

In an implementation, the execution of **efststlt** is likely to be faster than the execution of **efscmplt** instruction.

6.3.5 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,
b	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, the SPEFSCR[FINV, FINVH] bits are set appropriately, and the SPEFSCR[FGH, FXH, FG, FX] bits 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 **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 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).
- If RB is NaN or infinity, the result is either *pmax* (sb==0) or *nmax* (sb==1).
- If an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in RD.
- If an underflow occurs, +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, the SPEFSCR[FINV, FINVH] bits are set appropriately, and the SPEFSCR[FGH, FXH, FG, FX] bits are cleared appropriately. If SPEFSCR[FINVE] is set, an exception is taken and the destination register is not updated. Otherwise, if an overflow occurs, the SPEFSCR[FOVF, FOVFH] bits are set appropriately, or if an underflow occurs, the SPEFSCR[FUNF, FUNFH] bits 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, the SPEFSCR[FINXS] bit will be 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 results. 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

$rD_{0:31} \leftarrow rA_{0:31} +_{sp} rB_{0:31}$
$rD_{32:63} \leftarrow rA_{32:63} -_{sp} rB_{32:63}$

Description:

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).
- If an element of rB is NaN or infinity, the corresponding result is either *pmax* (sb==0) or *nmax* (sb==1).
- 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.

If an overflow occurs, SPEFSCR[FOVF, FOVFH] are set appropriately. 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).

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

$rD_{0:31} \leftarrow rA_{32:63} +_{sp} rB_{0:31}$
$rD_{32:63} \leftarrow rA_{0:31} -_{sp} rB_{32:63}$

Description:

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).
- If an element of rB is NaN or infinity, the corresponding result is either *pmax* (sb==0) or *nmax* (sb==1).
- 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. 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.

SPEFSCR[FINXS, FINXSH] is set 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. 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

$rD_{0:31} \leftarrow rA_{32:63} +_{sp} rB_{0:31}$
$rD_{32:63} \leftarrow rA_{0:31} +_{sp} rB_{32:63}$

Description:

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).
- If an element of rB is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate).
- 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. 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.

SPEFSCR[FINXS, FINXSH] is set 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. 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

Vector Convert Floating-Point Single-Precision from Half-Precision

evfscfh **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 0 0 1 0 0 0 1

```
FP16format f;
FP32format result;

$fh \leftarrow rB_{24:31}$
$fl \leftarrow rB_{48:63}$

if ($fh_{exp}$ = 0) & ($fh_{frac}$ = 0)) then
    $resulth \leftarrow fh_{sign} \,||\, {}^{31}0$   // signed zero value
else if Isa16NaNorInfinity(fh) then
    $SPEFSCR_{FINVH} \leftarrow 1$
    $resulth \leftarrow fh_{sign} \,||\, 0b11111110 \,||\, {}^{23}1$   // max value
else if Isa16Denorm(fh) then
    $SPEFSCR_{FINVH} \leftarrow 1$
    $resulth \leftarrow fh_{sign} \,||\, {}^{31}0$
else
    $resulth_{sign} \leftarrow fh_{sign}$
    $resulth_{exp} \leftarrow fh_{exp} - 15 + 127$
    $resulth_{frac} \leftarrow fh_{frac} \,||\, {}^{13}0$

if ($fl_{exp}$ = 0) & ($fl_{frac}$ = 0)) then
    $resultl \leftarrow fl_{sign} \,||\, {}^{31}0$   // signed zero value
else if Isa16NaNorInfinity(fl) then
    $SPEFSCR_{FINV} \leftarrow 1$
    $resultl \leftarrow fl_{sign} \,||\, 0b11111110 \,||\, {}^{23}1$   // max value
else if Isa16Denorm(fl) then
    $SPEFSCR_{FINV} \leftarrow 1$
    $resultl \leftarrow fl_{sign} \,||\, {}^{31}0$
else
    $resultl_{sign} \leftarrow fl_{sign}$
    $resultl_{exp} \leftarrow fl_{exp} - 15 + 127$
    $resultl_{frac} \leftarrow fl_{frac} \,||\, {}^{13}0$

$rD_{0:31}$ = resulth; $rD_{32:63}$ = resultl
```

Description:

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, the SPEFSCR[FINV, FINVH] bits are set appropriately and the SPEFSCR[FGH, FXH, FG, 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.

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

$RD_{0:31}$ = CnvtSF32ToFP32($RB_{0:31}$)
$RD_{32:63}$ = CnvtSF32ToFP32($RB_{32:63}$)

Description:

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

$RD_{0:31}$ = CnvtSI32ToFP32($RB_{0:31}$)
$RD_{32:63}$ = CnvtSI32ToFP32($RB_{32:63}$)

Description:

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($RB_{32:63}$)

Description:

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

$RD_{0:31}$ = CnvtUI32ToFP32($RB_{0:31}$)
$RD_{32:63}$ = CnvtUI32ToFP32($RB_{32:63}$)

Description:

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

```
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)
```

Description:

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, the SPEFSCR[FINV, FINVH] bits are set appropriately, and the SPEFSCR[FGH, FXH, FG, FX] bits are cleared appropriately. 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.

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

```
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)
```

Description:

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, the SPEFSCR[FINV, FINVH] bits are set appropriately and the SPEFSCR[FGH, FXH, FG, FX] bits are cleared appropriately. 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.

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, the SPEFSCR[FINV, FINVH] bits are set appropriately, and the SPEFSCR[FGH, FXH, FG, FX] bits 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

Vector Convert Floating-Point Single-Precision to Half-Precision

evfscth **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 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 || $^{10}1$    // 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
```

$rD_{0:31} = {}^{16}0 \,||\, \text{resulth}; \; rD_{32:63} = {}^{16}0 \,||\, \text{resultl}$

Description:

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.

If an overflow occurs, SPEFSCR[FOVF, FOVFH] are set appropriately. 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.

SPEFSCR[FINXS, FINXSH] is set 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. 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 **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 1

```
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
```

Description:

Each single-precision floating-point element in RB is converted to a signed fraction using the current rounding mode; 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, the SPEFSCR[FINV, FINVH] bits are set appropriately and the SPEFSCR[FGH, FXH, FG, FX] bits 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, the SPEFSCR[FINXS] bit will be 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 **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 0 1

```
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
```

Description:

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, then the SPEFSCR[FINV, FINVH] bits are set appropriately, and the SPEFSCR[FGH, FXH, FG, FX] bits 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, the SPEFSCR[FINXS] bit will be 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 **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 1 0 1 0

```
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
```

Description:

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

```
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
```

Description:

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 **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 0 0

```
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
```

Description:

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] bit will be 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 **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 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] 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.

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. 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).
- If the low order element of rA or rB is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate).
- 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, $SPEFSCR_{FINXS,FINXSH}$ 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 **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 1 0 0 1

$RD_{0:31} = RA_{0:31} \div_{sp} RB_{0:31}$
$RD_{32:63} = RA_{32:63} \div_{sp} RB_{32:63}$

Description:

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.
- 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).
- If an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in RD.
- If an underflow occurs, +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, 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.

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. If an overflow occurs, then SPEFSCR[FOVF, FOVFH] are set appropriately, 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, the SPEFSCR[FINXS] bit will be 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 **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 0

$RD_{0:31} = ((RA_{0:31}\ X_{fp}\ RB_{0:31})\ +_{sp}\ RD_{0:31})$
$RD_{32:63} = ((RA_{32:63}\ X_{fp}\ RB_{32:63})\ +_{sp}\ RD_{32:63})$

Description:

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.
- 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).
- If an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in RD.
- If an underflow occurs, +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, then the SPEFSCR[FOVF, FOVFH] bits are set appropriately, or if an underflow occurs, then the SPEFSCR[FUNF, FUNFH] bits 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.

SPEFSCR[FINXS] is set if either result element of this instruction is inexact, 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. 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
```

Description:

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*.
- 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*.
- If the selected element is –NaN or –infinity, the corresponding result is *nmax*.

evfsmsub evfsmsub

Vector Floating-Point Single-Precision Multiply-Subtract

evfsmsub 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	1

$RD_{0:31} = ((RA_{0:31} \; X_{fp} \; RB_{0:31}) -_{sp} RD_{0:31})$
$RD_{32:63} = ((RA_{32:63} \; X_{fp} \; RB_{32:63}) -_{sp} RD_{32:63})$

Description:

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.
- 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).
- If an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in RD.
- If an underflow occurs, +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.

If an overflow occurs, SPEFSCR[FOVF, FOVFH] are set appropriately. 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.

SPEFSCR[FINXS] is set if either result element of this instruction is inexact, 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. 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

$RD_{0:31} = RA_{0:31}\ X_{sp}\ RB_{0:31}$
$RD_{32:63} = RA_{32:63}\ X_{sp}\ RB_{32:63}$

Description:

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.
- If RA or RB are either NaN or infinity, the result is either *pmax* (sa==sb), or *nmax* (sa!=sb).
- If an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in RD.
- If an underflow occurs, +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, then SPEFSCR[FOVF, FOVFH] are set appropriately. 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, the SPEFSCR[FINXS] bit will be 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

$rD_{0:31} \leftarrow rA_{0:31} \times_{sp} rB_{0:31}$
$rD_{32:63} \leftarrow rA_{0:31} \times_{sp} rB_{32:63}$

Description:

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.
- 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}!=b_{sign}).
- 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. 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.

SPEFSCR[FINXS] is set 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. 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

$rD_{0:31} \leftarrow rA_{32:63} \times_{sp} rB_{0:31}$
$rD_{32:63} \leftarrow rA_{32:63} \times_{sp} rB_{32:63}$

Description:

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.
- 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}!=b_{sign}).
- 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. 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.

SPEFSCR[FINXS] is set 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. 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

$rD_{0:31} \leftarrow rA_{32:63} \times_{sp} rB_{0:31}$
$rD_{32:63} \leftarrow rA_{0:31} \times_{sp} rB_{32:63}$

Description:

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.
- 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}!=b_{sign}).
- 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.

If an overflow occurs, SPEFSCR[FOVF, FOVFH] are set appropriately. 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.

SPEFSCR[FINXS] is set 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. 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, 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

$RD_{0:31} = \neg RA_{0} \parallel RA_{1:31}$
$RD_{32:63} = \neg RA_{32} \parallel RA_{33:63}$

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, an exception is taken and the destination register is not updated.

evfsnmadd evfsnmadd

Vector Floating-Point Single-Precision Negative Multiply-Add

evfsnmadd **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 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.
- 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).
- If an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in RD.
- If an underflow occurs, –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. 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.

SPEFSCR[FINXS] is set if either result element of this instruction is inexact, 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. 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 **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 1 0 1 1

$RD_{0:31} = -((RA_{0:31}\ X_{fp}\ RB_{0:31})\ -_{sp}\ RD_{0:31})$
$RD_{32:63} = -((RA_{32:63}\ X_{fp}\ RB_{32:63}) -_{sp}\ RD_{32:63})$

Description:

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.
- 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).
- If an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in RD.
- If an underflow occurs, –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] bits 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. 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.

SPEFSCR[FINXS] is set if either result element of this instruction is inexact, 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. 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

$rD_{0:31} \leftarrow SQRT(rA_{0:31})$
$rD_{32:63} \leftarrow SQRT(rA_{32:63})$

Description:

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*.
- If an element of rA is non-zero and has a negative sign, including -NaN or -infinity, the corresponding result is –0.
- 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.

SPEFSCR[FINXS, FINXSH] is set if either result element of this instruction is inexact or underflows but underflow exceptions are disabled and no other interrupt is taken. 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 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 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).
- If RB is NaN or infinity, the result is either *nmax* (sb==0), or *pmax* (sb==1).
- If an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in RD.
- If an underflow occurs, +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 SPEFSCRpFGH, 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. 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.

SPEFSCR[FINXS] is set if either result element of this instruction is inexact, 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. 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).

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

$rD_{0:31} \leftarrow rA_{0:31} -_{sp} rB_{0:31}$
$rD_{32:63} \leftarrow rA_{32:63} +_{sp} rB_{32:63}$

Description:

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).
- If an element of rB is NaN or infinity, the corresponding result is either *nmax* or *pmax* (as appropriate).
- 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. 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.

SPEFSCR[FINXS] is set 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. 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

$rD_{0:31} \leftarrow rA_{32:63} -_{sp} rB_{0:31}$
$rD_{32:63} \leftarrow rA_{0:31} +_{sp} rB_{32:63}$

Description:

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).
- If an element of rB is NaN or infinity, the corresponding result is either *nmax* or *pmax* (as appropriate).
- 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. 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.

SPEFSCR[FINXS] is set 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. 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).
- If an element of rB is NaN or infinity, the corresponding result is either *nmax* or *pmax* (as appropriate).
- 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. 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.

SPEFSCR[FINXS] is set 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. 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).
- If the low order element of rA or rB is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate).
- 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. 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).

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).
- If the low order element of rA or rB is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate).
- 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, SPEFSC[RFOVF, FOVFH] are set appropriately. 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.

SPEFSCR[FINXS, FINXSH] is set 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. 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

```
$ah = RA_{0:31}$
$al = RA_{32:63}$
$bh = RB_{0:31}$
$bl = RB_{32:63}$
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)
```

Description:

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.

Exceptions:

No exceptions are taken during the execution of **evfststeq**. If strict conformity to IEEE Std. 754 standard is required, the program should use **evfscmpeq**.

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

```
$ah = RA_{0:31}$
$al = RA_{32:63}$
$bh = RB_{0:31}$
$bl = RB_{32:63}$
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)
```

Description:

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.

Exceptions:

No exceptions are taken during the execution of **evfststgt**. If strict conformity to IEEE Std. 754 standard is required, the program should use **evfscmpgt**.

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

```
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)
```

Description:

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.

Exceptions:

No exceptions are taken during the execution of **evfststlt**. If strict conformity to IEEE Std. 754 standard is required, the program should use **evfscmplt**.

NOTE

In an implementation, the execution of **evfststlt** is likely to be faster than the execution of **evfscmplt**.

6.4 Embedded Floating-point Results Summary

The following table summarizes the results of floating-point operations on various combinations of input operands. Flag settings are performed on appropriate element flags.

Table 6-2. Floating-point Results Summary—Add, Sub, Mul, Div

Operand A	Operand B	Result	F INV	FOVF	FUNF	FDBZ	F INX
Add							
∞	∞	amax	1	0	0	0	0
∞	NaN	amax	1	0	0	0	0
∞	Denorm	amax	1	0	0	0	0
∞	Zero	amax	1	0	0	0	0

Table 6-2. Floating-point Results Summary—Add, Sub, Mul, Div (Continued)

Operand A	Operand B	Result	F INV	FOVF	FUNF	FDBZ	F INX
∞	Norm	amax	1	0	0	0	0
NaN	∞	amax	1	0	0	0	0
NaN	NaN	amax	1	0	0	0	0
NaN	Denorm	amax	1	0	0	0	0
NaN	Zero	amax	1	0	0	0	0
NaN	Norm	amax	1	0	0	0	0
Denorm	∞	bmax	1	0	0	0	0
Denorm	NaN	bmax	1	0	0	0	0
Denorm	Denorm	Zero[1]	1	0	0	0	0
Denorm	Zero	Zero[1]	1	0	0	0	0
Denorm	Norm	operand_b	1	0	0	0	0
Zero	∞	bmax	1	0	0	0	0
Zero	NaN	bmax	1	0	0	0	0
Zero	Denorm	Zero[1]	1	0	0	0	0
Zero	Zero	Zero[1]	0	0	0	0	0
Zero	Norm	operand_b	0	0	0	0	0
Norm	∞	bmax	1	0	0	0	0
Norm	NaN	bmax	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	*
Subtract							
∞	∞	amax	1	0	0	0	0
∞	NaN	amax	1	0	0	0	0
∞	Denorm	amax	1	0	0	0	0
∞	Zero	amax	1	0	0	0	0
∞	Norm	amax	1	0	0	0	0
NaN	∞	amax	1	0	0	0	0
NaN	NaN	amax	1	0	0	0	0
NaN	Denorm	amax	1	0	0	0	0
NaN	Zero	amax	1	0	0	0	0
NaN	Norm	amax	1	0	0	0	0

Table 6-2. Floating-point Results Summary—Add, Sub, Mul, Div (Continued)

Operand A	Operand B	Result	F INV	FOVF	FUNF	FDBZ	F INX
Denorm	∞	—bmax	1	0	0	0	0
Denorm	NaN	—bmax	1	0	0	0	0
Denorm	Denorm	Zero[2]	1	0	0	0	0
Denorm	Zero	Zero[2]	1	0	0	0	0
Denorm	Norm	—operand_b	1	0	0	0	0
Zero	∞	—bmax	1	0	0	0	0
Zero	NaN	—bmax	1	0	0	0	0
Zero	Denorm	Zero[2]	1	0	0	0	0
Zero	Zero	Zero[2]	0	0	0	0	0
Zero	Norm	—operand_b	0	0	0	0	0
Norm	∞	—bmax	1	0	0	0	0
Norm	NaN	—bmax	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	*
Multiply[3]							
∞	∞	max	1	0	0	0	0
∞	NaN	max	1	0	0	0	0
∞	Denorm	Zero	1	0	0	0	0
∞	Zero	Zero	1	0	0	0	0
∞	Norm	max	1	0	0	0	0
NaN	∞	max	1	0	0	0	0
NaN	NaN	max	1	0	0	0	0
NaN	Denorm	Zero	1	0	0	0	0
NaN	Zero	Zero	1	0	0	0	0
NaN	Norm	max	1	0	0	0	0
Denorm	∞	Zero	1	0	0	0	0
Denorm	NaN	Zero	1	0	0	0	0
Denorm	Denorm	Zero	1	0	0	0	0
Denorm	Zero	Zero	1	0	0	0	0
Denorm	Norm	Zero	1	0	0	0	0
Zero	∞	Zero	1	0	0	0	0

Table 6-2. Floating-point Results Summary—Add, Sub, Mul, Div (Continued)

Operand A	Operand B	Result	F INV	FOVF	FUNF	FDBZ	F INX
Zero	NaN	Zero	1	0	0	0	0
Zero	Denorm	Zero	1	0	0	0	0
Zero	Zero	Zero	0	0	0	0	0
Zero	Norm	Zero	0	0	0	0	0
Norm	∞	max	1	0	0	0	0
Norm	NaN	max	1	0	0	0	0
Norm	Denorm	Zero	1	0	0	0	0
Norm	Zero	Zero	0	0	0	0	0
Norm	Norm	_Calc_	0	*	*	0	*
Divide[3]							
∞	∞	Zero	1	0	0	0	0
∞	NaN	Zero	1	0	0	0	0
∞	Denorm	max	1	0	0	0	0
∞	Zero	max	1	0	0	0	0
∞	Norm	max	1	0	0	0	0
NaN	∞	Zero	1	0	0	0	0
NaN	NaN	Zero	1	0	0	0	0
NaN	Denorm	max	1	0	0	0	0
NaN	Zero	max	1	0	0	0	0
NaN	Norm	max	1	0	0	0	0
Denorm	∞	Zero	1	0	0	0	0
Denorm	NaN	Zero	1	0	0	0	0
Denorm	Denorm	max	1	0	0	0	0
Denorm	Zero	max	1	0	0	0	0
Denorm	Norm	Zero	1	0	0	0	0
Zero	∞	Zero	1	0	0	0	0
Zero	NaN	Zero	1	0	0	0	0
Zero	Denorm	max	1	0	0	0	0
Zero	Zero	max	1	0	0	0	0
Zero	Norm	Zero	0	0	0	0	0
Norm	∞	Zero	1	0	0	0	0
Norm	NaN	Zero	1	0	0	0	0

Table 6-2. Floating-point Results Summary—Add, Sub, Mul, Div (Continued)

Operand A	Operand B	Result	F INV	FOVF	FUNF	FDBZ	F INX
Norm	Denorm	max	1	0	0	0	0
Norm	Zero	max	0	0	0	1	0
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 6-3. Floating-point Results Summary—madd, msub, nmadd, nmsub

Operand A	Operand B	Operand D	Result	F INV	FOVF	FUNF	FDBZ	F INX
			madd					
∞ , NaN	∞ , NaN, Norm	∞ , NaN, denorm, zero, Norm	abmax	1	0	0	0	0
∞ , NaN	denorm, zero	∞ , NaN	dmax	1	0	0	0	0
∞ , NaN	denorm, zero	denorm, zero	zero[1]	1	0	0	0	0
∞ , NaN	denorm, zero	Norm	operand_d	1	0	0	0	0
denorm	∞ , NaN, denorm, zero, Norm	∞ , NaN	dmax	1	0	0	0	0
denorm	∞ , NaN, denorm, zero, Norm	denorm, zero	zero[1]	1	0	0	0	0
denorm	∞ , NaN, denorm, zero, Norm	Norm	operand_d	1	0	0	0	0
zero	∞ , NaN, denorm,	∞ , NaN	dmax	1	0	0	0	0
zero	∞ , NaN, denorm	denorm, zero	zero[1]	1	0	0	0	0
zero	∞ , NaN, denorm	Norm	operand_d	1	0	0	0	0
zero	zero, Norm	∞ , NaN	dmax	1	0	0	0	0
zero	zero, Norm	denorm	zero[1]	1	0	0	0	0
zero	zero, Norm	zero	zero[1]	0	0	0	0	0
zero	zero, Norm	Norm	operand_d	0	0	0	0	0

Table 6-3. Floating-point Results Summary—madd, msub, nmadd, nmsub (Continued)

Operand A	Operand B	Operand D	Result	F INV	FOVF	FUNF	FDBZ	F INX
norm	∞ , NaN	∞ , NaN, denorm, zero, Norm	abmax	1	0	0	0	0
norm	denorm	∞ , NaN	dmax	1	0	0	0	0
norm	denorm	denorm, zero	zero[1]	1	0	0	0	0
norm	denorm	norm	operand_d	1	0	0	0	0
norm	zero	∞ , NaN	dmax	1	0	0	0	0
norm	zero	denorm	zero[1]	1	0	0	0	0
norm	zero	zero	zero[1]	0	0	0	0	0
norm	zero	norm	operand_d	0	0	0	0	0
norm	norm	∞ , NaN	dmax	1	0	0	0	0
norm	norm	denorm	ab_Calc	1	*	*	0	*
norm	norm	zero	ab_Calc	0	*	*	0	*
norm	norm	norm	_Calc_	0	*	*	0	*
nmadd								
∞ , NaN	∞ , NaN, Norm	∞ , NaN, denorm, zero, Norm	-abmax	1	0	0	0	0
∞ , NaN	denorm, zero	∞ , NaN	-dmax	1	0	0	0	0
∞ , NaN	denorm, zero	denorm, zero	zero[3]	1	0	0	0	0
∞ , NaN	denorm, zero	Norm	-operand_d	1	0	0	0	0
denorm	∞ , NaN, denorm, zero, Norm	∞ , NaN	-dmax	1	0	0	0	0
denorm	∞ , NaN, denorm, zero, Norm	denorm, zero	zero[3]	1	0	0	0	0
denorm	∞ , NaN, denorm, zero, Norm	Norm	-operand_d	1	0	0	0	0
zero	∞ , NaN, denorm,	∞ , NaN	-dmax	1	0	0	0	0
zero	∞ , NaN, denorm	denorm, zero	zero[3]	1	0	0	0	0
zero	∞ , NaN, denorm	Norm	-operand_d	1	0	0	0	0
zero	zero, Norm	∞ , NaN	-dmax	1	0	0	0	0
zero	zero, Norm	denorm	zero[3]	1	0	0	0	0
zero	zero, Norm	zero	zero[3]	0	0	0	0	0
zero	zero, Norm	Norm	-operand_d	0	0	0	0	0
norm	∞ , NaN	∞ , NaN, denorm, zero, Norm	-abmax	1	0	0	0	0
norm	denorm	∞ , NaN	-dmax	1	0	0	0	0

Table 6-3. Floating-point Results Summary—madd, msub, nmadd, nmsub (Continued)

Operand A	Operand B	Operand D	Result	F INV	FOVF	FUNF	FDBZ	F INX
norm	denorm	denorm, zero	zero[3]	1	0	0	0	0
norm	denorm	norm	-operand_d	1	0	0	0	0
norm	zero	∞ , NaN	-dmax	1	0	0	0	0
norm	zero	denorm	zero[3]	1	0	0	0	0
norm	zero	zero	zero[3]	0	0	0	0	0
norm	zero	norm	-operand_d	0	0	0	0	0
norm	norm	∞ , NaN	-dmax	1	0	0	0	0
norm	norm	denorm	-ab_Calc	1	*	*	0	*
norm	norm	zero	-ab_Calc	0	*	*	0	*
norm	norm	norm	-(_Calc_)	0	*	*	0	*
msub								
∞ , NaN	∞ , NaN, Norm	∞ , NaN, denorm, zero, Norm	abmax	1	0	0	0	0
∞ , NaN	denorm, zero	∞ , NaN	-dmax	1	0	0	0	0
∞ , NaN	denorm, zero	denorm, zero	zero[2]	1	0	0	0	0
∞ , NaN	denorm, zero	Norm	-operand_d	1	0	0	0	0
denorm	∞ , NaN, denorm, zero, Norm	∞ , NaN	-dmax	1	0	0	0	0
denorm	∞ , NaN, denorm, zero, Norm	denorm, zero	zero[2]	1	0	0	0	0
denorm	∞ , NaN, denorm, zero, Norm	Norm	-operand_d	1	0	0	0	0
zero	∞ , NaN, denorm,	∞ , NaN	-dmax	1	0	0	0	0
zero	∞ , NaN, denorm	denorm, zero	zero[2]	1	0	0	0	0
zero	∞ , NaN, denorm	Norm	-operand_d	1	0	0	0	0
zero	zero, Norm	∞ , NaN	-dmax	1	0	0	0	0
zero	zero, Norm	denorm	zero[2]	1	0	0	0	0
zero	zero, Norm	zero	zero[2]	0	0	0	0	0
zero	zero, Norm	Norm	-operand_d	0	0	0	0	0
norm	∞ , NaN	∞ , NaN, denorm, zero, Norm	abmax	1	0	0	0	0
norm	denorm	∞ , NaN	-dmax	1	0	0	0	0
norm	denorm	denorm, zero	zero[2]	1	0	0	0	0
norm	denorm	norm	-operand_d	1	0	0	0	0

Table 6-3. Floating-point Results Summary—madd, msub, nmadd, nmsub (Continued)

Operand A	Operand B	Operand D	Result	F INV	FOVF	FUNF	FDBZ	F INX
norm	zero	∞ , NaN	-dmax	1	0	0	0	0
norm	zero	denorm	zero[2]	1	0	0	0	0
norm	zero	zero	zero[2]	0	0	0	0	0
norm	zero	norm	-operand_d	0	0	0	0	0
norm	norm	∞ , NaN	-dmax	1	0	0	0	0
norm	norm	denorm	ab_Calc	1	*	*	0	*
norm	norm	zero	ab_Calc	0	*	*	0	*
norm	norm	norm	_Calc_	0	*	*	0	*
nmsub								
∞ , NaN	∞ , NaN, Norm	∞ , NaN, denorm, zero, Norm	-abmax	1	0	0	0	0
∞ , NaN	denorm, zero	∞ , NaN	dmax	1	0	0	0	0
∞ , NaN	denorm, zero	denorm, zero	zero[4]	1	0	0	0	0
∞ , NaN	denorm, zero	Norm	operand_d	1	0	0	0	0
denorm	∞ , NaN, denorm, zero, Norm	∞ , NaN	dmax	1	0	0	0	0
denorm	∞ , NaN, denorm, zero, Norm	denorm, zero	zero[4]	1	0	0	0	0
denorm	∞ , NaN, denorm, zero, Norm	Norm	operand_d	1	0	0	0	0
zero	∞ , NaN, denorm,	∞ , NaN	dmax	1	0	0	0	0
zero	∞ , NaN, denorm	denorm, zero	zero[4]	1	0	0	0	0
zero	∞ , NaN, denorm	Norm	operand_d	1	0	0	0	0
zero	zero, Norm	∞ , NaN	dmax	1	0	0	0	0
zero	zero, Norm	denorm	zero[4]	1	0	0	0	0
zero	zero, Norm	zero	zero[4]	0	0	0	0	0
zero	zero, Norm	Norm	-operand_d	0	0	0	0	0
norm	∞ , NaN	∞ , NaN, denorm, zero, Norm	-abmax	1	0	0	0	0
norm	denorm	∞ , NaN	dmax	1	0	0	0	0
norm	denorm	denorm, zero	zero[4]	1	0	0	0	0
norm	denorm	norm	operand_d	1	0	0	0	0
norm	zero	∞ , NaN	dmax	1	0	0	0	0
norm	zero	denorm	zero[4]	1	0	0	0	0

Table 6-3. Floating-point Results Summary—madd, msub, nmadd, nmsub (Continued)

Operand A	Operand B	Operand D	Result	F INV	FOVF	FUNF	FDBZ	F INX
norm	zero	zero	zero[4]	0	0	0	0	0
norm	zero	norm	operand_d	0	0	0	0	0
norm	norm	∞ , NaN	dmax	1	0	0	0	0
norm	norm	denorm	-ab_Calc	1	*	*	0	*
norm	norm	zero	-ab_Calc	0	*	*	0	*
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 6-4. Floating-Point Results Summary—sqrt

Operand A	Result	F INV	FOVF	FUNF	FDBZ	FINX
+∞	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	0	0	*
–norm	–0	1	0	0	0	0

Table 6-5. 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
+denorm	–∞	azero	1	0	0	0	0
+denorm	+NaN	azero	1	0	0	0	0

Table 6-5. Floating-Point Results Summary—Min, Max (Continued)

Operand A	Operand B	Result	FINV	FOVF	FUNF	FDBZ	FINX
+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
+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

Table 6-5. Floating-Point Results Summary—Min, Max (Continued)

Operand A	Operand B	Result	FINV	FOVF	FUNF	FDBZ	FINX
+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
+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

Table 6-5. Floating-Point Results Summary—Min, Max (Continued)

Operand A	Operand B	Result	FINV	FOVF	FUNF	FDBZ	FINX
–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
+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

Table 6-5. 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	–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 6-6. 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 6-7. Floating-point Results Summary—Convert to signed

Operand B	integer result efsctsi[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 6-8. Floating-point Results Summary—Convert from unsigned

Operand B	integer source efscfui	fractional source efscfuf	F INV	FOVF	FUNF	FDBZ	F INX
zero	zero	zero	0	0	0	0	0
norm	_Calc_	_Calc_	0	0	0	0	*

Table 6-9. 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 6-10. 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 6-11. 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 6-12. 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	*

6.5 EFPU Instruction Timing

Instruction timing in number of processor clock cycles for EFPU instructions is shown in Table 6-13 and Table 6-14. 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. The 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.

6.5.1 EFPU Single-Precision Vector Floating-Point Instruction Timing

Instruction timing for EFPU vector floating-point instructions is shown in Table 6-13. The table is sorted by opcode.

Table 6-13. EFPU Vector Floating-Point Instruction Timing

Instruction	Latency	Throughput	Comments
evfsabs	2	1	
evfsadd	2	1	
evfsaddx	2	1	
evfsaddsub	2	1	
evfsaddsubx	2	1	
evfscfh	2	1	
evfscfsf	2	1	
evfscfsi	2	1	
evfscfuf	2	1	
evfscfui	2	1	
evfscmpeq	2	1	
evfscmpgt	2	1	
evfscmplt	2	1	
evfscth	2	1	
evfsctsf	2	1	
evfsctsi	2	1	
evfsctsiz	2	1	
evfsctuf	2	1	
evfsctui	2	1	
evfsctuiz	2	1	
evfsdiff	2	1	
evfsdiffsum	2	1	
evfsdiv	13	13	blocking, no overlap with next inst.
evfsmax	2	1	
evfsmin	2	1	
evfsmadd	2	1[1]	dest also used as source
evfsmsub	2	1[1]	dest also used as source
evfsmul	2	1	

Table 6-13. EFPU Vector Floating-Point Instruction Timing (Continued)

Instruction	Latency	Throughput	Comments
evfsmule	2	1	
evfsmulo	2	1	
evfsmulx	2	1	
evfsnabs	2	1	
evfsneg	2	1	
evfsnmadd	2	1[1]	dest also used as source
evfsnmsub	2	1[1]	dest also used as source
evfssqrt	11	11	blocking, no overlap with next inst.
evfssub	2	1	
evfssubx	2	1	
evfssubadd	2	1	
evfssubaddx	2	1	
evfssum	2	1	
evfssumdiff	2	1	
evfststeq	2	1	
evfststgt	2	1	
evfststlt	2	1	

[1] Destination register is also a source register, so for full throughput, back-to-back operations must use a different dest reg.

6.5.2 EFPU Single-Precision Scalar Floating-Point Instruction Timing

Instruction timing for EFPU scalar floating-point instructions is shown in Table 6-14. The table is sorted by opcode.

Table 6-14. EFPU Scalar Floating-Point Instruction Timing

Instruction	Latency	Throughput	Comments
efsabs	2	1	
efsadd	2	1	
efscfh	2	1	
efscfsf	2	1	
efscfsi	2	1	
efscfuf	2	1	

Table 6-14. EFPU Scalar Floating-Point Instruction Timing (Continued)

Instruction	Latency	Throughput	Comments
efscfui	2	1	
efscmpeq	2	1	
efscmpgt	2	1	
efscmplt	2	1	
efscth	2	1	
efsctsf	2	1	
efsctsi	2	1	
efsctsiz	2	1	
efsctuf	2	1	
efsctui	2	1	
efsctuiz	2	1	
efsdiv	13	13	blocking, no execution overlap with next instruction
efsmadd	2	1[1]	dest also used as source
efsmsub	2	1[1]	dest also used as source
efsmax	2	1	
efsmin	2	1	
efsmul	2	1	
efsnabs	2	1	
efsneg	2	1	
efsnmadd	2	1[1]	dest also used as source
efsnmsub	2	1[1]	dest also used as source
efssqrt	11	11	blocking, no overlap with next inst.
efssub	2	1	
efststeq	2	1	
efststgt	2	1	
efststlt	2	1	

[1] Destination register is also a source register, so for full throughput, back-to-back operations must use a different dest reg.

6.6 Instruction Forms and Opcodes

Table 6-15 gives the division of the opcode space for the EFPU instructions.

Table 6-15. Opcode Space Division

Opcode Bits		Instruction Class
0–5	21–28	
4	0101 00xx	Embedded vector floating-point instructions
4	0101 010x	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 10xx	Embedded scalar floating-point single-precision instructions
4	0101 11xx	Reserved (Embedded scalar floating-point double-precision instructions)[1]

[1] Attempted execution of a defined EFP double-precision instruction will result in an Unimplemented instruction exception if MSR_{SPE} =1, or an EFPU Unavailable exception if MSR_{SPE}=0

6.6.1 Opcodes for EFPU Vector Floating-Point Instructions

Table 6-16 shows the embedded vector floating-point instruction opcodes.

Table 6-16. 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	—

Table 6-16. Embedded Vector Floating-Point Instruction Opcodes (Continued)

Instruction	Opcode Bits						Comments
	0–5	6–10	11–15	16–20	21–24	25–31	
evfscmpeq	4	crfD 00	rA	rB	0101	0001110	—
	4				0101	0001111	—
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	—

Table 6-16. Embedded Vector Floating-Point Instruction Opcodes (Continued)

Instruction	Opcode Bits						Comments
	0–5	6–10	11–15	16–20	21–24	25–31	
evfssubaddx	4	rD	rA	rB	0101	0101011	rA – rB; rA + rB
evfsmulx	4	rD	rA	rB	0101	0101100	—
	4	rD	rA	rB	0101	0101101	—
evfsmule	4	rD	rA	rB	0101	0101110	—
evfsmulo	4	rD	rA	rB	0101	0101111	—

6.6.2 Opcodes for EFPU Scalar Single-precision Floating-Point Instructions

Table 6-17 shows the embedded scalar single-precision floating-point instruction opcodes.

Table 6-17. 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	—

Table 6-17. 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	
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	—
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 7
Signal Processing Extension Unit

This chapter describes the instruction set architecture of the signal processing extension unit (SPE), version 1.1. This unit implements instructions to accelerate signal processing and other algorithms.

7.1 Nomenclature and Conventions

Several conventions regarding nomenclature are used in this chapter:

- Due to historical precedent, the terms SPE and SIMD are used interchangeably
- 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 SPE instructions generally begin with the letters ‘ev’ (vector).

7.2 SPE Programming Model

The e200z446n3 core provides a register file with thirty-two 64-bit registers. Power ISA embedded category instructions operate on the lower (least significant) 32 bits of the 64-bit register. New SPE instructions are defined that view the 64-bit register as being composed of a vector of two 32-bit elements, and some of the instructions also read or write 16-bit elements. These new instructions can also be used to perform scalar operations by ignoring the results of the upper 32-bit half of the register file. Some instructions are defined that produce a 64-bit scalar result. Vector fixed-point instructions operate on a vector of two 32-bit or four 16-bit fixed-point numbers resident in the 64-bit GPRs. The SPE and Power ISA embedded category instructions issue from a single instruction stream.

There are no record forms of SPE instructions. Vector 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. Vector compare instructions specify a CR field, two source registers, and the type of compare: greater than, less than, or equal. Two bits in the CR field are written with the result of the vector compare, one for each element. The remaining two bits reflect the ‘and’ing and ‘or’ing of the vector compare results.

A partially visible accumulator register is architected for the SPE integer and fractional multiply accumulate forms of instructions. Its usage is described in Section 7.2.2, “Accumulator.”

7.2.1 SPE Status and Control Register (SPEFSCR)

The e200z446n3 core implements the SPEFSCR register for status reporting and control of SPE instructions. This register is also used by the embedded floating-point unit. 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. The SPEFSCR is shown in Figure 7-1.

SOVH	OVH	FGH	FXH	FINVH	FDBZH	FUNFH	FOVFH	0		FINXS	FINVS	FDBZS	FUNFS	FOVFS	MODE	SOV	OV	FG	FX	FINV	FDBZ	FUNF	FOVF	0	FINXE	FINVE	FDBZE	FUNFE	FOVFE	FRMC	
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

SPR - 512; Read/Write; Reset - 0x0

Figure 7-1. SPE Status and Control Register (SPEFSCR)

The SPEFSCR bits are defined in Table 7-1.

Table 7-1. 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 Units.
3 (35)	FXH	Embedded Floating-Point Inexact Bit High Defined by Embedded Floating-Point Units.
4 (36)	FINVH	Embedded Floating-Point Invalid Operation/Input eError High Defined by Embedded Floating-Point Units.
5 (37)	FDBZH	Embedded Floating-Point Divide by Zero High Defined by Embedded Floating-Point Units.
6 (38)	FUNFH	Embedded Floating-Point Underflow High Defined by Embedded Floating-Point Units.
7 (39)	FOVFH	Embedded Floating-Point Overflow High Defined by Embedded Floating-Point Units.
8:9 (40:41)	—	Reserved
10 (42)	FINXS	Embedded Floating-Point Inexact Sticky Flag Defined by Embedded Floating-Point Units.
11 (43)	FINVS	Embedded Floating-Point Invalid Operation Sticky Flag Defined by Embedded Floating-Point Units.
12 (44)	FDBZS	Embedded Floating-Point Divide by Zero Sticky Flag Defined by Embedded Floating-Point Units.
13 (45)	FUNFS	Embedded Floating-Point Underflow Sticky Flag Defined by Embedded Floating-Point Units.
14 (46)	FOVFS	Embedded Floating-Point Overflow Sticky Flag Defined by Embedded Floating-Point Units.

Table 7-1. SPE Status and Control Register (Continued)

Bits	Name	Description
15 (47)	MODE	Embedded Floating-Point Operating Mode Defined by Embedded Floating-Point Units.
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 a **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 Units.
19 (51)	FX	Embedded Floating-Point Inexact bit (low/scalar) Defined by Embedded Floating-Point Units.
20 (52)	FINV	Embedded Floating-Point Invalid Operation/Input Error (low/scalar) Defined by Embedded Floating-Point Units.
21 (53)	FDBZ	Embedded Floating-Point Divide by Zero (low/scalar) Defined by Embedded Floating-Point Units.
22 (54)	FUNF	Embedded Floating-Point Underflow (low/scalar) Defined by Embedded Floating-Point Units.
23 (55)	FOVF	Embedded Floating-Point Overflow (low/scalar) Defined by Embedded Floating-Point Units.
24 (56)	—	Reserved
25 (57)	FINXE	Embedded Floating-Point Round (Inexact) Exception Enable Defined by Embedded Floating-Point Units.
26 (58)	FINVE	Embedded Floating-Point Invalid Operation/Input Error Exception Enable Defined by Embedded Floating-Point Units.
27 (59)	FDBZE	Embedded Floating-Point Divide by Zero Exception Enable Defined by Embedded Floating-Point Units.
28 (60)	FUNFE	Embedded Floating-Point Underflow Exception Enable Defined by Embedded Floating-Point Units.
29 (61)	FOVFE	Embedded Floating-Point Overflow Exception Enable Defined by Embedded Floating-Point Units.
30:31 (62:63)	FRMC	Embedded Floating-Point Rounding Mode Control Defined by Embedded Floating-Point Units.

7.2.2 Accumulator

The z446n3 core has a 64-bit architectural accumulator register that holds the results of the SPE multiply accumulate (MAC) fixed-point instructions. The accumulator allows back-to-back execution of dependent fixed-point MAC instructions, something that is found in inner loops of DSP code such as filters. The accumulator is partially visible to the programmer in that its results do not have to be explicitly read to use them. Instead, they are always copied into a 64-bit destination GPR specified as part of the instruction. The accumulator, however, has to be explicitly cleared when starting a new MAC loop. Based upon the type of instruction, an accumulator can hold either a single 64-bit value or a vector of two 32-bit elements.

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 (see Figure 7-51); 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. Thus the result of this instruction can be used by accessing **r**D.

To read the accumulator contents into a **register**, a multiply-accumulate instruction where one of its operands is a zero should be used, as the following sequence shows:

```
evxor RD, RD, RD                    // Zero the contents of RD, not necessary if
                                    // a zero is available in some register.
evmwumiaa RD, RD, RD                // Multiply 0 with 0, add the 0 result to
                                    // accumulator and store back the value in acc and RD
```

To initialize the accumulator, the **evmra** instruction is used.

7.2.2.1 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.

7.2.3 GPRs and Power ISA Embedded Category Instructions

The z446n3 core implements the 32-bit forms of the Power ISA embedded category instructions. All 32-bit Power ISA embedded category instructions operate upon the lower half of the 64-bit GPR. These instructions do not affect the upper half of a GPR.

7.2.4 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 s **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.

7.2.5 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.

7.2.6 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 the Power ISA. These registers are privileged.

7.2.6.1 SPE Unavailable Exception

The SPE Unavailable exception is taken if MSR[SPE] is cleared and execution of an SPE instruction other than the **brinc** instruction is attempted. When the SPE 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.
- The ESR[SPE] bit is set. All other ESR bits are cleared.

Instruction execution resumes at address $IVPR_{0:15}||IVOR32_{16:27}||0b0000$.

7.2.6.2 SPE Vector Alignment Exception

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**, **evlddx**, **evldw**, **evldwx**, **evldh**, **evldhx**, **evstdd**, **evstddx**, **evstdw**, **evstdwx**, **evstdh**, 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 $IVPR_{0:15}||IVOR5_{16:27}||0b0000$.

7.2.7 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 will be taken if an SPE double-word vector load or store access is attempted with an address which is not 32-bit aligned.

7.3 Integer SPE Simple Instructions

This section explains the following integer SPE simple instructions:

Table 7-2. Integer SPE Simple Instructions

Bit Reversed Increment (brinc)	Vector Divide Word Unsigned (evdivwu)
Vector Absolute Value (evabs)	Vector Equivalent (eveqv)
Vector Add Immediate Word (evaddiw)	Vector Extend Sign Byte (evextsb)
Vector Add Word (evaddw)	Vector Extend Sign Half Word (evextsh)
Vector AND (evand)	Vector Merge High (evmergehi)
Vector AND with Complement (evandc)	Vector Merge High/Low (evmergehilo)
Vector Compare Equal (evcmpeq)	Vector Merge Low (evmergelo)
Vector Compare Greater Than Signed (evcmpgts)	Vector Merge Low/High (evmergelohi)
Vector Compare Greater Than Unsigned (evcmpgts)	Vector NAND (evnand)
Vector Compare Less Than Signed (evcmplts)	Vector Negate (evneg)
Vector Compare Less Than Unsigned (evcmpltu)	Vector NOR (evnor)
Vector Count Leading Sign Bits Word (evcntlsw)	Vector OR (evor)
Vector Count Leading Zeros Word (evcntlzw)	Vector OR with Complement (evorc)
Vector Divide Word Signed (evdivws)	Vector Rotate Left Word (evrlw)
Vector Rotate Left Word Immediate (evrlwi)	Vector Shift Right Word Immediate Signed (evsrwis)
Vector Round Word (evrndw)	Vector Shift Right Word Immediate Unsigned (evsrwiu)
Vector Select (evsel)	Vector Shift Right Word Signed (evsrws)
Vector Shift Left Word (evslw)	Vector Shift Right Word Unsigned (evsrwu)
Vector Shift Left Word Immediate (evslwi)	Vector Subtract from Word (evsubfw)
Vector Splat Fractional Immediate (evsplatfi)	Vector Subtract Immediate from Word (evsubifw)
Vector Splat Immediate (evsplati)	Vector XOR (evxor)

brinc brinc

Bit Reversed Increment

brinc **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	rD	RA	RB	010 0000 1111

```
n = 16                          // Implementation dependent value
mask = $rB_{64-n:63}$           // Least sig. n bits of 32-bit reg
a = $rA_{64-n:63}$
d = bitreverse(1 + bitreverse(a | (¬ mask)))
$rD_{32:63}$ = $rA_{32:63-n}$ || (d & mask) // || is concatenation
```

The **brinc** instruction provides a way for software to access FFT data in a bit-reversed manner. rA contains the index into a buffer that contains data on which FFT is to be performed. rB contains a mask that allows the index to be updated with bit-reversed addressing. Typically this instruction precedes a load with index instruction, for example,

```
brinc r2, r3, r4
lhax r8, r5, r2
```

rB contains a bit-mask that is based upon the number of points in an FFT. To access a buffer containing n byte sized data that is to be accessed with bit-reversed addressing, the mask has $\log_2 n$ ‘1’s in the lsb positions and ‘0’s in the remaining most significant position. If however, the data size is a multiple of a half word or a word, the mask is constructed so that the ‘1’s are shifted left by $\log_2$ (size of the data) and ‘0’s are placed in the lsb positions. Table 7-3 shows example values of masks for different data sizes and number of data.

Table 7-3. Data Samples and Sizes

Number of Data Samples	Data size			
	Byte	Half Word	Word	Double Word
8	000...00000111	000...00001110	000...000011100	000...0000111000
16	000...00001111	000...00011110	000...000111100	000...0001111000
32	000...00011111	000...00111110	000...001111100	000...0011111000
64	000...00111111	000...01111110	000...011111100	000...0111111000

Note: An implementation can restrict the number of bits specified in a mask. In the z446n3 implementation, the number of bits is 16 which allows the user to perform bit-reversed address computations for 65536 byte sized samples.

evabs evabs

Vector Absolute Value

evabs **r**D,**r**A

0 5	6 10	11 15	16 20	21 31
4	RD	RA	0000 0	010 0000 1000

$RD_{0:31} = ABS(RA_{0:31})$
$RD_{32:63} = ABS(RA_{32:63})$

The absolute value of each element of **r**A is placed into the corresponding element of **r**D. Absolute value of 0x8000_0000 (most negative number) returns 0x8000_0000. No overflow is detected

evaddiw evaddiw

Vector Add Immediate Word

evaddiw **r**D**,r**B,UIMM

0 5	6 10	11 15	16 20	21 31
4	RD	UIMM	RB	010 0000 0010

$RD_{0:31}$ = $RB_{0:31}$ + EXTZ(UIMM)// Modulo sum
$RD_{32:63}$ = $RB_{32:63}$ + EXTZ(UIMM)// Modulo sum

The 5-bit UIMM value is zero-extended and added to each element of **r**B. The results are placed into the corresponding elements of **r**D.

evaddw evaddw

Vector Add Word

evaddw **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	010 0000 0000

```
$RD_{0:31} = RA_{0:31} + RB_{0:31}$     // Modulo sum
$RD_{32:63} = RA_{32:63} + RB_{32:63}$ // Modulo sum
```

Adds each element of **r**A to the corresponding element of **r**B and places the results into the corresponding elements of **r**D. The sum is a modulo sum.

evand evand

Vector AND

evand **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	010 0001 0001

$RD_{0:31} = RA_{0:31}$ & $RB_{0:31}$ // Bitwise AND
$RD_{32:63} = RA_{32:63}$ & $RB_{32:63}$// Bitwise AND

Performs a bitwise AND of each element of **r**A and **r**B and places the results into the corresponding elements of **r**D.

evandc evandc

Vector AND with Complement

evandc **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	010 0001 0010

$RD_{0:31} = RA_{0:31}$ & $(\neg RB_{0:31})$ // Bitwise ANDC
$RD_{32:63} = RA_{32:63}$ & $(\neg RB_{32:63})$ // Bitwise ANDC

Performs a bitwise AND of each element of **r**A and complement of **r**B and places the results into the corresponding elements of **r**D.

evcmpeq

evcmpeq

Vector Compare Equal

evcmpeq **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 0 0 1 1 0 1 0 0

```
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)
```

The msb in **crf**D is set if the high-order element of **r**A is equal to the high-order element of **r**B and cleared otherwise. The next most significant bit in crfD is set if the lower order element of **r**A is equal to the lower order element of **r**B and cleared otherwise. The last two bits of **crf**D are set to the OR and AND of the result of the compare of the high and low elements.

evcmpgts evcmpgts

Vector Compare Greater Than Signed

evcmpgts **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 0 0 1 1 0 0 0 1

```
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)
```

The msb in crfD is set if the high-order element of **r**A is greater than the high-order element of **r**B and cleared otherwise. The next most significant bit in crfD is set if the lower order element of **r**A is greater than the lower order element of **r**B, and cleared otherwise. The last two bits of crfD are set to the OR and AND of the result of the compare of the high and low elements.

evcmpgtu

evcmpgtu

Vector Compare Greater Than Unsigned

evcmpgtu **crf**D,**r**A,**r**B

0 5	6 8	9 10	11 15	16 20	21 31
4	crfD	0 0	RA	RB	010 0011 0000

```
ah = $RA_{0:31}$
al = $RA_{32:63}$
bh = $RB_{0:31}$
bl = $RB_{32:63}$
if (ah >U bh) then ch = 1
else ch = 0
if (al >U bl) then cl = 1
else cl = 0
$CR_{4*crfD:4*crfD+3}$ = ch || cl || (ch | cl) || (ch & cl)
```

The msb in crfD is set if the high-order element of **r**A is greater than the high-order element of **r**B, and cleared otherwise. The next most significant bit in crfD is set if the lower order element of **r**A is greater than the lower order element of **r**B, and cleared otherwise. The last two bits of crfD are set to the OR and AND of the result of the compare of the high and low elements.

evcmplts evcmplts

Vector Compare Less Than Signed

evcmplts **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 0 0 1 1 0 0 1 1

```
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)
```

The msb in crfD is set if the high-order element of **r**A is less than the high-order element of **r**B and cleared otherwise. The next most significant bit in crfD is set if the lower order element of **r**A is less than the lower order element of **r**B and cleared otherwise. The last two bits of crfD are set to the OR and AND of the result of the compare of the high and low elements.

evcmpltu evcmpltu

Vector Compare Less Than Unsigned

evcmpltu **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 0 0 1 1 0 0 1 0

```
ah = $RA_{0:31}$
al = $RA_{32:63}$
bh = $RB_{0:31}$
bl = $RB_{32:63}$
if (ah <U bh) then ch = 1
else ch = 0
if (al <U bl) then cl = 1
else cl = 0
$CR_{4*crfD:4*crfD+3}$ = ch || cl || (ch | cl) || (ch & cl)
```

The msb in crfD is set if the high-order element of **r**A is less than the high-order element of **r**B and cleared otherwise. The next most significant bit in crfD is set if the lower order element of **r**A is less than the lower order element of **r**B and cleared otherwise. The last two bits of crfD are set to the OR and AND of the result of the compare of the high and low elements.

evcntlsw evcntlsw

Vector Count Leading Sign Bits Word

evcntlsw **r**D,**r**A

0 5	6 10	11 15	16 20	21 31
4	RD	RA	0000 0	010 0000 1110

Counts the leading number of sign bits in each element of **r**A and places the counts into corresponding elements of **r**D.

evcntlzw evcntlzw

Vector Count Leading Zeros Word

evcntlzw **r**D,**r**A

0 5	6 10	11 15	16 20	21 31
4	RD	RA	0000 0	010 0000 1101

Counts the leading number of zeros in each element of **r**A and places the counts into corresponding elements of **r**D.

evdivws

Vector Divide Word Signed

evdivws **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 1100 0110

```
    dividendh = $RA_{0:31}$
    dividendl = $RA_{32:63}$
    divisorh = $RB_{0:31}$
    divisorl = $RB_{32:63}$
    $RD_{0:31}$ = dividendh ÷ divisorh
    $RD_{32:63}$ = dividendl ÷ divisorl
Implementation Details:
    ovh = 0
    ovl = 0

        if ((dividendh<0) && (divisorh==0)) then
            $RD_{0:31}$ = 0x80000000
            ovh = 1
        else if ((dividendh>=0) && (divisorh==0)) then
            $RD_{0:31}$ = 0x7FFFFFFF
            ovh = 1
        else if ((dividendh==0x80000000) && (divisorh==-1)) then
            $RD_{0:31}$ = 0x7FFFFFFF
            ovh = 1
        if ((dividendl<0) && (divisorl==0)) then
            $RD_{32:63}$ = 0x80000000
            ovl = 1
        else if ((dividendl>=0) && (divisorl==0)) then
            $RD_{32:63}$ = 0x7FFFFFFF
            ovl = 1
        else if ((dividendl==0x80000000) && (divisorl==-1)) then
            $RD_{32:63}$ = 0x7FFFFFFF
            ovl = 1

    $SPEFSCR_{OVH}$ = ovh
    $SPEFSCR_{OV}$ = ovl
    $SPEFSCR_{SOVH}$ = $SPEFSCR_{SOVH}$ | ovh
    $SPEFSCR_{SOV}$ = $SPEFSCR_{SOV}$ | ovl
```

The two dividends are the two elements of the contents of **r**A. The two divisors are the two elements of the contents of **r**B. Two 32-bit quotients are formed as a result of the division on each of the upper and lower elements and the quotients are placed into **r**D. The remainders are not supplied as a result of this operation. Both the operands and quotients are interpreted as signed integers. If an overflow occurs (see the PowerPC ISA **divw** instruction for the cases), the corresponding SPEFSCR bits are set. Otherwise, they are cleared. In case of overflow, a saturated value is delivered into the destination register.

evdivwu evdivwu

Vector Divide Word Unsigned

evdivwu **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 1100 0111

```
    dividendh = $RA_{0:31}$
    dividendl = $RA_{32:63}$
    divisorh = $RB_{0:31}$
    divisorl = $RB_{32:63}$
    $RD_{0:31}$ = dividendh ÷ divisorh
    $RD_{32:63}$ = dividendl ÷ divisorl
Implementation Details:
    ovh = 0
    ovl = 0
        if (divisorh == 0) then
            $RD_{0:31}$ = 0xFFFFFFFF
            ovh = 1
        if (divisorl == 0) then
            $RD_{32:63}$ = 0xFFFFFFFF
            ovl = 1
    $SPEFSCR_{OVH}$ = ovh
    $SPEFSCR_{OV}$ = ovl
    $SPEFSCR_{SOVH}$ = $SPEFSCR_{SOVH}$ | ovh
    $SPEFSCR_{SOV}$ = $SPEFSCR_{SOV}$ | ovl
```

The two dividends are the two elements of the contents of **r**A. The two divisors are the two elements of the contents of **r**B. Two 32-bit quotients are formed as a result of the division on each of the upper and lower elements and the quotients are placed into **r**D. The remainders are not supplied as a result of this operation. Both the operands and quotients are interpreted as unsigned integers. If an overflow occurs (see the PowerPC ISA **divuw** instruction for the cases), the corresponding SPEFSCR bits are set. Otherwise, they are cleared. In case of overflow, a saturated value is delivered into the destination register.

eveqv eveqv

Vector Equivalent

eveqv **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	010 0001 1001

$RD_{0:31} = RA_{0:31} \equiv RB_{0:31}$ // Bitwise XNOR
$RD_{32:63} = RA_{32:63} \equiv RB_{32:63}$ // Bitwise XNOR

Performs a bitwise XNOR of each element of **r**A and **r**B and places the results into the corresponding elements of **r**D.

evextsb evextsb

Vector Extend Sign Byte

evextsb **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 0 0 0 0 1 0 1 0

$RD_{0:31} = EXTS(RA_{24:31})$
$RD_{32:63} = EXTS(RA_{56:63})$

Extends the sign of the low-order byte in each of the elements in **r**A and places the results into **r**D.

evextsh evextsh

Vector Extend Sign Half Word

evextsh **r**D,**r**A

0 5	6 10	11 15	16 20	21 31
4	RD	RA	0000 0	010 0000 1011

$RD_{0:31} = EXTS(RA_{16:31})$
$RD_{32:63} = EXTS(RA_{48:63})$

Extends the sign of the half words in each of the elements in **r**A and places the results into **r**D.

evmergehi evmergehi

Vector Merge High

evmergehi **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	010 0010 1100

$RD_{0:31} = RA_{0:31}$
$RD_{32:63} = RB_{0:31}$

The high-order elements of **r**A and **r**B are merged and placed into **r**D as shown in Figure 7-2.

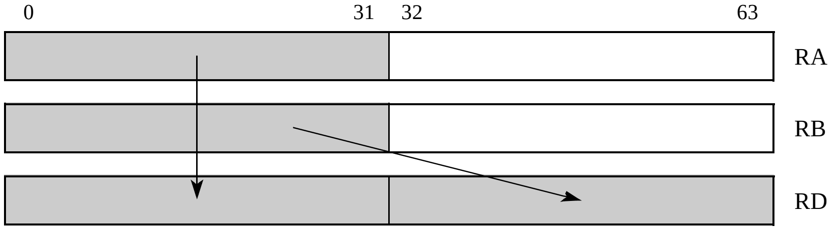


Figure 7-2. High Order Element Merging with evmergehi

Note: A vector splat high can be performed by specifying the same register in **r**A and **r**B.

evmergehilo evmergehilo

Vector Merge High/Low

evmergehilo **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	010 0010 1110

$RD_{0:31} = RA_{0:31}$
$RD_{32:63} = RB_{32:63}$

The high-order element of **r**A and the low-order element of **r**B are merged and placed into **r**D as shown in Figure 7-3.

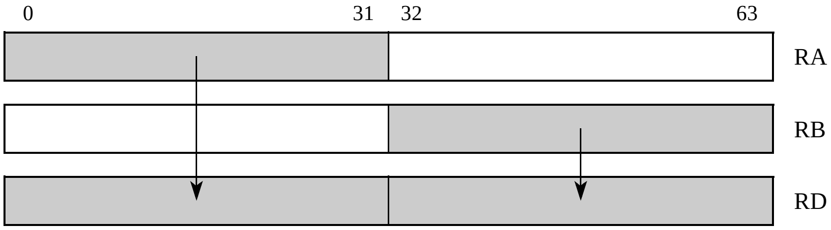


Figure 7-3. High Order Element Merging with evmergehilo

evmergelo evmergelo

Vector Merge Low

evmergelo **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	010 0010 1101

$RD_{0:31} = RA_{32:63}$
$RD_{32:63} = RB_{32:63}$

The low-order elements of **r**A and **r**B are merged and placed in **r**D as shown in Figure 7-4.

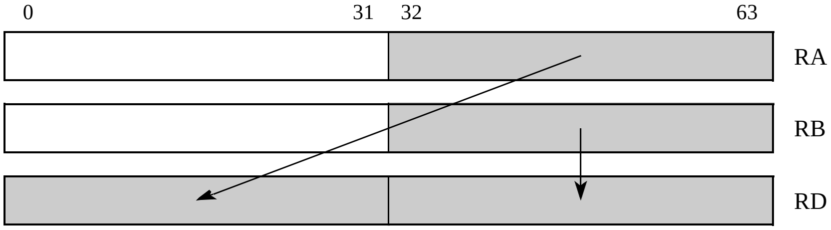


Figure 7-4. Low Order Element Merging evmergelo

Note: A vector splat low can be performed by specifying the same register in **r**A and **r**B.

evmergelohi evmergelohi

Vector Merge Low/High

evmergelohi **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	010 0010 1111

$RD_{0:31} = RA_{32:63}$
$RD_{32:63} = RB_{0:31}$

The low-order element of **r**A and the high-order element of **r**B are merged and placed into **r**D as shown in Figure 7-5.

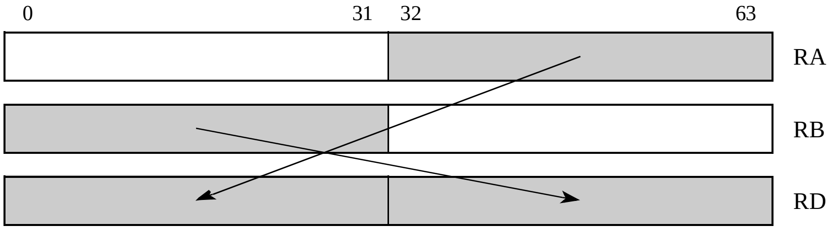


Figure 7-5. Low Order Element Merging evmergelohi

Note: A vector swap can be performed by specifying the same register in **r**A and **r**B.

evnand evnand

Vector NAND

evnand **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	010 0001 1110

$RD_{0:31}$ = ¬($RA_{0:31}$ & $RB_{0:31}$)// Bitwise NAND
$RD_{32:63}$ = ¬($RA_{32:63}$ & $RB_{32:63}$) // Bitwise NAND

Performs a bitwise NAND of each element of **r**A and **r**B and places the results into the corresponding elements of **r**D.

evneg evneg

Vector Negate

evneg **r**D,**r**A

0 5	6 10	11 15	16 20	21 31
4	RD	RA	0000 0	010 0000 1001

$RD_{0:31}$ = NEG($RA_{0:31}$)
$RD_{32:63}$ = NEG($RA_{32:63}$)

The negative value of each element of **r**A is placed in **r**D. The negative value of 0x8000_0000 (most negative number) returns 0x8000_0000. No overflow is detected.

evnor evnor

Vector NOR

evnor **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	010 0001 1000

$RD_{0:31} = \neg(RA_{0:31} \mid RB_{0:31})$ // Bitwise NOR
$RD_{32:63} = \neg(RA_{32:63} \mid RB_{32:63})$ // Bitwise NOR

Performs a bitwise NOR of each element of **r**A and **r**B and places the result into the corresponding element of **r**D.

evor evor

Vector OR

evor **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	010 0001 0111

```
$RD_{0:31} = RA_{0:31} \mid RB_{0:31}$      //Bitwise OR
$RD_{32:63} = RA_{32:63} \mid RB_{32:63}$   // Bitwise OR
```

Performs a bitwise OR of each element of **r**A and **r**B and places the results into the corresponding elements of **r**D.

evorc

Vector OR with Complement

evorc **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	010 0001 1011

$RD_{0:31} = RA_{0:31} \mid (\neg RB_{0:31})$ // Bitwise ORC
$RD_{32:63} = RA_{32:63} \mid (\neg RB_{32:63})$ // Bitwise ORC

Performs a bitwise OR of each element of **r**A and complement of **r**B and places the results in the corresponding elements of **r**D.

evrlw

evrlw

Vector Rotate Left Word

evrlw **r**D,**r**A,**r**B

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	010 0010 1000

```
$nh = RB_{27:31}$
$nl = RB_{59:63}$
$RD_{0:31} = ROTL(RA_{0:31}, nh)$
$RD_{32:63} = ROTL(RA_{32:63}, nl)$
```

Rotates left each of the elements of **r**A by amounts specified in **r**B and places the results into **r**D. The rotate amounts are specified by 5 bit fields in **r**B. Separate rotate values for each element of **r**A are specified in bit positions **r**B$_{27:31}$ and **r**B$_{59:63}$.

evrlwi evrlwi

Vector Rotate Left Word Immediate

evrlwi **r**D,**r**A,UIMM

0 5	6 10	11 15	16 20	21 31
4	RD	RA	UIMM	010 0010 1010

n = UIMM
$RD_{0:31}$ = ROTL($RA_{0:31}$, n)
$RD_{32:63}$ = ROTL($RA_{32:63}$, n)

Rotates left both elements of **r**A by an amount specified by the 5-bit UIMM immediate value and places the results into **r**D.

evrndw evrndw

Vector Round Word

evrndw **r**D,**r**A

0 5	6 10	11 15	16 20	21 31
4	RD	RA	0000 0	010 0000 1100

```
$RD_{0:31}$ = ($RA_{0:31}$+0x00008000) & 0xFFFF0000 // Modulo sum
$RD_{32:63}$ = ($RA_{32:63}$+0x00008000) & 0xFFFF0000 // Modulo sum
```

Rounds the 32-bit elements of **r**A into 16 bits and places the results into **r**D. The resulting 16 bits of each element are placed in the most significant 16 bits of each element of **r**D, zeroing out the low order 16 bits of each element.

evsel evsel

Vector Select

evsel **r**D,**r**A,**r**B,**crf**S

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

```
ch = $CR_{crfS*4}$
cl = $CR_{crfS*4+1}$
if (ch == 1) then $RD_{0:31} = RA_{0:31}$
else $RD_{0:31} = RB_{0:31}$
if (cl == 1) then $RD_{32:63} = RA_{32:63}$
else $RD_{32:63} = RB_{32:63}$
```

If the msb if the **crf**S field of CR is set, the high-order element of **r**A is placed in the high-order element of **r**D; otherwise, the high-order element of **r**B is placed into the higher order element of **r**D. If the next most significant bit in the **crf**S field of CR is set, the low-order element of **r**A is placed in the low-order element of **r**D. Otherwise, the low-order element of **r**B is placed into the lower order element of **r**D. This is shown in Figure 7-6.

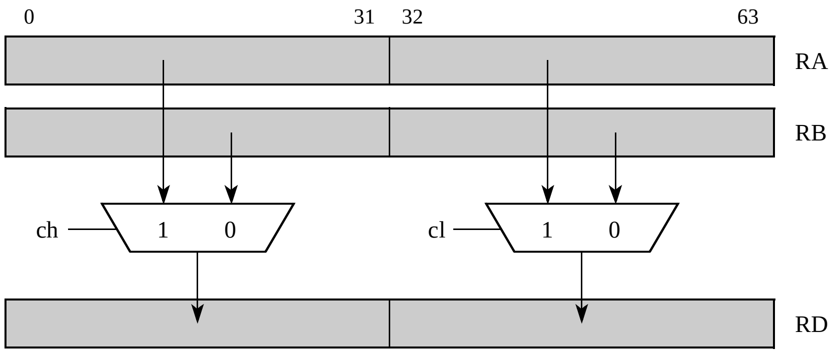


Figure 7-6. evsel

evslw evslw

Vector Shift Left Word

evslw **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	010 0010 0100

$nh = RB_{26:31}$
$nl = RB_{58:63}$
$RD_{0:31} = SL(RA_{0:31}, nh)$
$RD_{32:63} = SL(RA_{32:63}, nl)$

Shifts left each element of **r**A by amounts specified in **r**B and places the results into **r**D. The separate shift amounts for each element are specified by 6-bit fields in **r**B in bit positions 26:31 and 58:63. Shift amounts from 32 to 63 give a zero result.

evslwi evslwi

Vector Shift Left Word Immediate

evslwi **r**D,**r**A,UIMM

0 5	6 10	11 15	16 20	21 31
4	RD	RA	UIMM	010 0010 0110

```
n = UIMM
$RD_{0:31} = SL(RA_{0:31}, n)$
$RD_{32:63} = SL(RA_{32:63}, n)$
```

Shifts left each element of **r**A by the 5-bit UIMM value and places the results into **rD.**

evsplatfi

evsplatfi

Vector Splat Fractional Immediate

evsplatfi **r**D,SIMM

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

$RD_{0:31}$ = SIMM || $^{27}0$
$RD_{32:63}$ = SIMM || $^{27}0$

The 5-bit SIMM value is padded with trailing zeros and placed into both elements of **r**D as shown in Figure 7-7. The SIMM value is placed in bit positions **r**$D_{0:4}$ and **r**$D_{32:36}$.

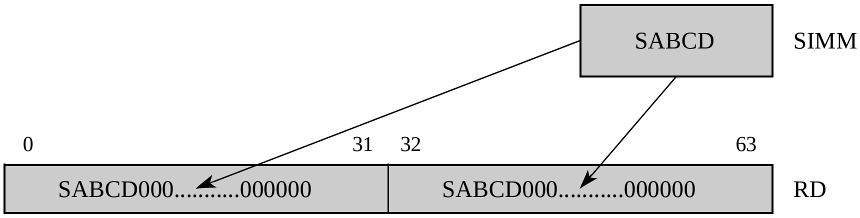


Figure 7-7. Splat for evsplatfi

evsplati evsplati

Vector Splat Immediate

evsplati **r**D,SIMM

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

$RD_{0:31}$ = EXTS(SIMM)
$RD_{32:63}$ = EXTS(SIMM)

The 5-bit SIMM immediate value is sign extended and placed into both elements of **r**D as shown in Figure 7-8.

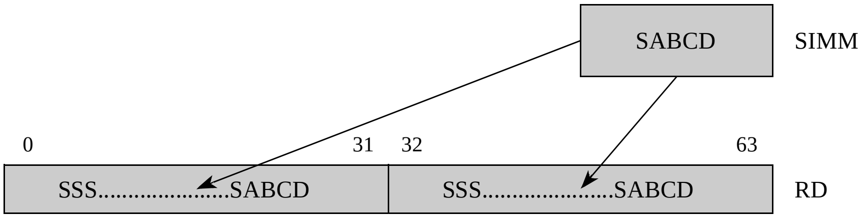

SABCD
SIMM
0
31
32
63
SSS......................SABCD
SSS......................SABCD
RD

Figure 7-8. Sign Extend in evsplati

evsrwis

Vector Shift Right Word Immediate Signed

evsrwis **r**D,**r**A,UIMM

0–5	6–10	11–15	16–20	21–31
4	RD	RA	UIMM	010 0010 0011

n = UIMM
$RD_{0:31}$ = EXTS($RA_{0:31-n}$)
$RD_{32:63}$ = EXTS($RA_{32:63-n}$)

Shifts right arithmetically each element of **r**A by the 5-bit UIMM value and places the results into **r**D. The sign bit of each source element in **r**A is extended right into the most significant bit positions of each result element.

evsrwiu evsrwiu

Vector Shift Right Word Immediate Unsigned

evsrwiu **r**D,**r**A,UIMM

0 5	6 10	11 15	16 20	21 31
4	RD	RA	UIMM	010 0010 0010

n = UIMM
$RD_{0:31}$ = EXTZ($RA_{0:31-n}$)
$RD_{32:63}$ = EXTZ($RA_{32:63-n}$)

Shifts right logically each element of **r**A by the 5-bit UIMM value and places the results into **r**D. ‘0’ bits are shifted in to the most significant bit positions of each result element.

evsrws

evsrws

Vector Shift Right Word Signed

evsrws **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	010 0010 0001

```
nh = $RB_{26:31}$
nl = $RB_{58:63}$
$RD_{0:31}$ = EXTS($RA_{0:31-nh}$)
$RD_{32:63}$ = EXTS($RA_{32:63-nl}$)
```

Shifts right arithmetically each element of **r**A by an amount specified in **r**B and places the results into **r**D. Separate shift amounts for each element are specified by 6-bit fields in **r**B that lie in bit positions 26–31 and 58–63. The sign bit of each source element in **r**A is extended right into the most significant bit positions of each result element.

Shift amounts from 32 to 63 give a result of 32 sign bits.

evsrwu evsrwu

Vector Shift Right Word Unsigned

evsrwu **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	010 0010 0000

```
nh = $RB_{26:31}$
nl = $RB_{58:63}$
$RD_{0:31}$ = EXTZ($RA_{0:31-nh}$)
$RD_{32:63}$ = EXTZ($RA_{32:63-nl}$)
```

Shifts right logically each element of **r**A by amounts specified in **r**B and places the results into **r**D. Separate shift amounts for each element are specified by 6-bit fields in **r**B that lie in bit positions 26–31 and 58–63. Zero bits are shifted in to the most significant bit positions.

Shift amounts from 32 to 63 give a zero result.

evsubfw evsubfw

Vector Subtract from Word

evsubfw **r**D,**r**A,**r**B

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	010 0000 0100

$RD_{0:31} = RB_{0:31} - RA_{0:31}$ // Modulo sum
$RD_{32:63} = RB_{32:63} - RA_{32:63}$ // Modulo sum

Each element of **r**A is subtracted from the corresponding element of **r**B and the results are placed into the corresponding elements of **r**D.

evsubifw evsubifw

Vector Subtract Immediate from Word

evsubifw **r**D,UIMM,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	UIMM	RB	010 0000 0110

$RD_{0:31}$ = $RB_{0:31}$ - EXTZ(UIMM) // Modulo sum
$RD_{32:63}$ = $RB_{32:63}$ - EXTZ(UIMM)// Modulo sum

The 5-bit UIMM value is zero-extended and subtracted from each element of **r**B. The results are placed into the corresponding elements of **r**D. Note that the same value is subtracted from each element.

evxor evxor

Vector XOR

evxor **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	010 0001 0110

$RD_{0:31} = RA_{0:31} \oplus RB_{0:31}$ // Bitwise XOR
$RD_{32:63} = RA_{32:63} \oplus RB_{32:63}$// Bitwise XOR

Performs a bitwise exclusive-OR of each element of **r**A and **r**B and places the results into the corresponding elements of **r**D.

7.4 Integer SPE Multiply, Multiply-Accumulate, and Operation to Accumulator Instructions (Complex Integer Instructions)

A number of forms of multiply and multiply-accumulate operations are supported in the SPE, as are add and subtract to accumulator operations. The SPE supports signed and unsigned forms as well as optional fractional forms. For all of these instructions, the fractional form does not apply to unsigned forms because integer and fractional forms are identical for unsigned operands.

Table 7-4 defines mnemonic extensions for these instructions.

Table 7-4. 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 accum result
ho	half word odd	16 × 16→32
hog	half word odd guarded	16 × 16→32, 64-bit final accum result
w	word	32 × 32→64
wh	word high	32 × 32→32 high order 32-bits of product
wl	word low	32 × 32→32 low order 32-bits of product
data type		
smf	signed modulo fractional	(wrap, no saturate)
smi	signed modulo integer	(wrap, no saturate)
ssf	signed saturate fractional	—
ssi	signed saturate integer	—
umi	unsigned modulo integer	(wrap, no saturate)

Table 7-4. Mnemonic Extensions for Multiply-Accumulate Instructions (Continued)

extension	meaning	comments
usi	unsigned saturate integer	—
accumulate options		
a	update accumulator	update accumulator (no add)
aa	add to accumulator	add result to accumulator (64-bit sum)
aaw	add to accumulator (words)	add word results to accumulator words (pair of 32-bit sums)
an	add negated	add negated result to accumulator (64-bit sum)
anw	add negated to accumulator (words)	add negated word results to accumulator words (pair of 32-bit sums)

7.4.1 Multiply Half-Word Instructions

The following instructions perform 16 × 16 multiplies with optional saturation from the odd or even half of elements, with and without accumulates, using signed or unsigned integer or fractional operands.

evmhegsmfaa evmhegsmfaa

Multiply Half Words, Even, Guarded, Signed, Modulo, Fractional, and Accumulate

evmhegsmfaa **r**D,**r**A,**r**B (O=0, F=1, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 0010 1011

$prod_{0:31} = rA_{32:47} * rB_{32:47}$

$temp1_{0:63} = EXTS(prod_{0:31} \| 0)$

$temp2_{0:64} = ACC_{0:63} + temp1_{0:63}$

$rD_{0:63} = ACC_{0:63} = temp2_{1:64}$

The low even-numbered signed fractional half-word element in **r**A is multiplied by the corresponding signed fractional half-word element in **r**B. The 32-bit intermediate product is sign-extended to 64 bits and then shifted left by one bit and added to the contents of the 64-bit accumulator to form a 65-bit intermediate sum. The lower 64 bits of the intermediate sum are placed back into the accumulator and also written into **r**D.

NOTE

This is a modulo sum. There is no check for overflow and no saturation is performed. An overflow from the 64-bit sum, if one occurs, is not recorded into SPEFSCR.

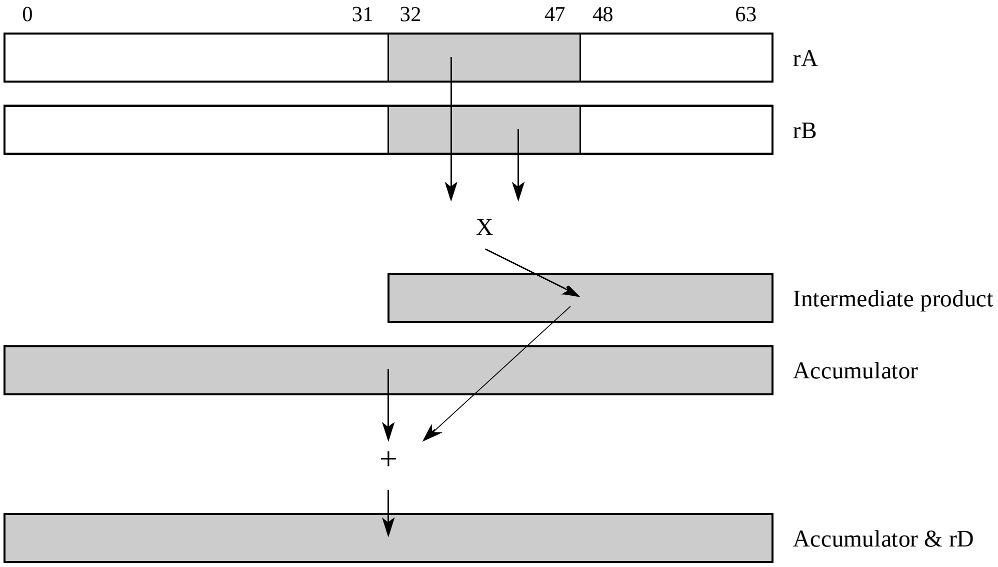


Figure 7-9. evmhegsmfaa

evmhegsmfan evmhegsmfan

Multiply Half Words, Even, Guarded, Signed, Modulo, Fractional, and Accumulate Negative

evmhegsmfan **r**D,**r**A,**r**B (O=0, F=1, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 1010 1011

$prod_{0:31} = rA_{32:47} * rB_{32:47}$

$temp1_{0:63} = EXTS(prod_{0:31} \| 0)$

$temp2_{0:64} = ACC_{0:63} - temp1_{0:63}$

$rD_{0:63} = ACC_{0:63} = temp2_{1:64}$

The low even-numbered signed fractional half word element in **r**A is multiplied by the corresponding signed fractional half word element in **r**B. The 32-bit intermediate product is sign-extended to 64 bits and then shifted left by one bit and subtracted from the contents of the 64-bit accumulator to form a 65-bit intermediate difference. The lower 64 bits of the intermediate difference is placed back into the accumulator and also written into **r**D.

NOTE

This is a modulo difference. There is no check for overflow and no saturation is performed. An overflow from the 64-bit difference, if one occurs, is not recorded into SPEFSCR.

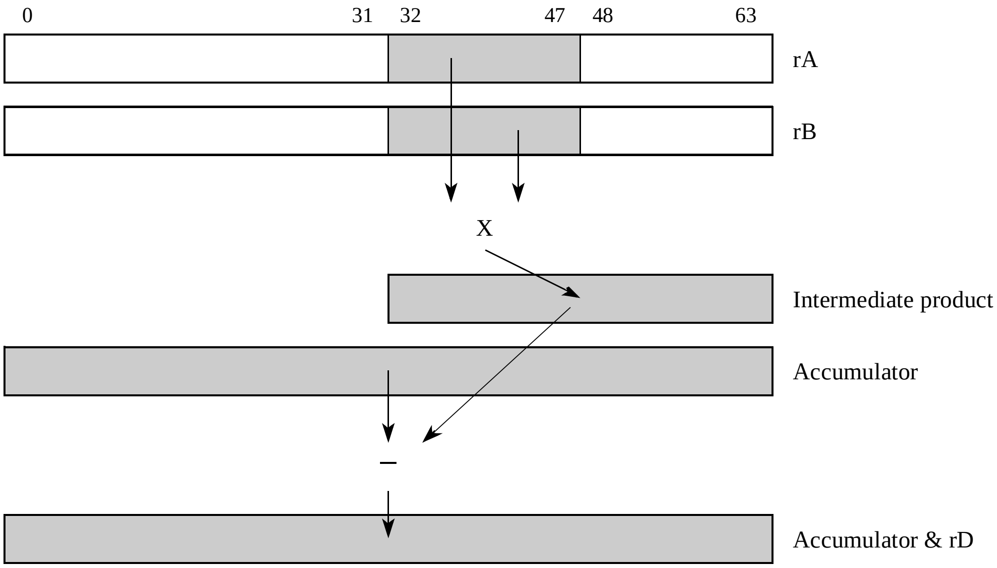


Figure 7-10. evmhegsmfan

evmhegsmiaa evmhegsmiaa

Multiply Half Words, Even, Guarded, Signed, Modulo, Integer, and Accumulate

evmhegsmiaa **r**D,**r**A,**r**B (O=0, F=0, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 0010 1001

```
$prod_{0:31} = rA_{32:47}$ *si $rB_{32:47}$
$temp1_{0:63} = EXTS(prod_{0:31})$
$temp2_{0:64} = ACC_{0:63} + temp1_{0:63}$
$rD_{0:63} = ACC_{0:63} = temp2_{1:64}$
```

The low even-numbered signed integer half-word element in **r**A is multiplied by the corresponding signed integer half-word element in **r**B. The intermediate product is sign-extended to 64 bits and added to the contents of the 64-bit accumulator to form a 65-bit intermediate sum. The lower 64 bits of the intermediate sum is placed back into the accumulator and also written into **r**D.

NOTE

This is a modulo sum. There is no check for overflow and no saturation is performed. An overflow from the 64-bit sum, if one occurs, is not recorded into SPEFSCR.

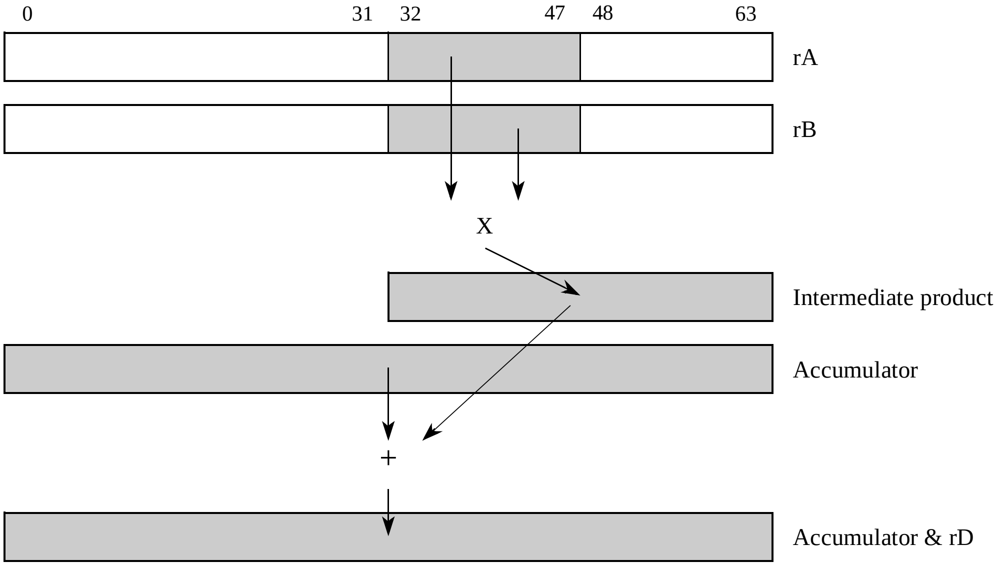


Figure 7-11. evmhegsmiaa

evmhegsmian evmhegsmian

Multiply Half Words, Even, Guarded, Signed, Modulo, Integer, and Accumulate Negative

evmhegsmian **r**D,**r**A,**r**B (O=0, F=0, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 1010 1001

```
$prod_{0:31} = rA_{32:47} *si\ rB_{32:47}$
$temp1_{0:63} = EXTS(prod_{0:31})$
$temp2_{0:64} = ACC_{0:63} - temp1_{0:63}$
$rD_{0:63} = ACC_{0:63} = temp2_{1:64}$
```

The low even-numbered signed integer half-word element in **r**A is multiplied by the corresponding signed integer half-word element in **r**B. The intermediate product is sign-extended to 64 bits and subtracted from the contents of the 64-bit accumulator to form a 65-bit intermediate difference. The lower 64 bits of the intermediate difference is placed back into the accumulator and also written into **r**D.

NOTE

This is a modulo difference. There is no check for overflow and no saturation is performed. An overflow from the 64-bit difference, if one occurs, is not recorded into SPEFSCR.

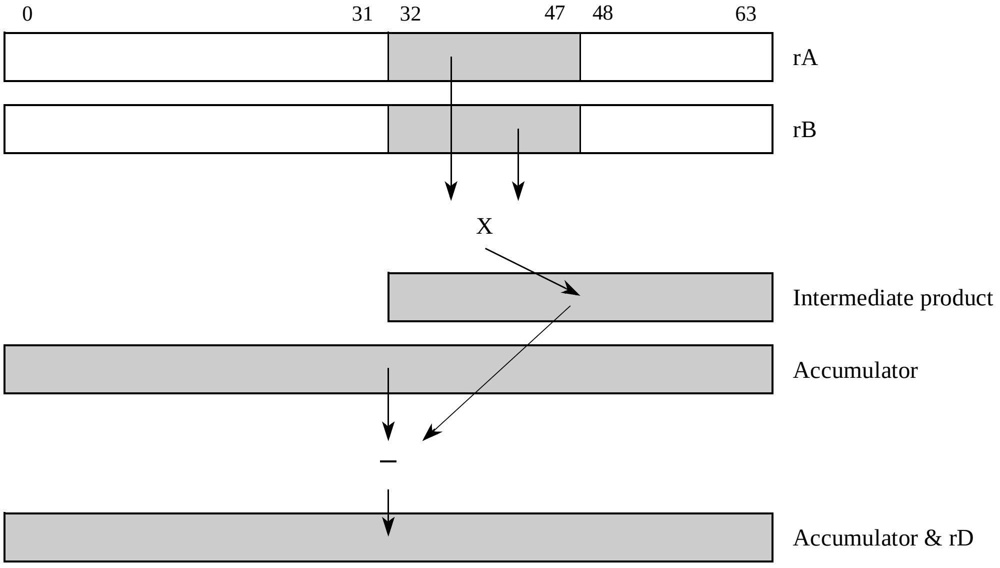


Figure 7-12. evmhegsmian

evmhegumiaa

Multiply Half Words, Even, Guarded, Unsigned, Modulo, Integer, and Accumulate

evmhegumiaa **r**D,**r**A,**r**B (O=0, F=0, S=0)

0 — 5	6 — 10	11 — 15	16 — 20	21 — 31
4	RD	RA	RB	101 0010 1000

```
$prod_{0:31} = rA_{32:47}$ *ui $rB_{32:47}$
$temp1_{0:63} = EXTZ(prod_{0:31})$
$temp2_{0:64} = ACC_{0:63} + temp1_{0:63}$
$rD_{0:63} = ACC_{0:63} = temp2_{1:64}$
```

The low even-numbered unsigned integer half-word element in **r**A is multiplied by the corresponding unsigned integer half-word element in **r**B. The intermediate product is sign-extended to 64 bits and added to the contents of the 64-bit accumulator to form a 65-bit intermediate sum. The lower 64 bits of the intermediate sum is placed back into the accumulator and also written into **r**D.

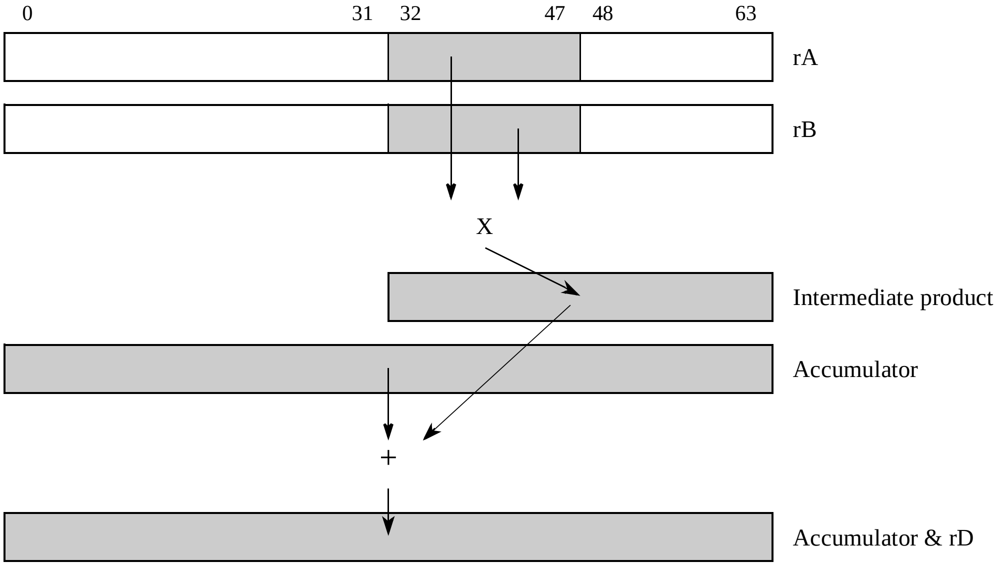


Figure 7-13. evmhegumiaa

evmhegumian evmhegumian

Multiply Half Words, Even, Guarded, Unsigned, Modulo, Integer, and Accumulate Negative

evmhegumian **r**D,**r**A,**r**B (O=0, F=0, S=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 1010 1000

$prod_{0:31}$ = $rA_{32:47}$ *ui $rB_{32:47}$

$temp1_{0:63}$ = EXTZ($prod_{0:31}$)

$temp2_{0:64}$ = $ACC_{0:63}$ - $temp1_{0:63}$

$rD_{0:63}$ = $ACC_{0:63}$ = $temp2_{1:64}$

The low even-numbered unsigned integer half word element in **r**A is multiplied by the corresponding unsigned integer half word element in **r**B. The intermediate product is sign-extended to 64 bits and subtracted from the contents of the 64-bit accumulator to form a 65-bit intermediate difference. The lower 64 bits of the intermediate difference is placed back into the accumulator and also written into **r**D.

NOTE

This is a modulo difference. There is no check for overflow and no saturation is performed. An overflow from the 64-bit difference, if one occurs, is not recorded into SPEFSCR.

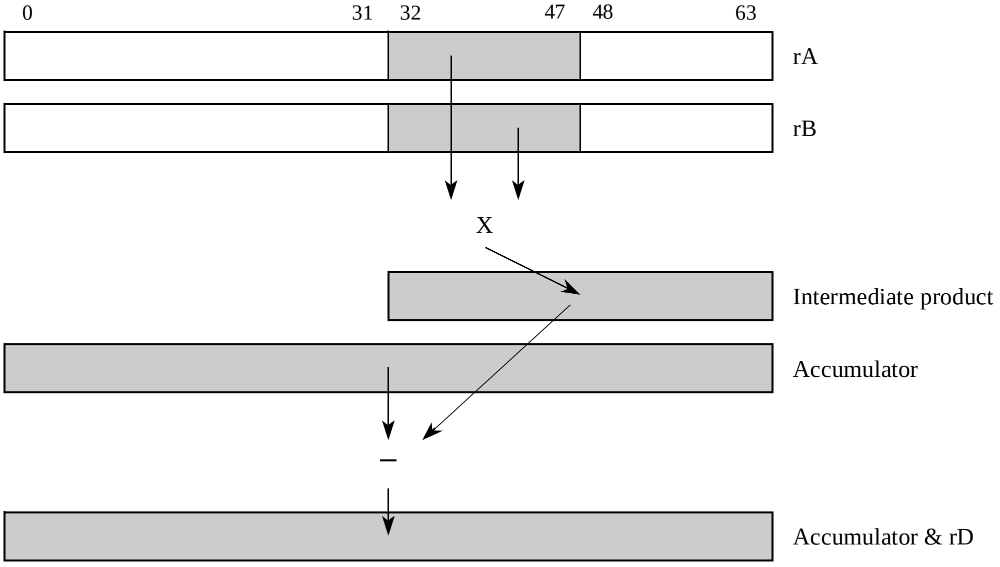


Figure 7-14. evmhegumian

evmhesmf evmhesmf

Vector Multiply Half Words, Even, Signed, Modulo, Fractional

evmhesmf **r**D,**r**A,**r**B (M=1, O=0, F=1, S=1, A=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 0000 1011

$prod_{0:31} = rA_{0:15} * rB_{0:15}$
$prod_{32:63} = rA_{32:47} * rB_{32:47}$

$temp1_{0:32} = prod_{0:31} \| 0$
$temp2_{0:32} = prod_{32:63} \| 0$

$rD_{0:31} = temp1_{1:32}$
$rD_{32:63} = temp2_{1:32}$

Each even-numbered signed fractional half-word element in **r**A is multiplied by the corresponding signed fractional half-word element in **r**B. The two 32-bit signed fractional products are shifted left by one bit to remove the redundant sign bit and then placed into the two word elements of **r**D.

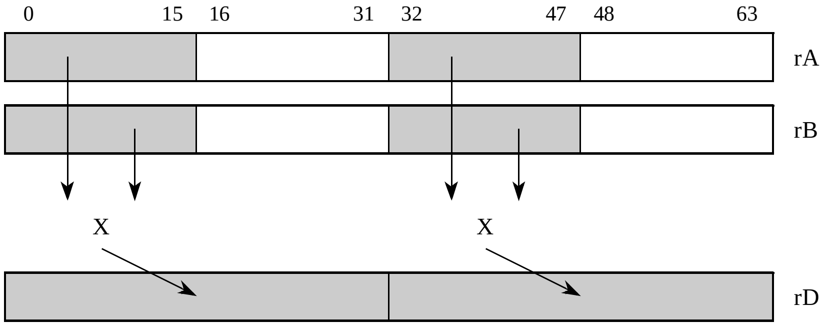


Figure 7-15. evmhesmf

evmhesmfa evmhesmfa

Vector Multiply Half Words, Even, Signed, Modulo, Fractional, to Accumulator

evmhesmfa **r**D,**r**A,**r**B (M=1, O=0, F=1, S=1, A=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 0010 1011

$\text{prod}_{0:31} = \text{rA}_{0:15} * \text{rB}_{0:15}$
$\text{prod}_{32:63} = \text{rA}_{32:47} * \text{rB}_{32:47}$

$\text{temp1}_{0:32} = \text{prod}_{0:31} \,||\, 0$
$\text{temp2}_{0:32} = \text{prod}_{32:63} \,||\, 0$

$\text{rD}_{0:31} = \text{temp1}_{1:32}$
$\text{rD}_{32:63} = \text{temp2}_{1:32}$

$\text{ACC}_{0:63} = \text{rD}_{0:63}$

Each even-numbered signed fractional half-word element in **r**A is multiplied by the corresponding signed fractional half-word element in **r**B. The two 32-bit signed fractional products are shifted left by one bit to remove the redundant sign bit and then placed into the two word elements of **r**D. The result in **r**D is also placed in the accumulator.

Other registers altered: ACC

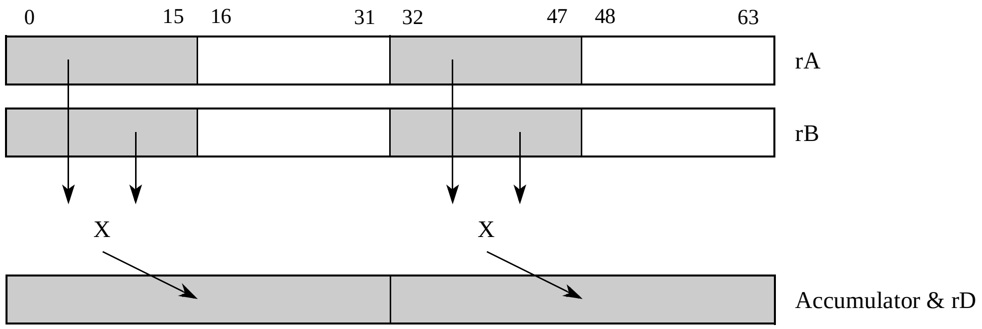


Figure 7-16. evmhesmfa

evmhesmfaaw   evmhesmfaaw

Vector Multiply Half Words, Even, Signed, Modulo, Fractional and Accumulate into Words

evmhesmfaaw **r**D,**r**A,**r**B (M=1, O=0, F=1, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 0000 1011

```
$temp1_{0:32} = (rA_{0:15} * rB_{0:15}) || 0$
$temp2_{0:32} = (rA_{32:47} * rB_{32:47}) || 0$

$temp3_{0:32} = ACC_{0:31} + temp1_{1:32}$
$temp4_{0:32} = ACC_{32:63} + temp2_{1:32}$

$ACC_{0:31} = rD_{0:31} = temp3_{1:32}$
$ACC_{32:63} = rD_{32:63} = temp4_{1:32}$
```

For each word element in the accumulator the following operations are performed in the order shown:

1. Each even-numbered signed fractional half word element in **r**A is multiplied by the corresponding signed fractional half word element in **r**B.
2. The intermediate 32-bit product is shifted left by one bit to remove the redundant sign bit.
3. It is added to the contents of the accumulator word to form a 33-bit intermediate sum.
4. The low 32 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word.

Other registers altered: ACC

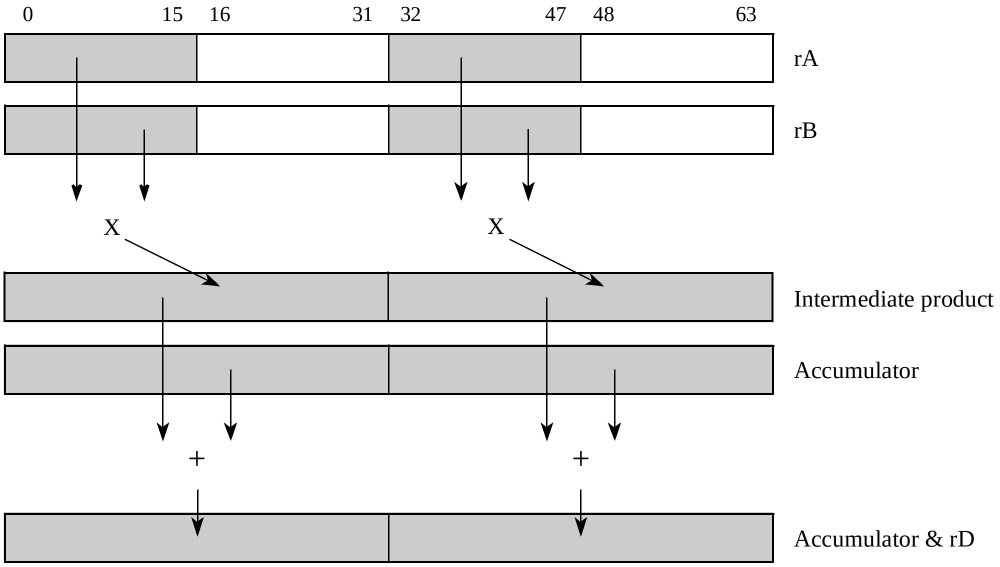


Figure 7-17. evmhesmfaaw

evmhesmfanw evmhesmfanw

Vector Multiply Half Words, Even, Signed, Modulo, Fractional and Accumulate Negative into Words

evmhesmfanw **r**D,**r**A,**r**B (M=1, O=0, F=1, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 1000 1011

$temp1_{0:32} = (rA_{0:15} * rB_{0:15}) \| 0$
$temp2_{0:32} = (rA_{32:47} * rB_{32:47}) \| 0$

$temp3_{0:32} = ACC_{0:31} - temp1_{1:32}$
$temp4_{0:32} = ACC_{32:63} - temp2_{1:32}$

$ACC_{0:31} = rD_{0:31} = temp3_{1:32}$
$ACC_{32:63} = rD_{32:63} = temp4_{1:32}$

For each word element in the accumulator the following operations are performed in the order shown:

1. Each even-numbered signed fractional half word element in **r**A is multiplied by the corresponding signed fractional half word element in **r**B.
2. The intermediate 32-bit product is shifted left by one bit to remove the redundant sign bit.
3. It is subtracted from the contents of the accumulator word to form a 33-bit intermediate difference.
4. The low 32 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word.

Other registers altered: ACC

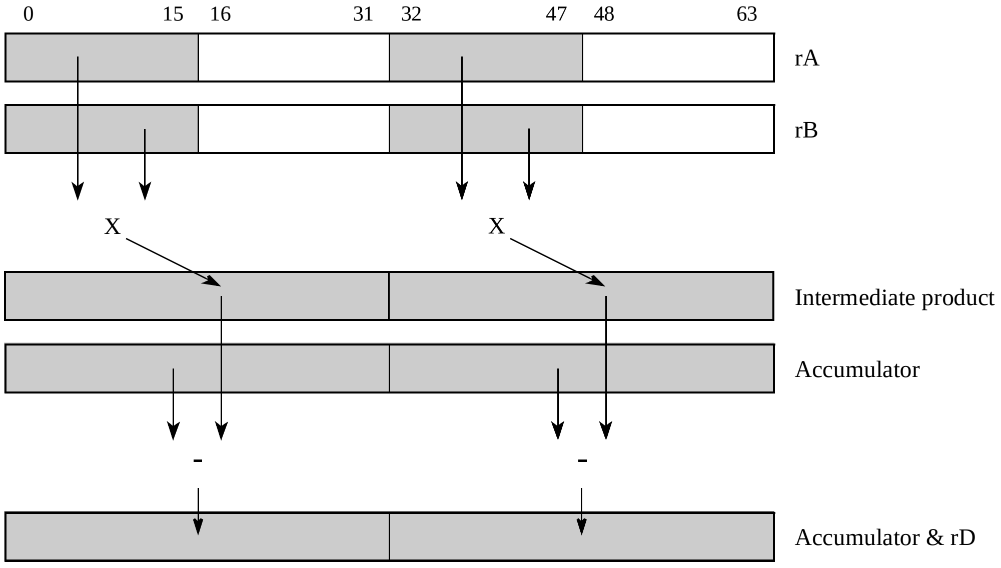


Figure 7-18. evmhesmfanw

evmhesmi

evmhesmi

Vector Multiply Half Words, Even, Signed, Modulo, Integer

evmhesmi **r**D,**r**A,**r**B (M=1, O=0, F=0, S=1, A=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 0000 1001

$rD_{0:31} = rA_{0:15}$ *si $rB_{0:15}$
$rD_{32:63} = rA_{32:47}$ *si $rB_{32:47}$

Each even-numbered signed integer half-word element in **r**A is multiplied by the corresponding signed integer half-word element in **r**B. The two 32-bit signed integer products are placed into the two word elements of **r**D.

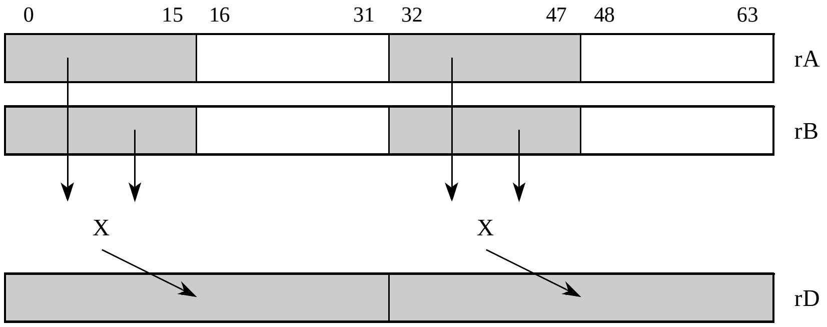


Figure 7-19. evmhesmi

evmhesmia

Vector Multiply Half Words, Even, Signed, Modulo, Integer, to Accumulator

evmhesmia **r**D,**r**A,**r**B (M=1, O=0, F=0, S=1, A=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 0010 1001

$rD_{0:31} = rA_{0:15} \text{ *si } rB_{0:15}$
$rD_{32:63} = rA_{32:47} \text{ *si } rB_{32:47}$

$ACC_{0:63} = rD_{0:63}$

Each even-numbered signed integer half-word element in **r**A is multiplied by the corresponding signed integer half-word element in **r**B. The two 32-bit signed integer products are placed into the two word elements of **r**D. The result in **r**D is also placed in the accumulator.

Other registers altered: ACC

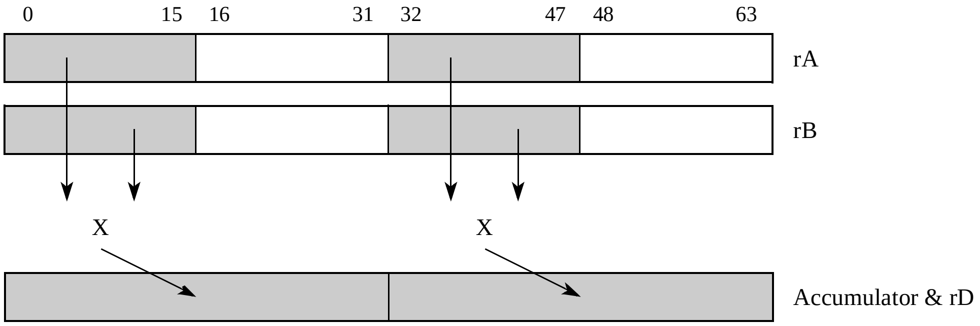


Figure 7-20. evmhesmia

evmhesmiaaw evmhesmiaaw

Vector Multiply Half Words, Even, Signed, Modulo, Integer and Accumulate into Words

evmhesmiaaw **r**D,**r**A,**r**B (M=1, O=0, F=0, S=1)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	101 0000 1001

$temp1_{0:31} = rA_{0:15} \text{ *si } rB_{0:15}$
$temp2_{0:31} = rA_{32:47} \text{ *si } rB_{32:47}$

$temp3_{0:32} = ACC_{0:31} + temp1_{0:31}$
$temp4_{0:32} = ACC_{32:63} + temp2_{0:31}$

$ACC_{0:31} = rD_{0:31} = temp3_{1:32}$
$ACC_{32:63} = rD_{32:63} = temp4_{1:32}$

For each word element in the accumulator the following operations are performed in the order shown:

1. Each even-numbered signed integer half word element in **r**A is multiplied by the corresponding signed integer half word element in **r**B.
2. The intermediate 32-bit product is added to the contents of the accumulator word to form a 33-bit intermediate sum.
3. The low 32 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word.

Other registers altered: ACC

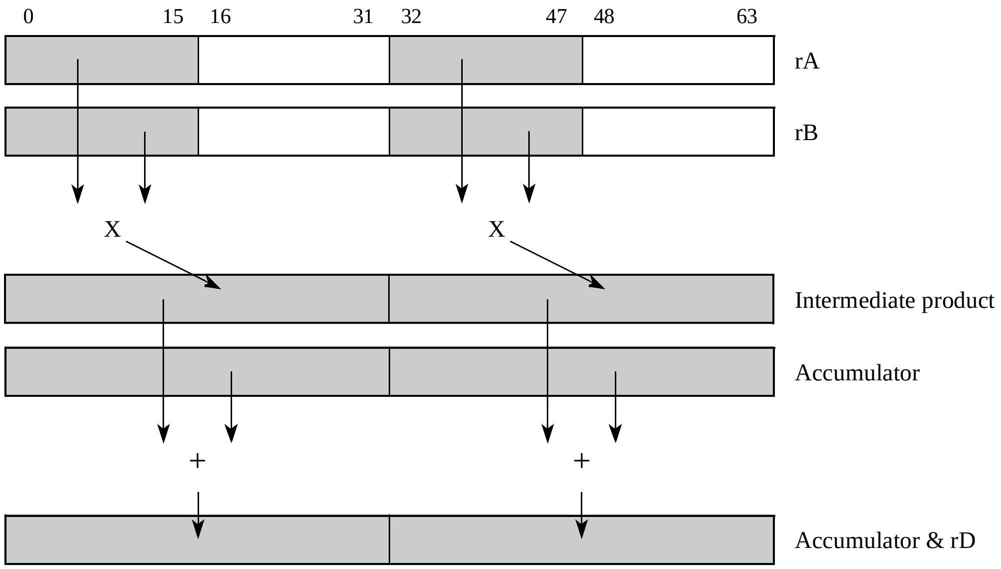


Figure 7-21. evmhesmiaaw

evmhesmianw evmhesmianw

Vector Multiply Half Words, Even, Signed, Modulo, Integer and Accumulate Negative into Words

evmhesmianw **r**D,**r**A,**r**B (M=1, O=0, F=0, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 1000 1001

$temp1_{0:31} = rA_{0:15}$ *si $rB_{0:15}$
$temp2_{0:31} = rA_{32:47}$ *si $rB_{32:47}$

$temp3_{0:32} = ACC_{0:31} - temp1_{0:31}$
$temp4_{0:32} = ACC_{32:63} - temp2_{0:31}$

$ACC_{0:31} = rD_{0:31} = temp3_{1:32}$
$ACC_{32:63} = rD_{32:63} = temp4_{1:32}$
For each word element in the accumulator the following operations are performed in the order shown:

Each even-numbered signed integer half-word element in **r**A is multiplied by the corresponding signed integer half-word element in **r**B.

The intermediate 32-bit product is subtracted from the contents of the accumulator word to form a 33-bit intermediate difference. The low 32 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word.

Other registers altered: ACC

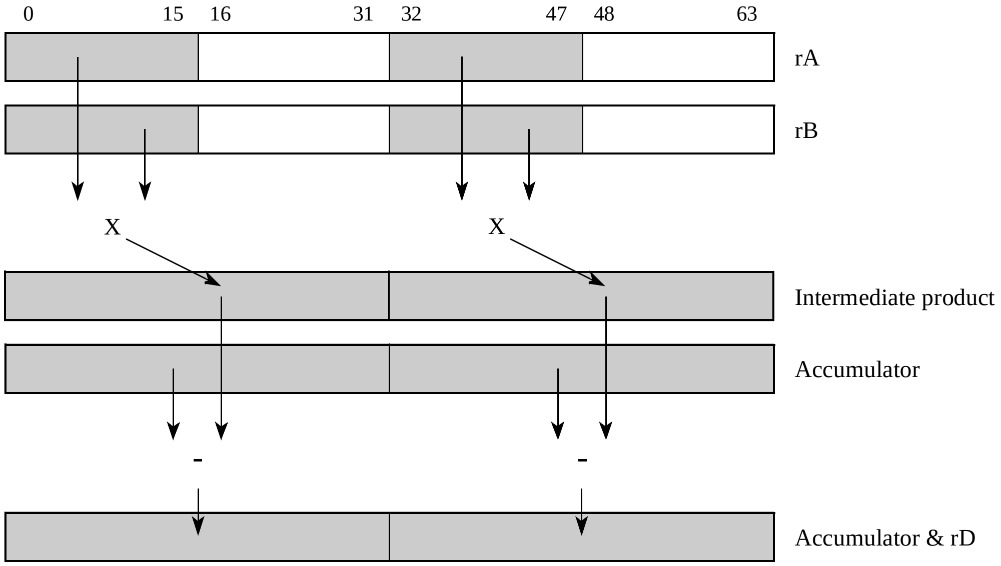


Figure 7-22. evmhesmianw

evmhessf evmhessf

Vector Multiply Half Words, Even, Signed, Saturate, Fractional

evmhessf **r**D,**r**A,**r**B (M=0, O=0, F=1, S=1, A=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 0000 0011

```
$temp1_{0:32} = (rA_{0:15} * rB_{0:15}) || 0$
$temp2_{0:32} = (rA_{32:47} * rB_{32:47}) || 0$

$movh = temp1_0 \oplus temp1_1$
$movl = temp2_0 \oplus temp2_1$

$rD_{0:31} = SATURATE(movh, 0x7FFFFFFF, temp1_{1:32})$
$rD_{32:63} = SATURATE(movl, 0x7FFFFFFF, temp2_{1:32})$

$SPEFSCR_{OVH} = movh$
$SPEFSCR_{OV} = movl$
$SPEFSCR_{SOVH} = SPEFSCR_{SOVH} | movh$
$SPEFSCR_{SOV} = SPEFSCR_{SOV} | movl$
```

Each even-numbered signed fractional half-word element in **r**A is multiplied by the corresponding signed fractional half-word element in **r**B. The two 32-bit signed fractional products are shifted left one bit to eliminate the redundant sign bit and then placed into the two word elements of **r**D. If the inputs are –1.0 and –1.0 the result is saturated to the most positive signed fraction (0x7FFF_FFFF). If saturation occurs, the appropriate overflow and summary overflow bits are recorded in SPEFSCR.

Other registers altered: SPEFSCR

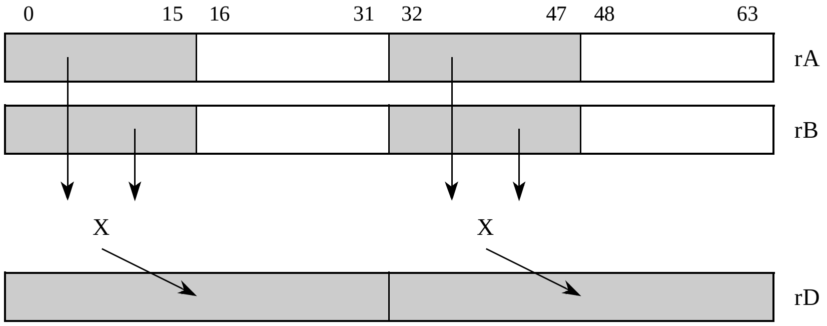


Figure 7-23. evmhessf

evmhessfa evmhessfa

Vector Multiply Half Words, Even, Signed, Saturate, Fractional to Accumulator

evmhessfa **r**D,**r**A,**r**B (M=0, O=0, F=1, S=1, A=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 0010 0011

```
$temp1_{0:32} = (rA_{0:15} * rB_{0:15}) || 0$
$temp2_{0:32} = (rA_{32:47} * rB_{32:47}) || 0$

$movh = temp1_0 \oplus temp1_1$
$movl = temp2_0 \oplus temp2_1$

$rD_{0:31} = SATURATE(movh, 0x7FFFFFFF, temp1_{1:32})$
$rD_{32:63} = SATURATE(movl, 0x7FFFFFFF, temp2_{1:32})$

$ACC_{0:63} = rD_{0:63}$

$SPEFSCR_{OVH} = movh$
$SPEFSCR_{OV} = movl$
$SPEFSCR_{SOVH} = SPEFSCR_{SOVH} | movh$
$SPEFSCR_{SOV} = SPEFSCR_{SOV} | movl$
```

Each even-numbered signed fractional half-word element in **r**A is multiplied by the corresponding signed fractional half-word element in **r**B. The two 32-bit signed fractional products are shifted left one bit to eliminate the redundant sign bit and then placed into the two word elements of **r**D. If the inputs are –1.0 and –1.0 the result is saturated to the most positive signed fraction (0x7FFF_FFFF). The result in **r**D is also placed in the accumulator. If saturation occurs, the appropriate overflow and summary overflow bits are recorded in SPEFSCR.

Other registers altered: SPEFSCR, ACC

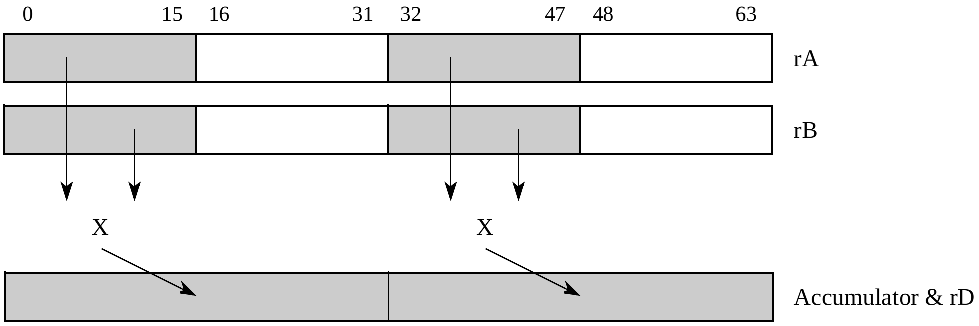


Figure 7-24. evmhessfa

evmhessfaaw

evmhessfaaw

Vector Multiply Half Words, Even, Signed, Saturate, Fractional, and Accumulate into Words

evmhessfaaw **r**D,**r**A,**r**B (M=0, O=0, F=1, S=1)

0 … 5	6 … 10	11 … 15	16 … 20	21 … 31
4	RD	RA	RB	101 0000 0011

```
$temp1_{0:32} = (rA_{0:15} * rB_{0:15}) \| 0$
$temp2_{0:32} = (rA_{32:47} * rB_{32:47}) \| 0$

$movh = temp1_0 \oplus temp1_1$
$movl = temp2_0 \oplus temp2_1$

$temp3_{0:31} = SATURATE(movh, 0x7FFFFFFF, temp1_{1:32})$
$temp4_{0:31} = SATURATE(movl, 0x7FFFFFFF, temp2_{1:32})$

$temp5_{0:32} = \{ACC_0,ACC_{0:31}\} + \{temp3_0,temp3_{0:31}\}$
$temp6_{0:32} = \{ACC_{32},ACC_{32:63}\} + \{temp4_0,temp4_{0:31}\}$

$ovh = temp5_0 \oplus temp5_1$
$ovl = temp6_0 \oplus temp6_1$

$rD_{0:31} = SATURATE\_ACC(ovh, temp5_0, 0x80000000, 0x7FFFFFFF, temp5_{1:32})$
$rD_{32:63} = SATURATE\_ACC(ovl, temp6_0, 0x80000000, 0x7FFFFFFF, temp6_{1:32})$

$ACC_{0:31} = rD_{0:31}$
$ACC_{32:63} = rD_{32:63}$

$SPEFSCR_{OVH} = movh | ovh$
$SPEFSCR_{OV} = movl | ovl$
$SPEFSCR_{SOVH} = SPEFSCR_{SOVH} | movh | ovh$
$SPEFSCR_{SOV} = SPEFSCR_{SOV} | movl | ovl$
```

For each word element in the accumulator the following operations are performed in the order shown:

Each even-numbered signed fractional half-word element in **r**A is multiplied by the corresponding signed fractional half-word element in **r**B. The two 32-bit signed fractional products are shifted left one bit to eliminate the redundant sign bit. If the inputs are –1.0 and –1.0 the intermediate result is saturated to the most positive signed fraction (0x7FFF_FFFF).

The intermediate 32-bit product is added to the contents of the accumulator word to form an intermediate sum. If the intermediate sum has overflowed, the appropriate saturation value (0x7FFF_FFFF if positive overflow or 0x8000_0000 if negative overflow) is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word.

If there is an overflow from either the multiply or the addition, the overflow information is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR, ACC

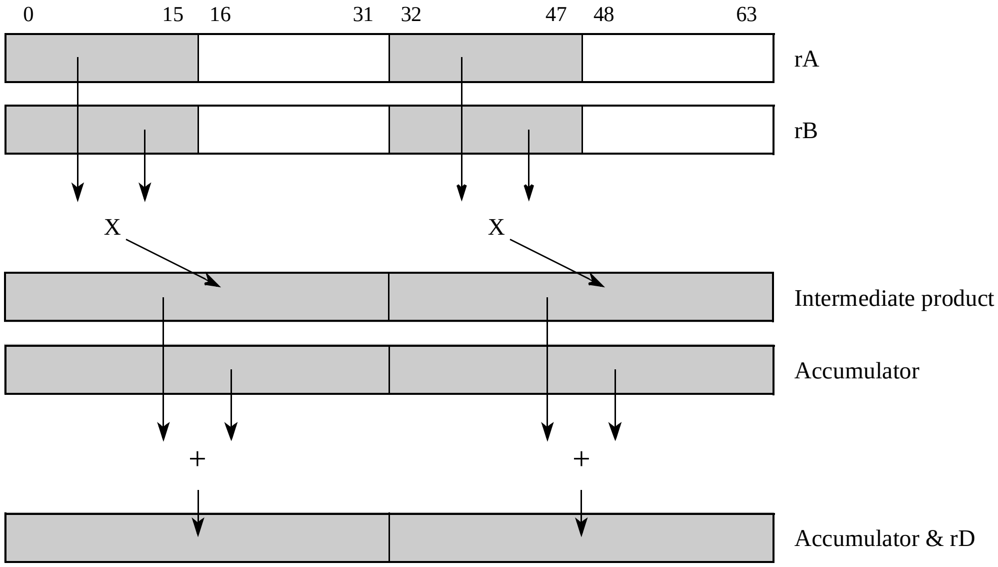


Figure 7-25. evmhessfaaw

evmhessfanw evmhessfanw

Vector Multiply Half Words, Even, Signed, Saturate, Fractional, and Accumulate Negative into Words

evmhessfanw **r**D,**r**A,**r**B (M=0, O=0, F=1, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 1000 0011

```
$temp1_{0:32} = (rA_{0:15} * rB_{0:15}) \| 0$
$temp2_{0:32} = (rA_{32:47} * rB_{32:47}) \| 0$

$movh = temp1_0 \oplus temp1_1$
$movl = temp2_0 \oplus temp2_1$

$temp3_{0:31} = SATURATE(movh, 0x7FFFFFFF, temp1_{1:32})$
$temp4_{0:31} = SATURATE(movl, 0x7FFFFFFF, temp2_{1:32})$

$temp5_{0:32} = \{ACC_0,ACC_{0:31}\} - \{temp3_0,temp3_{0:31}\}$
$temp6_{0:32} = \{ACC_{32},ACC_{32:63}\} - \{temp4_0,temp4_{0:31}\}$

$ovh = temp5_0 \oplus temp5_1$
$ovl = temp6_0 \oplus temp6_1$

$rD_{0:31} = SATURATE\_ACC(ovh, temp5_0, 0x80000000, 0x7FFFFFFF, temp5_{1:32})$
$rD_{32:63} = SATURATE\_ACC(ovl, temp6_0, 0x80000000, 0x7FFFFFFF, temp6_{1:32})$

$ACC_{0:31} = rD_{0:31}$
$ACC_{32:63} = rD_{32:63}$

$SPEFSCR_{OVH} = movh | ovh$
$SPEFSCR_{OV} = movl | ovl$

$SPEFSCR_{SOVH} = SPEFSCR_{SOVH} | movh | ovh$
$SPEFSCR_{SOV} = SPEFSCR_{SOV} | movl | ovl$
```

For each word element in the accumulator the following operations are performed in the order shown:

Each even-numbered signed fractional half-word element in **r**A is multiplied by the corresponding signed fractional half word element in **r**B. The two 32-bit signed fractional products are shifted left one bit to eliminate the redundant sign bit. If the inputs are –1.0 and –1.0 the intermediate result is saturated to the most positive signed fraction (0x7FFF_FFFF).

The intermediate 32-bit product is subtracted from the contents of the accumulator word to form an intermediate sum. If the intermediate difference has overflowed, the appropriate saturation value (0x7FFF_FFFF if positive overflow or 0x8000_0000 if negative overflow) is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word.

If there is an overflow from either the multiply or the addition, the overflow information is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR, ACC

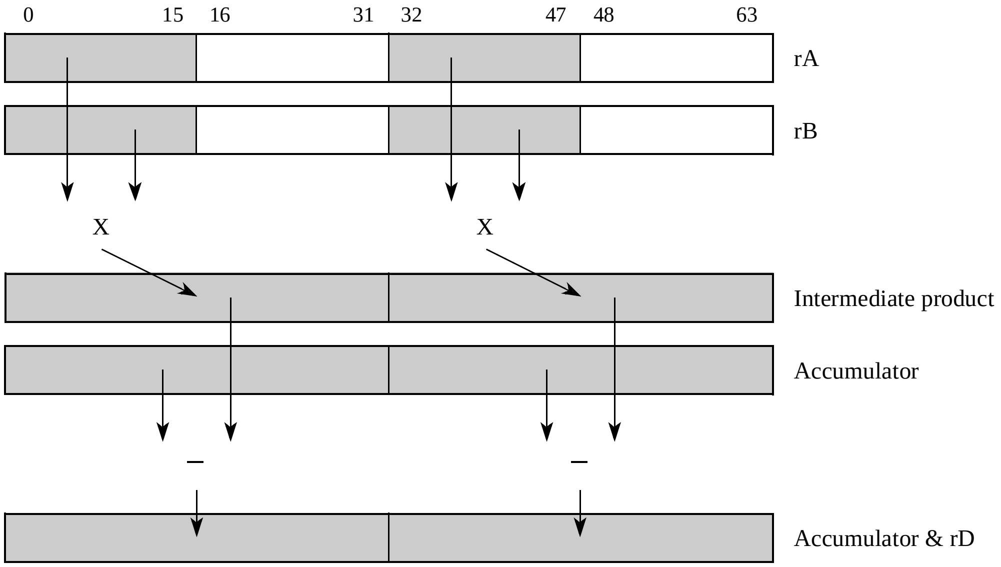


Figure 7-26. evmhessfanw

evmhessiaaw evmhessiaaw

Vector Multiply Half Words, Even, Signed, Saturate, Integer, and Accumulate into Words

evmhessiaaw **r**D,**r**A,**r**B (M=0, O=0, F=0, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 0000 0001

```
$temp1_{0:31} = rA_{0:15}$ *si $rB_{0:15}$
$temp2_{0:31} = rA_{32:47}$ *si $rB_{32:47}$

$temp3_{0:32} = \{ACC_0,ACC_{0:31}\} + \{temp1_0,temp1_{0:31}\}$
$temp4_{0:32} = \{ACC_{32},ACC_{32:63}\} + \{temp2_0,temp2_{0:31}\}$

$ovh = temp3_0 \oplus temp3_1$
$ovl = temp4_0 \oplus temp4_1$

$rD_{0:31}$ = SATURATE_ACC(ovh, $temp3_0$, 0x80000000, 0x7FFFFFFF, $temp3_{1:32}$)
$rD_{32:63}$ = SATURATE_ACC(ovl, $temp4_0$, 0x80000000, 0x7FFFFFFF, $temp4_{1:32}$)

$ACC_{0:31} = rD_{0:31}$
$ACC_{32:63} = rD_{32:63}$

$SPEFSCR_{OVH} = ovh$
$SPEFSCR_{OV} = ovl$
$SPEFSCR_{SOVH} = SPEFSCR_{SOVH} \mid ovh$
$SPEFSCR_{SOV} = SPEFSCR_{SOV} \mid ovl$
```

For each word element in the accumulator the following operations are performed in the order shown:

Each even-numbered signed integer half-word element in **r**A is multiplied by the corresponding signed integer half-word element in **r**B.

The intermediate 32-bit product is added to the contents of the accumulator word to form an intermediate sum. If the intermediate sum has overflowed, the appropriate saturation value (0x7FFF_FFFF if positive overflow or 0x8000_0000 if negative overflow) is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word.

If there is an overflow from the addition, the overflow information is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR, ACC

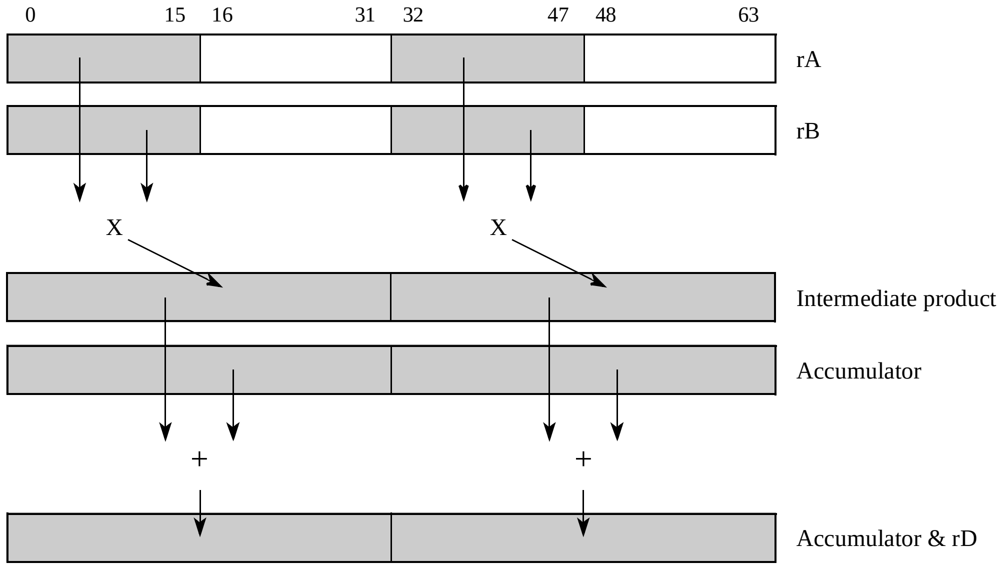


Figure 7-27. Even Form of Vector half word Multiply (evmhessiaaw)

evmhessianw evmhessianw

Vector Multiply Half Words, Even, Signed, Saturate, Integer and Accumulate Negative into Words

evmhessianw **r**D,**r**A,**r**B (M=0, O=0, F=0, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 1000 0001

```
$temp1_{0:31} = rA_{0:15}$ *si $rB_{0:15}$
$temp2_{0:31} = rA_{32:47}$ *si $rB_{32:47}$

$temp3_{0:32} = \{ACC_0, ACC_{0:31}\} - \{temp1_0, temp1_{0:31}\}$
$temp4_{0:32} = \{ACC_{32}, ACC_{32:63}\} - \{temp2_0, temp2_{0:31}\}$

$ovh = temp3_0 \oplus temp3_1$
$ovl = temp4_0 \oplus temp4_1$

$rD_{0:31}$ = SATURATE_ACC(ovh, $temp3_0$, 0x80000000, 0x7FFFFFFF, $temp3_{1:32}$)
$rD_{32:63}$ = SATURATE_ACC(ovl, $temp4_0$, 0x80000000, 0x7FFFFFFF, $temp4_{1:32}$)

$ACC_{0:31} = rD_{0:31}$
$ACC_{32:63} = rD_{32:63}$

$SPEFSCR_{OVH} = ovh$
$SPEFSCR_{OV} = ovl$
$SPEFSCR_{SOVH} = SPEFSCR_{SOVH} | ovh$
$SPEFSCR_{SOV} = SPEFSCR_{SOV} | ovl$
```

For each word element in the accumulator, the following operations are performed in the order shown:

1. Each even-numbered signed integer half word element in **r**A is multiplied by the corresponding signed integer half word element in **r**B.
2. The intermediate 32-bit product is subtracted from the contents of the accumulator word to form an 33-bit intermediate difference.
3. If the intermediate difference has overflowed, the appropriate saturation value (0x7FFF_FFFF if positive overflow or 0x8000_0000 if negative overflow) is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word.
4. If there is an overflow from the subtraction, the overflow information is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR, ACC

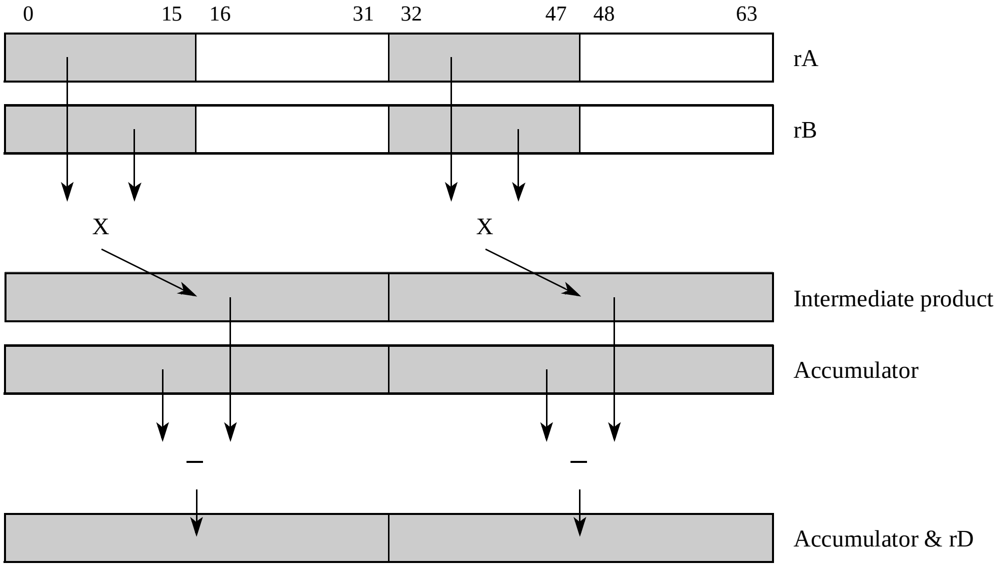


Figure 7-28. evmhessianw

evmheumi evmheumi

Vector Multiply Half Words, Even, Unsigned, Modulo, Integer

evmheumi **r**D,**r**A,**r**B (M=1, O=0, F=0, S=0, A=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 0000 1000

$rD_{0:31} = rA_{0:15}$ *ui $rB_{0:15}$
$rD_{32:63} = rA_{32:47}$ *ui $rB_{32:47}$

Each even-numbered unsigned integer half-word element in **r**A is multiplied by the corresponding unsigned integer half-word element in **r**B. The two 32-bit unsigned integer products are placed into the two word elements of **r**D.

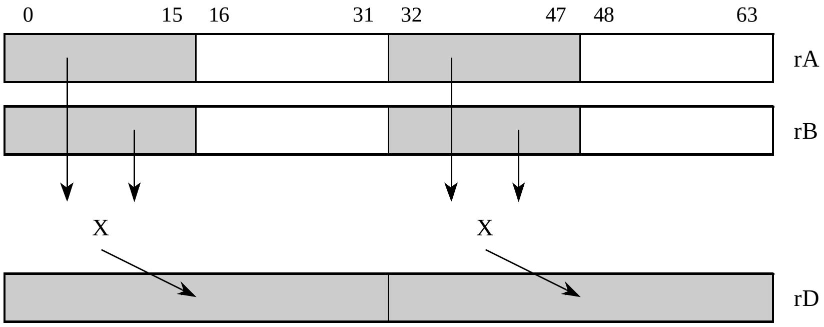


Figure 7-29. evmheumi—Even Multiply of Two Unsigned Modulo Integer Elements

evmheumia evmheumia

Vector Multiply Half Words, Even, Unsigned, Modulo, Integer, to Accumulator

evmheumia **r**D,**r**A,**r**B (M=1, O=0, F=0, S=0, A=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 0010 1000

$rD_{0:31} = rA_{0:15}$ *ui $rB_{0:15}$
$rD_{32:63} = rA_{32:47}$ *ui $rB_{32:47}$

$ACC_{0:63} = rD_{0:63}$

Each even-numbered unsigned integer half-word element in **r**A is multiplied by the corresponding unsigned integer half-word element in **r**B. The two 32-bit unsigned integer products are placed into the two word elements of **r**D. The result in **r**D is also placed in the accumulator.

Other registers altered: ACC

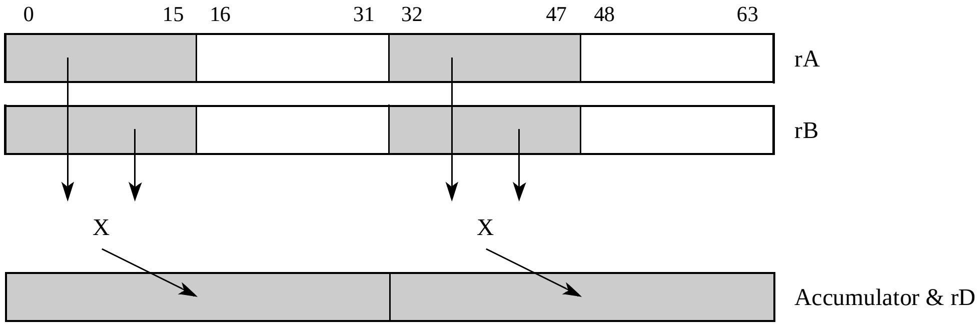


Figure 7-30. evmheumia

evmheumiaaw evmheumiaaw

Vector Multiply Half Words, Even, Unsigned, Modulo, Integer and Accumulate into Words

evmheumiaaw **r**D,**r**A,**r**B (M=1, O=0, F=0, S=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 0000 1000

```
$temp1_{0:31} = rA_{0:15}$ *ui $rB_{0:15}$
$temp2_{0:31} = rA_{32:47}$ *ui $rB_{32:47}$

$temp3_{0:32} = ACC_{0:31} + temp1_{0:31}$
$temp4_{0:32} = ACC_{32:63} + temp2_{0:31}$

$ACC_{0:31} = rD_{0:31} = temp3_{1:32}$
$ACC_{32:63} = rD_{32:63} = temp4_{1:32}$
```

For each word element in the accumulator the following operations are performed in the order shown:

1. Each even-numbered unsigned integer half word element in **r**A is multiplied by the corresponding unsigned integer half word element in **r**B.
2. The intermediate 32-bit product is added to the contents of the accumulator word to form a 33-bit intermediate sum.
3. The low 32 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word.

Other registers altered: ACC

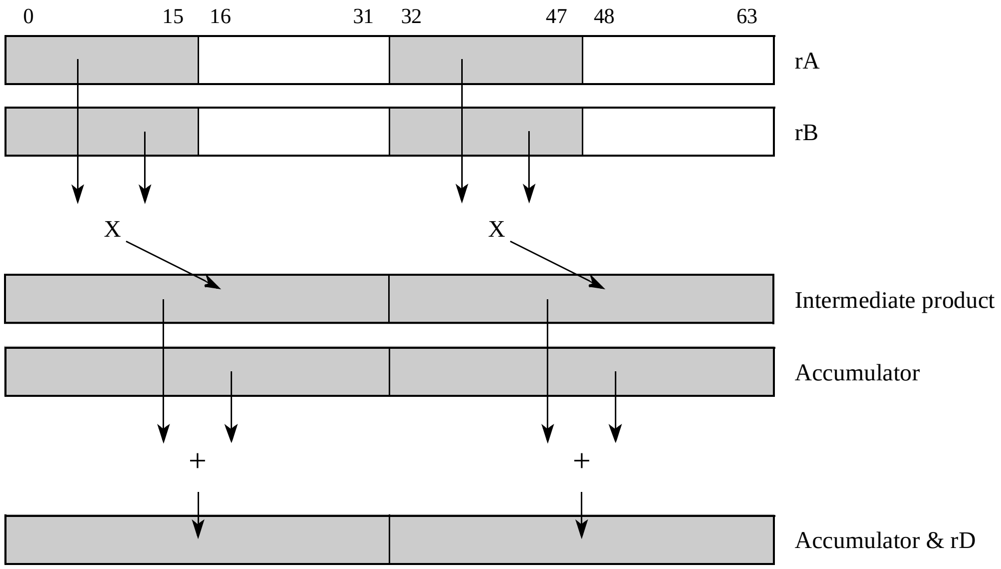


Figure 7-31. evmheumiaaw

evmheumianw   evmheumianw

Vector Multiply Half Words, Even, Unsigned, Modulo, Integer and Accumulate Negative into Words

evmheumianw **r**D,**r**A,**r**B (M=1, O=0, F=0, S=0)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	101 1000 1000

$temp1_{0:31} = rA_{0:15}$ *ui $rB_{0:15}$
$temp2_{0:31} = rA_{32:47}$ *ui $rB_{32:47}$

$temp3_{0:32} = ACC_{0:31} - temp1_{0:31}$
$temp4_{0:32} = ACC_{32:63} - temp2_{0:31}$

$ACC_{0:31} = rD_{0:31} = temp3_{1:32}$
$ACC_{32:63} = rD_{32:63} = temp4_{1:32}$

For each word element in the accumulator the following operations are performed in the order shown:

1. Each even-numbered unsigned integer half word element in **r**A is multiplied by the corresponding unsigned integer half word element in **r**B.
2. The intermediate 32-bit product is subtracted from the contents of the accumulator word to form a 33-bit intermediate difference.
3. The low 32 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word.

Other registers altered: ACC

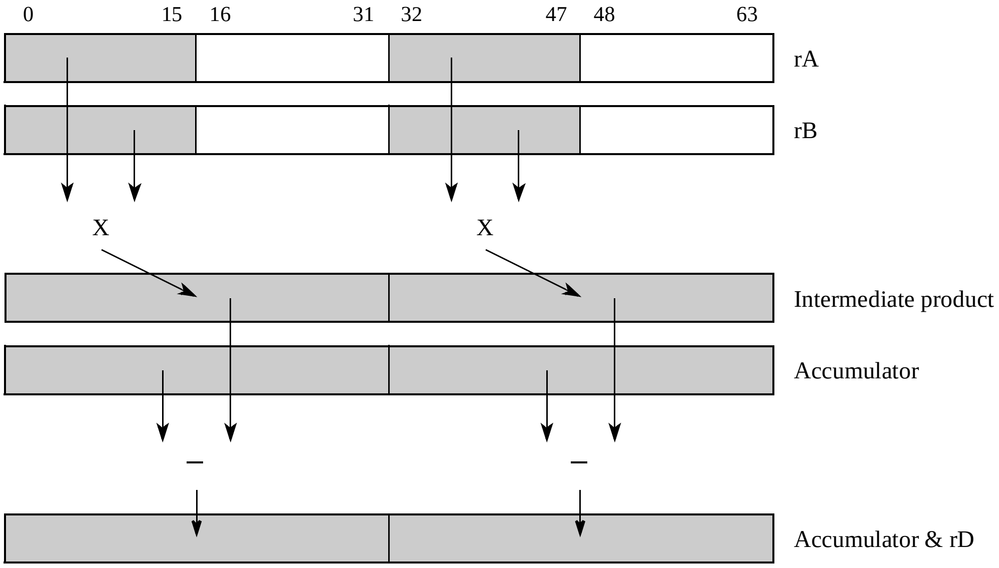


Figure 7-32. evmheumianw

evmheusiaaw evmheusiaaw

Vector Multiply Half Words, Even, Unsigned, Saturate, Integer and Accumulate into Words

evmheusiaaw **r**D,**r**A,**r**B (M=0, O=0, F=0, S=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 0000 0000

```
$temp1_{0:31}$ = $rA_{0:15}$ *ui $rB_{0:15}$
$temp2_{0:31}$ = $rA_{32:47}$ *ui $rB_{32:47}$

$temp3_{0:32}$ = $ACC_{0:31}$ + $temp1_{0:31}$
$temp4_{0:32}$ = $ACC_{32:63}$ + $temp2_{0:31}$

ovh = $temp3_0$
ovl = $temp4_0$

$rD_{0:31}$ = SATURATE_ACC(ovh, 0xFFFFFFFF, $temp3_{1:32}$)
$rD_{32:63}$ = SATURATE_ACC(ovl, 0xFFFFFFFF, $temp4_{1:32}$)

$ACC_{0:31}$ = $rD_{0:31}$
$ACC_{32:63}$ = $rD_{32:63}$

$SPEFSCR_{OVH}$ = ovh
$SPEFSCR_{OV}$ = ovl
$SPEFSCR_{SOVH}$ = $SPEFSCR_{SOVH}$ | ovh
$SPEFSCR_{SOV}$ = $SPEFSCR_{SOV}$ | ovl
```

For each word element in the accumulator the following operations are performed in the order shown:

1. Each even-numbered unsigned integer half word element in **r**A is multiplied by the corresponding unsigned integer half word element in **r**B.
2. The intermediate 32-bit product is added to the contents of the accumulator word to form a 33-bit intermediate sum.
3. If the intermediate sum has overflowed, the saturation value 0xFFFF_FFFF is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word.
4. If there is an overflow from the addition, the overflow information is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR, ACC

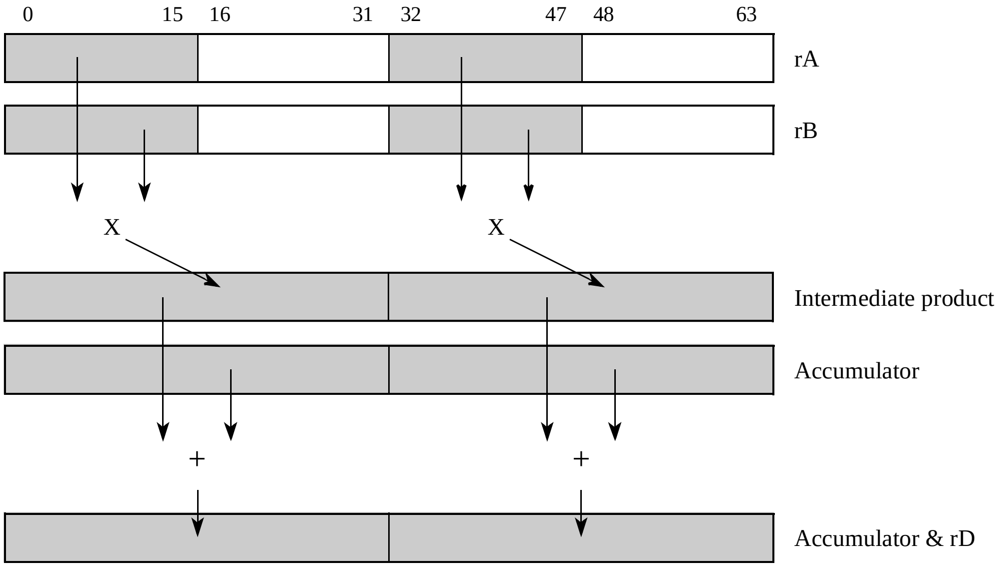


Figure 7-33. evmheusiaaw

evmheusianw evmheusianw

Vector Multiply Half Words, Even, Unsigned, Saturate, Integer, and Accumulate Negative into Words

evmheusianw **r**D,**r**A,**r**B (M=0, O=0, F=0, S=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 1000 0000

```
$temp1_{0:31}$ = $rA_{0:15}$ *ui $rB_{0:15}$
$temp2_{0:31}$ = $rA_{32:47}$ *ui $rB_{32:47}$

$temp3_{0:32}$ = $ACC_{0:31}$ - $temp1_{0:31}$
$temp4_{0:32}$ = $ACC_{32:63}$ - $temp2_{0:31}$

ovh = $temp3_0$
ovl = $temp4_0$

$rD_{0:31}$ = SATURATE_ACC(ovh, 0x00000000, $temp3_{1:32}$)
$rD_{32:63}$ = SATURATE_ACC(ovl, 0x00000000, $temp4_{1:32}$)

$ACC_{0:31}$ = $rD_{0:31}$
$ACC_{32:63}$ = $rD_{32:63}$

$SPEFSCR_{OVH}$ = ovh
$SPEFSCR_{OV}$ = ovl
$SPEFSCR_{SOVH}$ = $SPEFSCR_{SOVH}$ | ovh
$SPEFSCR_{SOV}$ = $SPEFSCR_{SOV}$ | ovl
```

For each word element in the accumulator the following operations are performed in the order shown:

1. Each even-numbered unsigned integer half word element in **r**A is multiplied by the corresponding unsigned integer half word element in **r**B.
2. The intermediate 32-bit product is subtracted from the contents of the accumulator word to form a 33-bit intermediate difference.
3. If the intermediate difference has underflowed (is negative), the saturation value 0x0000_0000 is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word.
4. If there is an underflow from the subtraction, the underflow information is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR, ACC

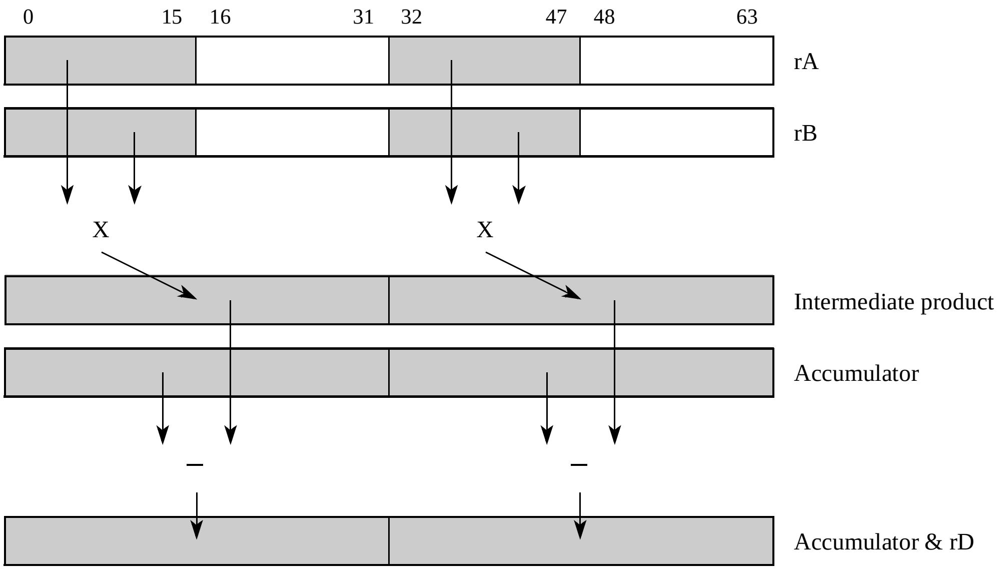


Figure 7-34. evmheusianw

evmhogsmfaa evmhogsmfaa

Multiply Half Words, Odd, Guarded, Signed, Modulo, Fractional and Accumulate

evmhogsmfaa **r**D,**r**A,**r**B (O=1, F=1, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 0010 1111

$prod_{0:31} = rA_{48:63} * rB_{48:63}$

$temp1_{0:63} = EXTS(prod_{0:31} \| 0)$

$temp2_{0:64} = ACC_{0:63} + temp1_{0:63}$

$rD_{0:63} = ACC_{0:63} = temp2_{1:64}$

The low odd-numbered signed fractional half-word element in **r**A is multiplied by the corresponding signed fractional half-word element in **r**B. The 32-bit intermediate product is sign-extended to 64 bits. Then it is shifted left by one bit and added to the contents of the 64-bit accumulator to form a 65-bit intermediate sum. The lower 64 bits of the intermediate sum is placed back into the accumulator and also written into **r**D.

NOTE

This is a modulo sum. There is no check for overflow and no saturation is performed. An overflow from the 64-bit sum, if one occurs, is not recorded into SPEFSCR.

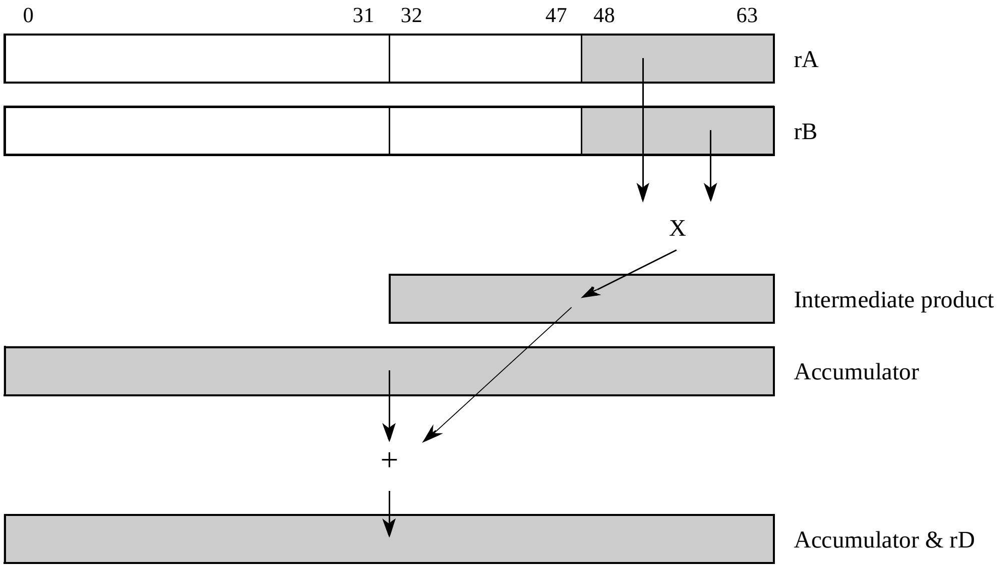


Figure 7-35. evmhogsmfaa

evmhogsmfan evmhogsmfan

Multiply Half Words, Odd, Guarded, Signed, Modulo, Fractional and Accumulate Negative

evmhogsmfan **r**D,**r**A,**r**B (O=1, F=1, S=1)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	101 1010 1111

$prod_{0:31} = rA_{48:63} * rB_{48:63}$

$temp1_{0:63} = EXTS(prod_{0:31} \| 0)$

$temp2_{0:64} = ACC_{0:63} - temp1_{0:63}$

$rD_{0:63} = ACC_{0:63} = temp2_{1:64}$

The low odd-numbered signed fractional half-word element in **r**A is multiplied by the corresponding signed fractional half-word element in **r**B. The 32-bit intermediate product is sign-extended to 64 bits. Then it is shifted left by one bit and subtracted from the contents of the 64-bit accumulator to form a 65-bit intermediate difference. The lower 64 bits of the intermediate difference is placed back into the accumulator and also written into **r**D.

NOTE

This is a modulo difference. There is no check for overflow and no saturation is performed. An overflow from the 64-bit difference, if one occurs, is not recorded into SPEFSCR.

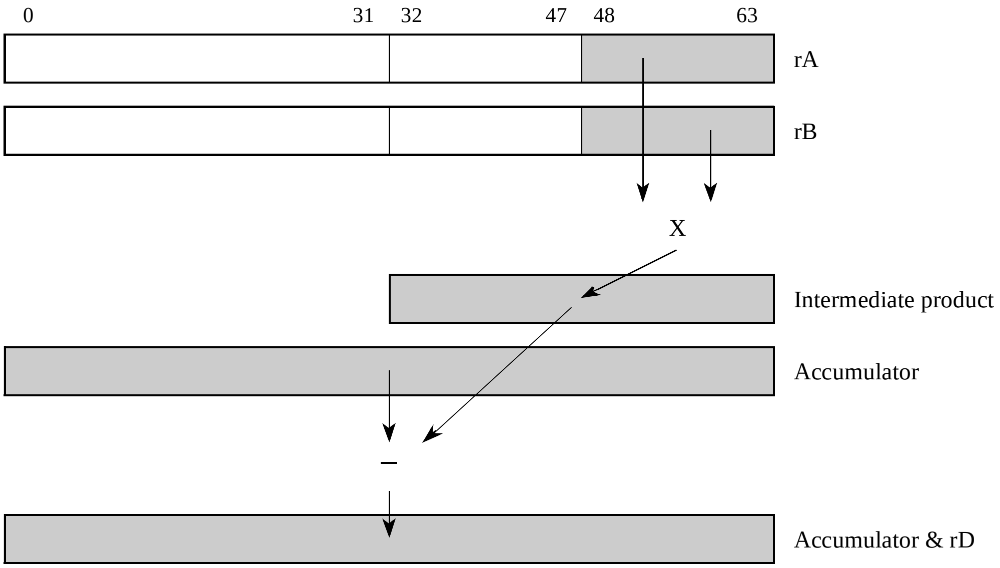


Figure 7-36. evmhogsmfan

evmhogsmiaa evmhogsmiaa

Multiply Half Words, Odd, Guarded, Signed, Modulo, Integer, and Accumulate

evmhogsmiaa **r**D,**r**A,**r**B (O=1, F=0, S=1)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	101 0010 1101

```
$prod_{0:31} = rA_{48:63} *si\ rB_{48:63}$
$temp1_{0:63} = EXTS(prod_{0:31})$
$temp2_{0:64} = ACC_{0:63} + temp1_{0:63}$
$rD_{0:63} = ACC_{0:63} = temp2_{1:64}$
```

The low odd-numbered signed integer half-word element in **r**A is multiplied by the corresponding signed integer half-word element in **r**B. The intermediate product is sign-extended to 64 bits and added to the contents of the 64-bit accumulator to form a 65-bit intermediate sum. The lower 64 bits of the intermediate sum is placed back into the accumulator and also written into **r**D.

NOTE

This is a modulo sum. There is no check for overflow and no saturation is performed. An overflow from the 64-bit sum, if one occurs, is not recorded into SPEFSCR.

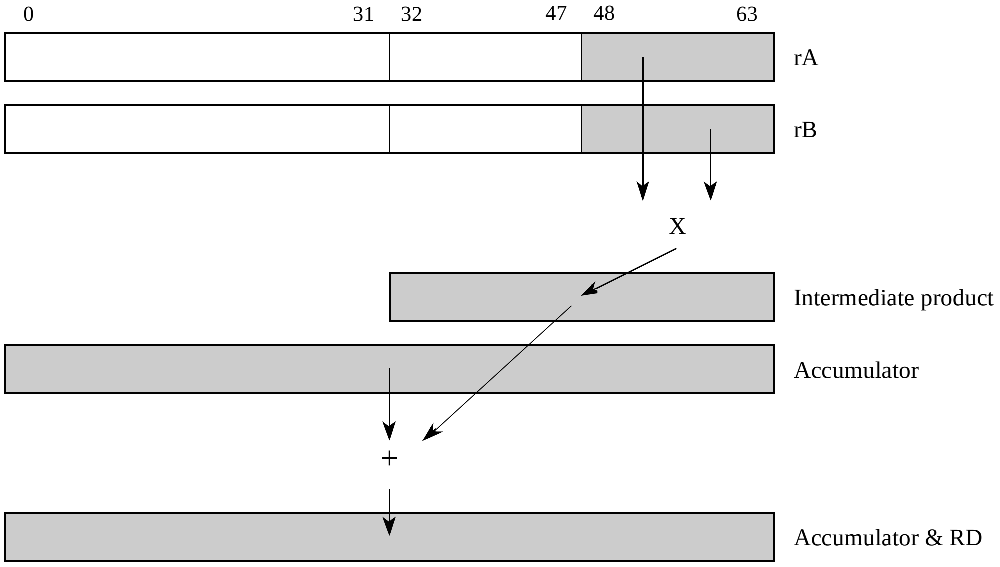


Figure 7-37. evmhogsmiaa

evmhogsmian evmhogsmian

Multiply Half Words, Odd, Guarded, Signed, Modulo, Integer, and Accumulate Negative

evmhogsmian **r**D,**r**A,**r**B (O=1, F=0, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 1010 1101

$prod_{0:31} = rA_{48:63} *si\ rB_{48:63}$

$temp1_{0:63} = EXTS(prod_{0:31})$

$temp2_{0:64} = ACC_{0:63} - temp1_{0:63}$

$rD_{0:63} = ACC_{0:63} = temp2_{1:64}$

The low odd-numbered signed integer half-word element in **r**A is multiplied by the corresponding signed integer half-word element in **r**B. The intermediate product is sign-extended to 64 bits and subtracted from the contents of the 64-bit accumulator to form a 65-bit intermediate difference. The lower 64 bits of the intermediate difference is placed back into the accumulator and also written into **r**D.

NOTE

This is a modulo difference. There is no check for overflow and no saturation is performed. An overflow from the 64-bit difference, if one occurs, is not recorded into SPEFSCR.

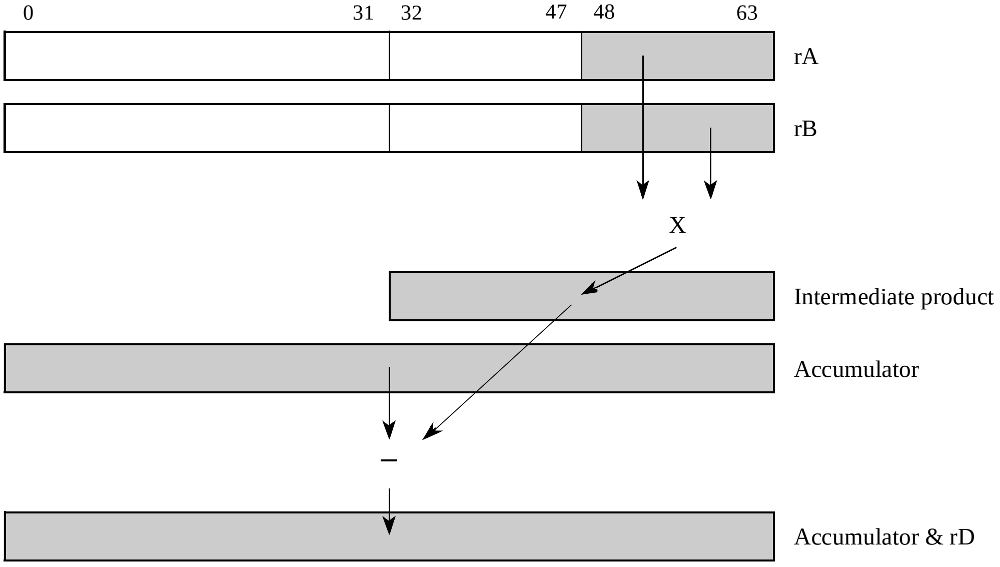


Figure 7-38. evmhogsmian

evmhogumiaa evmhogumiaa

Multiply Half Words, Odd, Guarded, Unsigned, Modulo, Integer, and Accumulate

evmhogumiaa **r**D,**r**A,**r**B (O=1, F=0, S=0)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	101 0010 1100

```
$prod_{0:31} = rA_{48:63}$ *ui $rB_{48:63}$
$temp1_{0:63} = EXTZ(prod_{0:31})$
$temp2_{0:64} = ACC_{0:63} + temp1_{0:63}$
$rD_{0:63} = ACC_{0:63} = temp2_{1:64}$
```

The low odd-numbered unsigned integer half-word element in **r**A is multiplied by the corresponding unsigned integer half-word element in **r**B. The intermediate product is sign-extended to 64 bits and added to the contents of the 64-bit accumulator to form a 65-bit intermediate sum. The lower 64 bits of the intermediate sum is placed back into the accumulator and also written into **r**D.

NOTE

This is a modulo sum. There is no check for overflow and no saturation is performed. An overflow from the 64-bit sum, if one occurs, is not recorded into SPEFSCR.

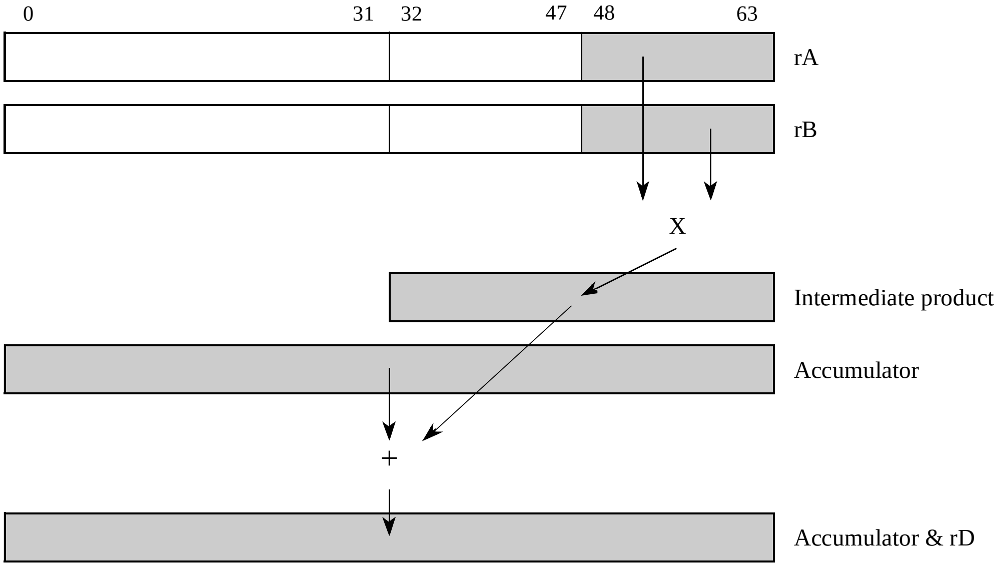


Figure 7-39. evmhogumiaa

evmhogumian evmhogumian

Multiply Half Words, Odd, Guarded, Unsigned, Modulo, Integer, and Accumulate Negative

evmhogumian **r**D,**r**A,**r**B (O=1, F=0, S=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 1010 1100

```
$prod_{0:31} = rA_{48:63} *ui\ rB_{48:63}$
$temp1_{0:63} = EXTZ(prod_{0:31})$
$temp2_{0:64} = ACC_{0:63} - temp1_{0:63}$
$rD_{0:63} = ACC_{0:63} = temp2_{1:64}$
```

The low odd-numbered unsigned integer half-word element in **r**A is multiplied by the corresponding unsigned integer half-word element in **r**B. The intermediate product is sign-extended to 64 bits and subtracted from the contents of the 64-bit accumulator to form a 65-bit intermediate difference. The lower 64 bits of the intermediate difference is placed back into the accumulator and also written into **r**D.

NOTE

This is a modulo difference. There is no check for overflow and no saturation is performed. An overflow from the 64-bit difference, if one occurs, is not recorded into SPEFSCR.

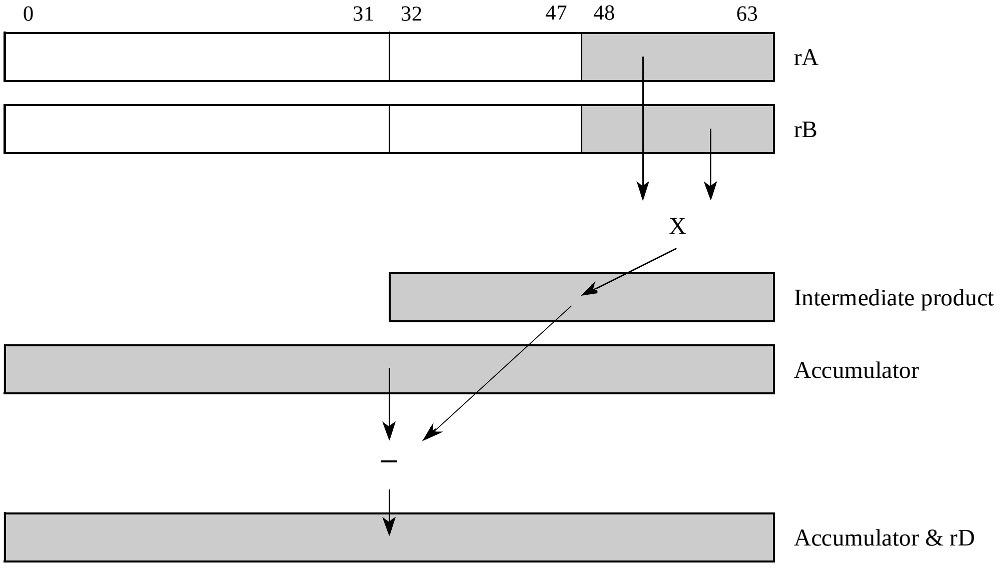


Figure 7-40. evmhogumian

evmhosmf evmhosmf

Vector Multiply Half Words, Odd, Signed, Modulo, Fractional

evmhosmf **r**D,**r**A,**r**B (M=1, O=1, F=1, S=1, A=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 0000 1111

```
$temp1_{0:32} = rA_{16:31} * rB_{16:31}$
$temp2_{0:32} = rA_{48:63} * rB_{48:63}$

$rD_{0:31} = temp1_{1:32}$
$rD_{32:63} = temp2_{1:32}$
```

Each odd-numbered signed fractional half-word element in **r**A is multiplied by the corresponding signed fractional half-word element in **r**B. The two 32-bit signed fractional products are placed into the two word elements of **r**D.

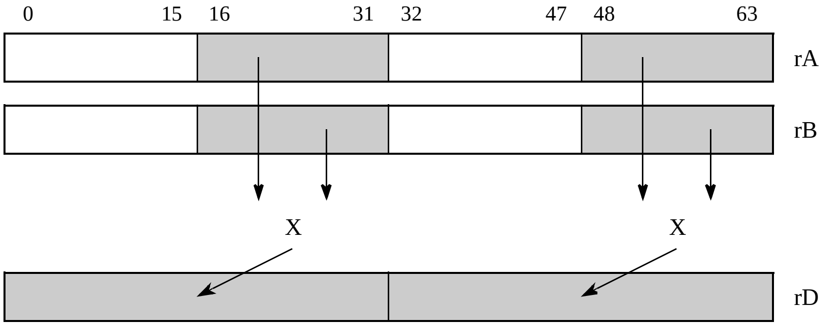


Figure 7-41. evmhosmf

evmhosmfa evmhosmfa

Vector Multiply Half Words, Odd, Signed, Modulo, Fractional, to Accumulator

evmhosmfa **r**D,**r**A,**r**B (M=1, O=1, F=1, S=1, A=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 0010 1111

```
$prod_{0:31} = rA_{16:31} * rB_{16:31}$
$prod_{32:63} = rA_{48:63} * rB_{48:63}$

$temp1_{0:32} = prod_{0:31} \| 0$
$temp2_{0:32} = prod_{32:63} \| 0$

$rD_{0:31} = temp1_{1:32}$
$rD_{32:63} = temp2_{1:32}$

$ACC_{0:63} = rD_{0:63}$
```

Each odd-numbered signed fractional half-word element in **r**A is multiplied by the corresponding signed fractional half-word element in **r**B. The two 32-bit signed fractional products are shifted left by one bit to remove the redundant sign bit and then placed into the two word elements of **r**D. The result in **r**D is also placed in the accumulator.

Other registers altered: ACC

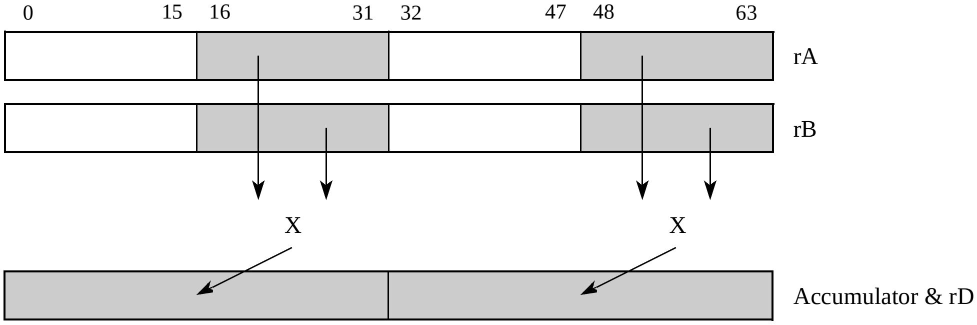


Figure 7-42. evmhosmfa

evmhosmfaaw

Vector Multiply Half Words, Odd, Signed, Modulo, Fractional, and Accumulate into Words

evmhosmfaaw **r**D,**r**A,**r**B (M=1, O=1, F=1, S=1)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	101 0000 1111

$temp1_{0:32} = (rA_{16:31} * rB_{16:31}) \,||\, 0$
$temp2_{0:32} = (rA_{48:63} * rB_{48:63}) \,||\, 0$

$temp3_{0:32} = ACC_{0:31} + temp1_{1:32}$
$temp4_{0:32} = ACC_{32:63} + temp2_{1:32}$

$ACC_{0:31} = rD_{0:31} = temp3_{1:32}$
$ACC_{32:63} = rD_{32:63} = temp4_{1:32}$

For each word element in the accumulator the following operations are performed in the order shown:

1. Each odd-numbered signed fractional half word element in **r**A is multiplied by the corresponding signed fractional half word element in **r**B.
2. The intermediate 32-bit product is shifted left by one bit to remove the redundant sign bit and then added to the contents of the accumulator word to form a 33-bit intermediate sum.
3. The low 32 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word.

Other registers altered: ACC

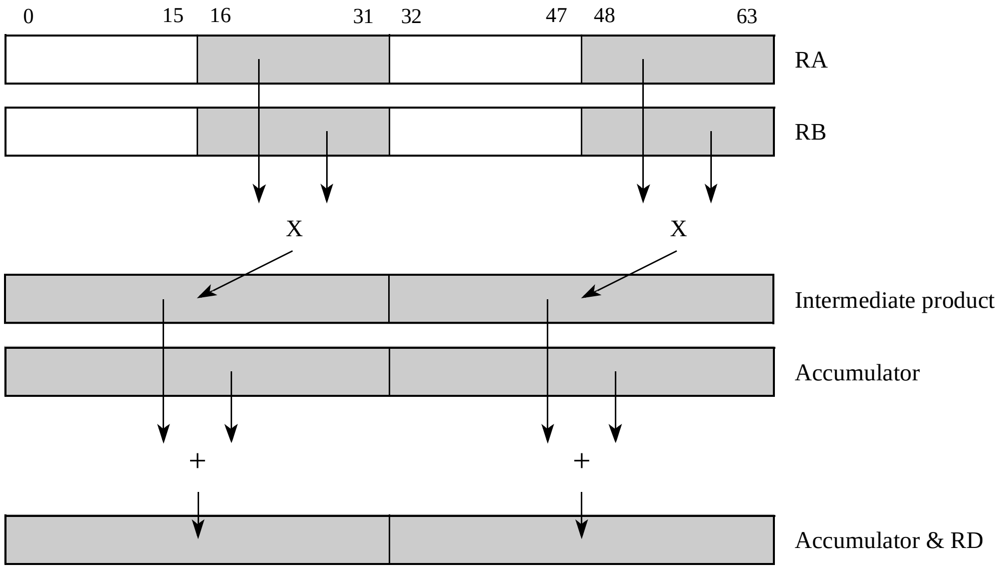


Figure 7-43. evmhosmfaaw

evmhosmfanw evmhosmfanw

Vector Multiply Half Words, Odd, Signed, Modulo, Fractional, and Accumulate Negative into Words

evmhosmfanw **r**D,**r**A,**r**B (M=1, O=1, F=1, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 1000 1111

```
$temp1_{0:32} = (rA_{16:31} * rB_{16:31}) || 0$
$temp2_{0:32} = (rA_{48:63} * rB_{48:63}) || 0$

$temp3_{0:32} = ACC_{0:31} - temp1_{1:32}$
$temp4_{0:32} = ACC_{32:63} - temp2_{1:32}$

$ACC_{0:31} = rD_{0:31} = temp3_{1:32}$
$ACC_{32:63} = rD_{32:63} = temp4_{1:32}$
```

For each word element in the accumulator the following operations are performed in the order shown:

1. Each odd-numbered signed fractional half word element in **r**A is multiplied by the corresponding signed fractional half word element in **r**B.
2. The intermediate 32-bit product is shifted left by one bit to remove the redundant sign bit and then subtracted from the contents of the accumulator word to form a 33-bit intermediate difference.
3. The low 32 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word.

Other registers altered: ACC

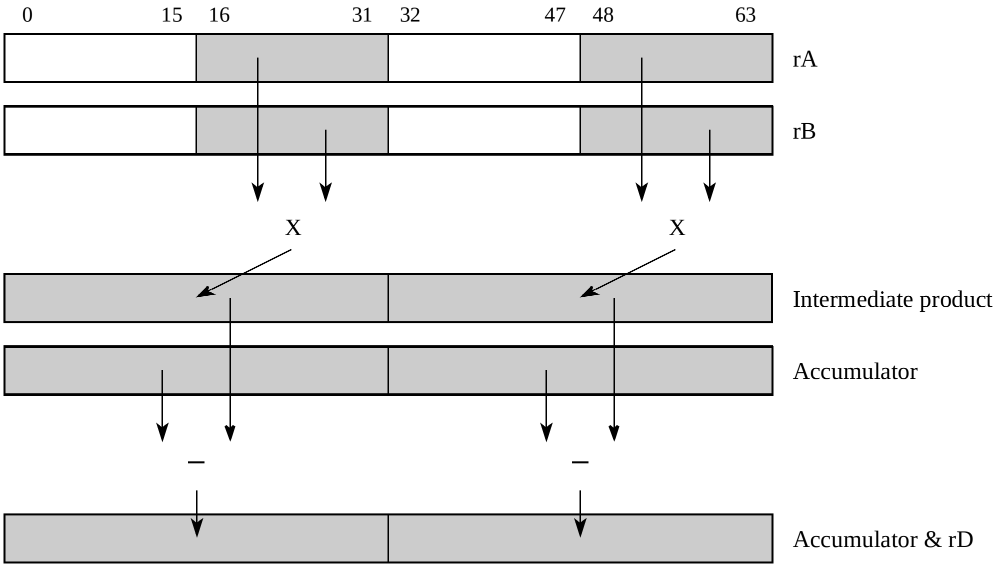


Figure 7-44. evmhosmfanw

evmhosmi evmhosmi

Vector Multiply Half Words, Odd, Signed, Modulo, Integer

evmhosmi **r**D,**r**A,**r**B (M=1, O=1, F=0, S=1, A=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 0000 1101

$rD_{0:31} = rA_{16:31} \text{ *si } rB_{16:31}$
$rD_{32:63} = rA_{48:63} \text{ *si } rB_{48:63}$

Each odd-numbered signed integer half word element in **r**A is multiplied by the corresponding signed integer half word element in **r**B. The two 32-bit signed integer products are placed into the two word elements of **r**D.

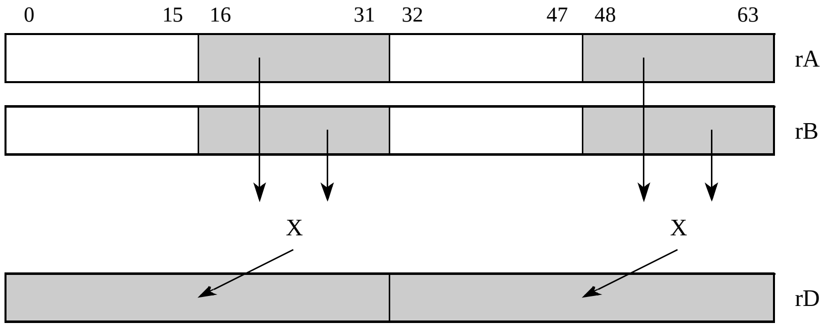


Figure 7-45. evmhosmi

evmhosmia

Vector Multiply Half Words, Odd, Signed, Modulo, Integer to Accumulator

evmhosmia **r**D,**r**A,**r**B (M=1, O=1, F=0, S=1, A=1)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	100 0010 1101

$rD_{0:31} = rA_{16:31} *si\ rB_{16:31}$
$rD_{32:63} = rA_{48:63} *si\ rB_{48:63}$

$ACC_{0:63} = rD_{0:63}$

Each odd-numbered signed integer half word element in **r**A is multiplied by the corresponding signed integer half word element in **r**B. The two 32-bit signed integer products are placed into the two word elements of **r**D. The result in **r**D is also placed in the accumulator.

Other registers altered: ACC

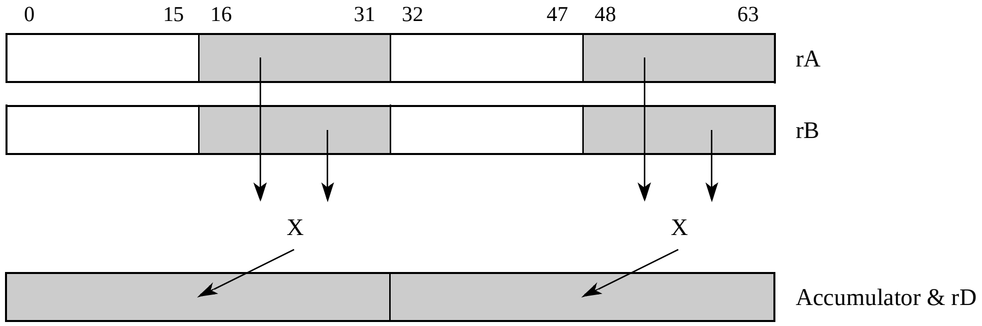


Figure 7-46. evmhosmia

evmhosmiaaw

evmhosmiaaw

Vector Multiply Half Words, Odd, Signed, Modulo, Integer, and Accumulate into Words

evmhosmiaaw **r**D,**r**A,**r**B (M=1, O=1, F=0, S=1)

0 … 5	6 … 10	11 … 15	16 … 20	21 … 31
4	RD	RA	RB	101 0000 1101

$temp1_{0:31} = rA_{16:31} *si\ rB_{16:31}$
$temp2_{0:31} = rA_{48:63} *si\ rB_{48:63}$

$temp3_{0:32} = ACC_{0:31} + temp1_{0:31}$
$temp4_{0:32} = ACC_{32:63} + temp2_{0:31}$

$ACC_{0:31} = rD_{0:31} = temp3_{1:32}$
$ACC_{32:63} = rD_{32:63} = temp4_{1:32}$

For each word element in the accumulator the following operations are performed in the order shown:

1. Each odd-numbered signed integer half word element in **r**A is multiplied by the corresponding signed integer half word element in **r**B.
2. The intermediate 32-bit product is added to the contents of the accumulator word to form a 33-bit intermediate sum.
3. The low 32 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word.

Other registers altered: ACC

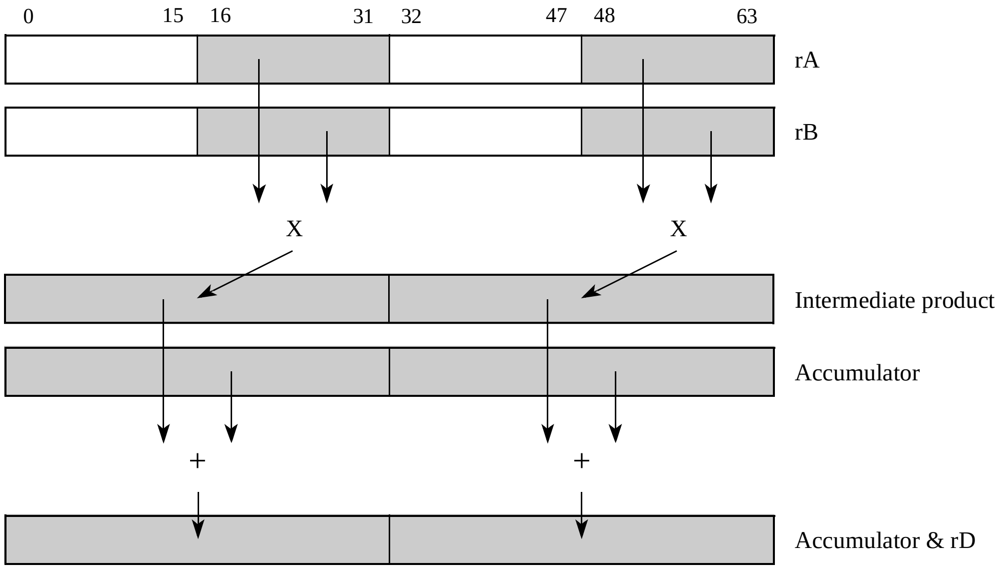


Figure 7-47. evmhosmiaaw

evmhosmianw evmhosmianw

Vector Multiply Half Words, Odd, Signed, Modulo, Integer, and Accumulate Negative into Words

evmhosmianw **r**D,**r**A,**r**B (M=1, O=1, F=0, S=1)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	101 1000 1101

$temp1_{0:31} = rA_{16:31} \ast si\ rB_{16:31}$
$temp2_{0:31} = rA_{48:63} \ast si\ rB_{48:63}$

$temp3_{0:32} = ACC_{0:31} - temp1_{0:31}$
$temp4_{0:32} = ACC_{32:63} - temp2_{0:31}$

$ACC_{0:31} = rD_{0:31} = temp3_{1:32}$
$ACC_{32:63} = rD_{32:63} = temp4_{1:32}$

For each word element in the accumulator the following operations are performed in the order shown:

1. Each odd-numbered signed integer half word element in **r**A is multiplied by the corresponding signed integer half word element in **r**B.
2. The intermediate 32-bit product is subtracted from the contents of the accumulator word to form a 33-bit intermediate difference.
3. The low 32 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word.

Other registers altered: ACC

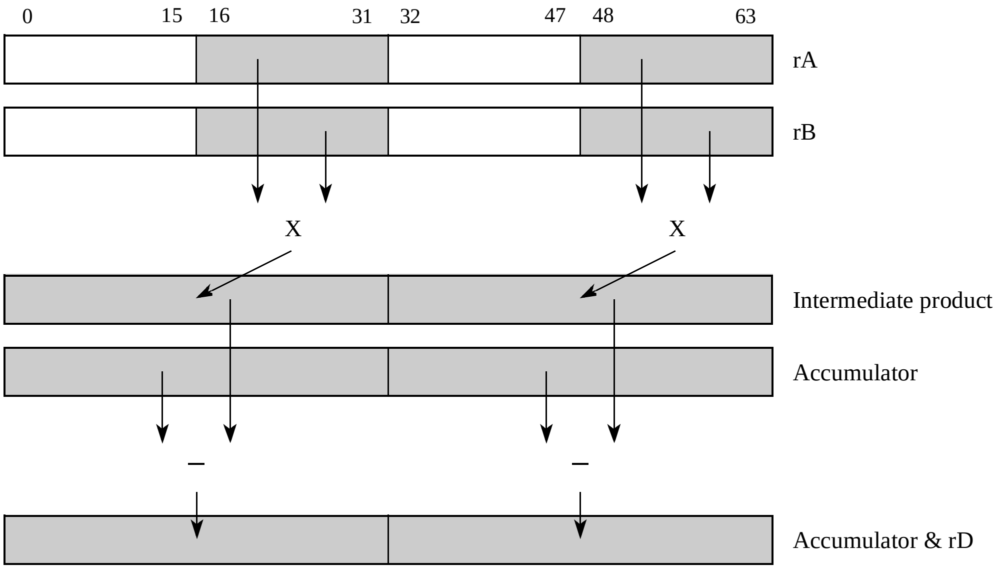


Figure 7-48. evmhosmianw

evmhossf evmhossf

Vector Multiply Half Words, Odd, Signed, Saturate, Fractional

evmhossf **r**D,**r**A,**r**B (M=0, O=1, F=1, S=1, A=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 0000 0111

```
$temp1_{0:32} = (rA_{16:31} * rB_{16:31}) || 0$
$temp2_{0:32} = (rA_{48:63} * rB_{48:63}) || 0$

$movh = temp1_0 \oplus temp1_1$
$movl = temp2_0 \oplus temp2_1$

$rD_{0:31} = SATURATE(movh, 0x7FFFFFFF, temp1_{1:32})$
$rD_{32:63} = SATURATE(movl, 0x7FFFFFFF, temp2_{1:32})$

$SPEFSCR_{OVH} = movh$
$SPEFSCR_{OV} = movl$
$SPEFSCR_{SOVH} = SPEFSCR_{SOVH} | movh$
$SPEFSCR_{SOV} = SPEFSCR_{SOV} | movl$
```

Each odd-numbered signed fractional half word element in **r**A is multiplied by the corresponding signed fractional half word element in **r**B. The two 32-bit signed fractional products are shifted left one bit to eliminate the redundant sign bit and then placed into the two word elements of **r**D. If the inputs are –1.0 and –1.0 the result is saturated to the most positive signed fraction (0x7FFF_FFFF). If saturation occurs, the overflow and summary overflow bits are recorded.

Other registers altered: SPEFSCR

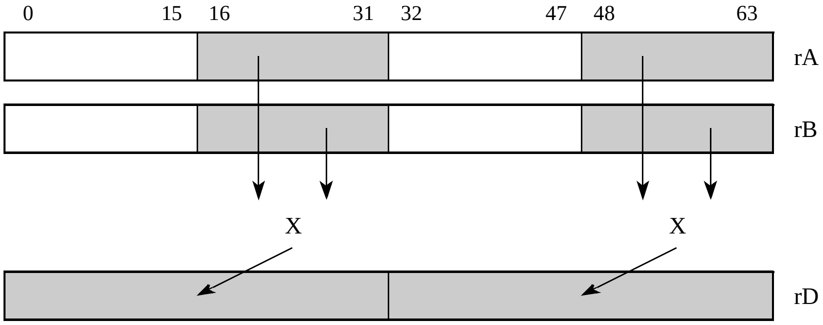


Figure 7-49. evmhossf

evmhossfa evmhossfa

Vector Multiply Half Words, Odd, Signed, Saturate, Fractional to Accumulator

evmhossfa **r**D,**r**A,**r**B (M=0, O=1, F=1, S=1, A=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 0010 0111

```
$temp1_{0:32} = (rA_{16:31} * rB_{16:31}) || 0$
$temp2_{0:32} = (rA_{48:63} * rB_{48:63}) || 0$

$movh = temp1_0 \oplus temp1_1$
$movl = temp2_0 \oplus temp2_1$

$rD_{0:31} = SATURATE(movh, 0x7FFFFFFF, temp1_{1:32})$
$rD_{32:63} = SATURATE(movl, 0x7FFFFFFF, temp2_{1:32})$

$ACC_{0:63} = rD_{0:63}$

$SPEFSCR_{OVH} = movh$
$SPEFSCR_{OV} = movl$
$SPEFSCR_{SOVH} = SPEFSCR_{SOVH} | movh$
$SPEFSCR_{SOV} = SPEFSCR_{SOV} | movl$
```

Each odd-numbered signed fractional half-word element in **r**A is multiplied by the corresponding signed fractional half-word element in **r**B. The two 32-bit signed fractional products are shifted left one bit to eliminate the redundant sign bit and then placed into the two word elements of **r**D. If the inputs are –1.0 and –1.0 the result is saturated to the most positive signed fraction (0x7FFF_FFFF). If saturation occurs, the overflow and summary overflow bits are recorded. The result in **r**D is also placed in the accumulator.

Other registers altered: SPEFSCR, ACC

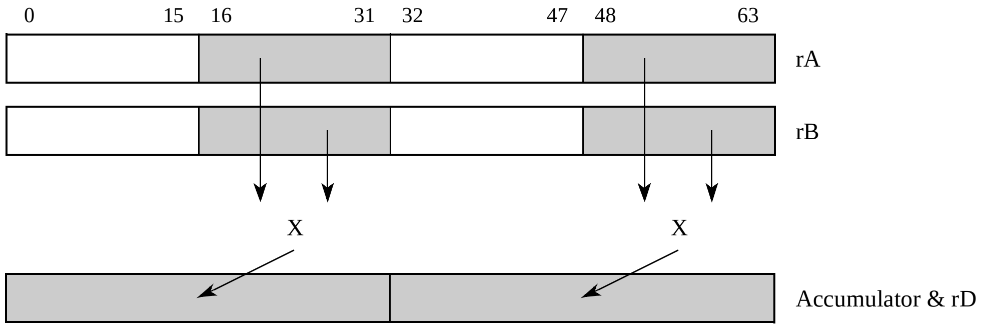


Figure 7-50. evmhossfa

evmhossfaaw evmhossfaaw

Vector Multiply Half Words, Odd, Signed, Saturate, Fractional, and Accumulate into Words

evmhossfaaw **r**D,**r**A,**r**B (M=0, O=1, F=1, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 0000 0111

```
$temp1_{0:32} = (rA_{16:31} * rB_{16:31}) || 0$
$temp2_{0:32} = (rA_{48:63} * rB_{48:63}) || 0$

$movh = temp1_0 \oplus temp1_1$
$movl = temp2_0 \oplus temp2_1$

$temp3_{0:31}$ = SATURATE(movh, 0x7FFFFFFF, $temp1_{1:32}$)
$temp4_{0:31}$ = SATURATE(movl, 0x7FFFFFFF, $temp2_{1:32}$)

$temp5_{0:32} = \{ACC_0,ACC_{0:31}\} + \{temp3_0,temp3_{0:31}\}$
$temp6_{0:32} = \{ACC_{32},ACC_{32:63}\} + \{temp4_0,temp4_{0:31}\}$

$ovh = temp5_0 \oplus temp5_1$
$ovl = temp6_0 \oplus temp6_1$

$rD_{0:31}$ = SATURATE_ACC(ovh, $temp5_0$, 0x80000000, 0x7FFFFFFF, $temp5_{1:32}$)
$rD_{32:63}$ = SATURATE_ACC(ovl, $temp6_0$, 0x80000000, 0x7FFFFFFF, $temp6_{1:32}$)

$ACC_{0:31} = rD_{0:31}$
$ACC_{32:63} = rD_{32:63}$

$SPEFSCR_{OVH}$ = movh | ovh
$SPEFSCR_{OV}$ = movl | ovl

$SPEFSCR_{SOVH} = SPEFSCR_{SOVH}$ | movh | ovh
$SPEFSCR_{SOV} = SPEFSCR_{SOV}$ | movl | ovl
```

Each odd-numbered signed fractional half-word element in **r**A is multiplied by the corresponding signed fractional half-word element in **r**B. The two 32-bit signed fractional products are shifted left one bit to eliminate the redundant sign bit. If the inputs are –1.0 and –1.0 the intermediate result is saturated to the most positive signed fraction (0x7FFF_FFFF). The intermediate 32-bit products are added to the respective accumulator word to form an intermediate sum. If the intermediate sum has overflowed, the appropriate saturation value (0x7FFF_FFFF if positive overflow or 0x8000_0000 if negative overflow) is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word.

If there is an overflow from either the multiply or the addition, the overflow information is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR, ACC

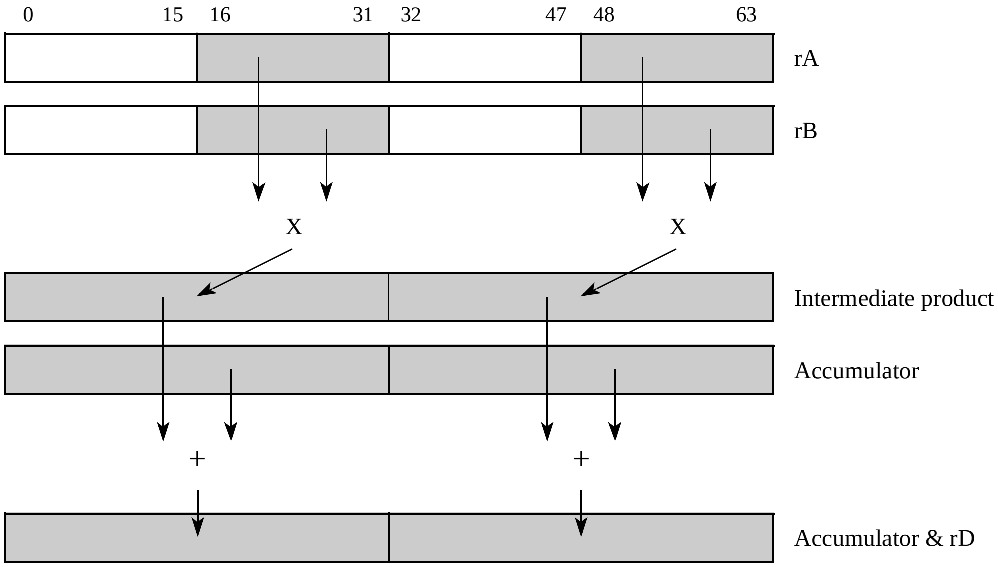


Figure 7-51. evmhossfaaw

evmhossfanw evmhossfanw

Vector Multiply Half Words, Odd, Signed, Saturate, Fractional, and Accumulate Negative into Words

evmhossfanw **r**D,**r**A,**r**B (M=0, O=1, F=1, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 1000 0111

```
$temp1_{0:32} = (rA_{16:31} * rB_{16:31}) || 0$
$temp2_{0:32} = (rA_{48:63} * rB_{48:63}) || 0$

$movh = temp1_0 \oplus temp1_1$
$movl = temp2_0 \oplus temp2_1$

$temp3_{0:31} = SATURATE(movh, 0x7FFFFFFF, temp1_{1:32})$
$temp4_{0:31} = SATURATE(movl, 0x7FFFFFFF, temp2_{1:32})$

$temp5_{0:32} = \{ACC_0,ACC_{0:31}\} - \{temp3_0,temp3_{0:31}\}$
$temp6_{0:32} = \{ACC_{32},ACC_{32:63}\} - \{temp4_0,temp4_{0:31}\}$

$ovh = temp5_0 \oplus temp5_1$
$ovl = temp6_0 \oplus temp6_1$

$rD_{0:31} = SATURATE\_ACC(ovh, temp5_0, 0x80000000, 0x7FFFFFFF, temp5_{1:32})$
$rD_{32:63} = SATURATE\_ACC(ovl, temp6_0, 0x80000000, 0x7FFFFFFF, temp6_{1:32})$

$ACC_{0:31} = rD_{0:31}$
$ACC_{32:63} = rD_{32:63}$

$SPEFSCR_{OVH} = movh | ovh$
$SPEFSCR_{OV} = movl | ovl$

$SPEFSCR_{SOVH} = SPEFSCR_{SOVH} | movh | ovh$
$SPEFSCR_{SOV} = SPEFSCR_{SOV} | movl | ovl$
```

For each word element in the accumulator the following operations are performed in the order shown:

1. Each odd-numbered signed fractional half-word element in **r**A is multiplied by the corresponding signed fractional half-word element in **r**B. The two 32-bit signed fractional products are shifted left one bit to eliminate the redundant sign bit. If the inputs are –1.0 and –1.0 the intermediate result is saturated to the most positive signed fraction (0x7FFF_FFFF).
2. The intermediate 32-bit product is subtracted from the contents of the accumulator word to form an intermediate difference.
3. If the intermediate difference has overflowed, the appropriate saturation value (0x7FFF_FFFF if positive overflow or 0x8000_0000 if negative overflow) is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word.
4. If there is an overflow from either the multiply or the subtraction, the overflow information is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR, ACC

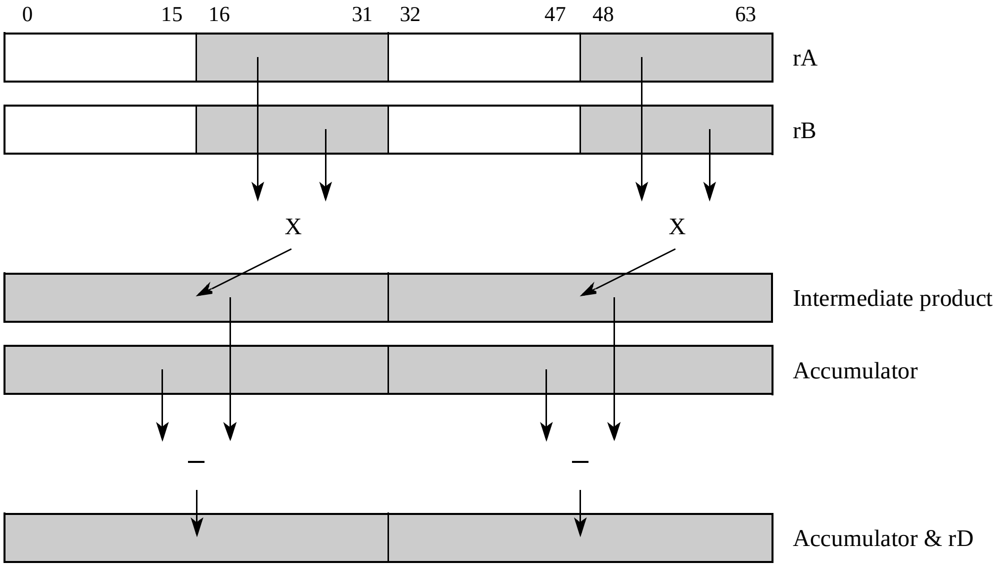


Figure 7-52. evmhossfanw

evmhossiaaw evmhossiaaw

Vector Multiply Half Words, Odd, Signed, Saturate, Integer, and Accumulate into Words

evmhossiaaw **r**D,**r**A,**r**B (M=0, O=1, F=0, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 0000 0101

```
$temp1_{0:31} = rA_{16:31}$ *si $rB_{16:31}$
$temp2_{0:31} = rA_{48:63}$ *si $rB_{48:63}$

$temp3_{0:32} = \{ACC_0, ACC_{0:31}\} + \{temp1_0, temp1_{0:31}\}$
$temp4_{0:32} = \{ACC_{32}, ACC_{32:63}\} + \{temp2_0, temp2_{0:31}\}$

$ovh = temp3_0 \oplus temp3_1$
$ovl = temp4_0 \oplus temp4_1$

$rD_{0:31}$ = SATURATE_ACC(ovh, $temp3_0$, 0x80000000, 0x7FFFFFFF, $temp3_{1:32}$)
$rD_{32:63}$ = SATURATE_ACC(ovl, $temp4_0$, 0x80000000, 0x7FFFFFFF, $temp4_{1:32}$)

$ACC_{0:31} = rD_{0:31}$
$ACC_{32:63} = rD_{32:63}$

$SPEFSCR_{OVH} = ovh$
$SPEFSCR_{OV} = ovl$
$SPEFSCR_{SOVH} = SPEFSCR_{SOVH} \mid ovh$
$SPEFSCR_{SOV} = SPEFSCR_{SOV} \mid ovl$
```

For each word element in the accumulator the following operations are performed in the order shown:

1. Each odd-numbered signed integer half word element in **r**A is multiplied by the corresponding signed integer half word element in **r**B.
2. The intermediate 32-bit product is added to the contents of the accumulator word to form an intermediate sum.
3. If the intermediate sum has overflowed, the appropriate saturation value (0x7FFF_FFFF if positive overflow or 0x8000_0000 if negative overflow) is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word.
4. If there is an overflow from the addition, the overflow information is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR, ACC

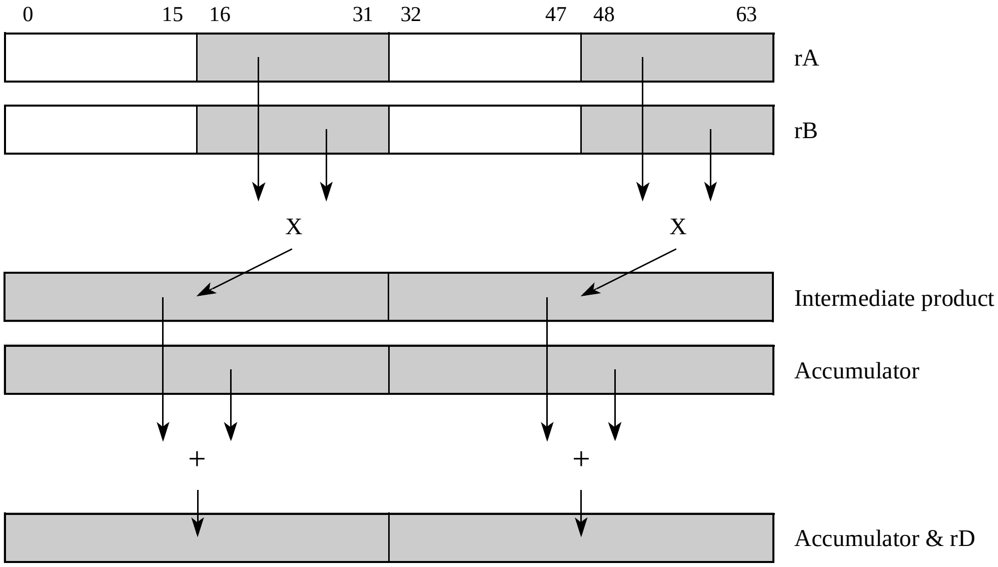


Figure 7-53. evmhossiaaw

evmhossianw evmhossianw

Vector Multiply Half Words, Odd, Signed, Saturate, Integer and Accumulate Negative into Words

evmhossianw **r**D,**r**A,**r**B (M=0, O=1, F=0, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 1000 0101

```
$temp1_{0:31} = rA_{16:31}$ *si $rB_{16:31}$
$temp2_{0:31} = rA_{48:63}$ *si $rB_{48:63}$

$temp3_{0:32} = \{ACC_0,ACC_{0:31}\} - \{temp1_0,temp1_{0:31}\}$
$temp4_{0:32} = \{ACC_{32},ACC_{32:63}\} - \{temp2_0,temp2_{0:31}\}$

$ovh = temp3_0 \oplus temp3_1$
$ovl = temp4_0 \oplus temp4_1$

$rD_{0:31}$ = SATURATE_ACC(ovh, $temp3_0$, 0x80000000, 0x7FFFFFFF, $temp3_{1:32}$)
$rD_{32:63}$ = SATURATE_ACC(ovl, $temp4_0$, 0x80000000, 0x7FFFFFFF, $temp4_{1:32}$)

$ACC_{0:31} = rD_{0:31}$
$ACC_{32:63} = rD_{32:63}$

$SPEFSCR_{OVH}$ = ovh
$SPEFSCR_{OV}$ = ovl
$SPEFSCR_{SOVH} = SPEFSCR_{SOVH}$ | ovh
$SPEFSCR_{SOV} = SPEFSCR_{SOV}$ | ovl
```

For each word element in the accumulator the following operations are performed in the order shown:

1. Each odd-numbered signed integer half-word element in **r**A is multiplied by the corresponding signed integer half-word element in **r**B.
2. The intermediate 32-bit product is subtracted from the contents of the accumulator word to form an intermediate difference.
3. If the intermediate difference has overflowed, the appropriate saturation value (0x7FFF_FFFF if positive overflow or 0x8000_0000 if negative overflow) is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word.
4. If there is an overflow from the subtraction, the overflow information is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR, ACC

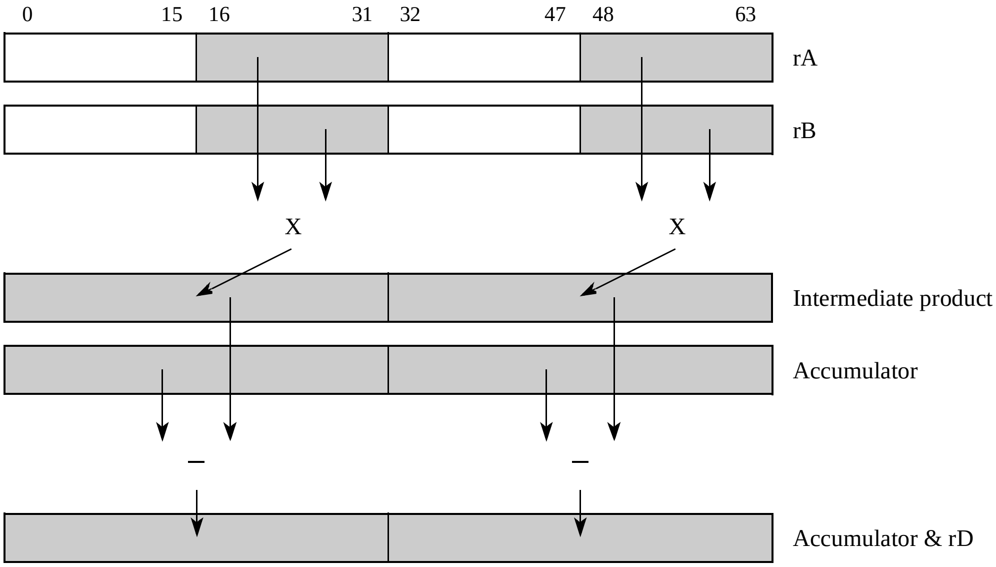


Figure 7-54. evmhossianw

evmhoumi evmhoumi

Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer

evmhoumi **r**D,**r**A,**r**B (M=1, O=1, F=0, S=0, A=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 0000 1100

$rD_{0:31} = rA_{16:31}$ *ui $rB_{16:31}$
$rD_{32:63} = rA_{48:63}$ *ui $rB_{48:63}$

Each odd-numbered unsigned integer half-word element in **r**A is multiplied by the corresponding unsigned integer half word element in **r**B. The two 32-bit unsigned integer products are placed into the two word elements of **r**D.

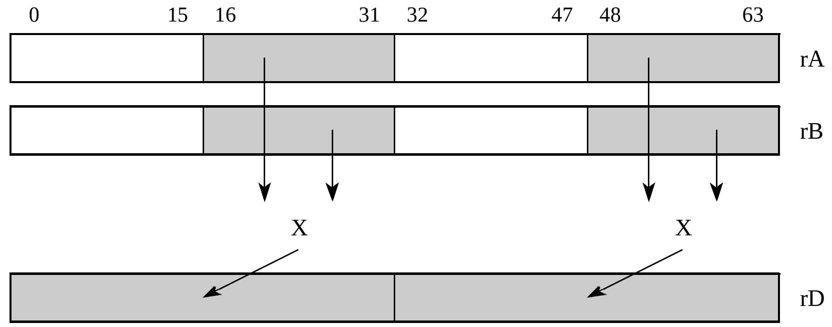


Figure 7-55. evmhoumi

evmhoumia evmhoumia

Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer, to Accumulator

evmhoumia **r**D,**r**A,**r**B (M=1, O=1, F=0, S=0, A=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 0010 1100

$rD_{0:31} = rA_{16:31}$ *ui $rB_{16:31}$
$rD_{32:63} = rA_{48:63}$ *ui $rB_{48:63}$

$ACC_{0:63} = rD_{0:63}$

Each odd-numbered unsigned integer half-word element in **r**A is multiplied by the corresponding unsigned integer half-word element in **r**B. The two 32-bit unsigned integer products are placed into the two word elements of **r**D. The result in **r**D is also placed in the accumulator.

Other registers altered: ACC

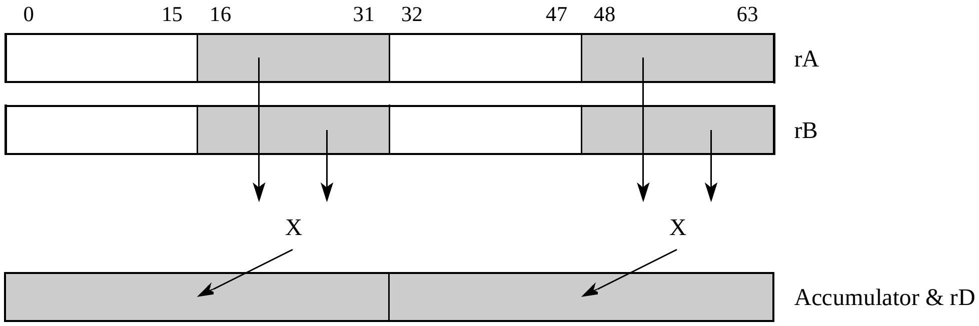


Figure 7-56. evmhoumia

evmhoumiaaw

evmhoumiaaw

Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer and Accumulate into Words

evmhoumiaaw **r**D,**r**A,**r**B (M=1, O=1, F=0, S=0)

0 … 5	6 … 10	11 … 15	16 … 20	21 … 31
4	RD	RA	RB	101 0000 1100

$temp1_{0:31} = rA_{16:31} \text{ *ui } rB_{16:31}$
$temp2_{0:31} = rA_{48:63} \text{ *ui } rB_{48:63}$

$temp3_{0:32} = ACC_{0:31} + temp1_{0:31}$
$temp4_{0:32} = ACC_{32:63} + temp2_{0:31}$

$ACC_{0:31} = rD_{0:31} = temp3_{1:32}$
$ACC_{32:63} = rD_{32:63} = temp4_{1:32}$

For each word element in the accumulator the following operations are performed in the order shown:

1. Each odd-numbered unsigned integer half-word element in **r**A is multiplied by the corresponding unsigned integer half-word element in **r**B.
2. The intermediate 32-bit product is added to the contents of the accumulator word to form a 33-bit intermediate sum. The low 32 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word.

Other registers altered: ACC

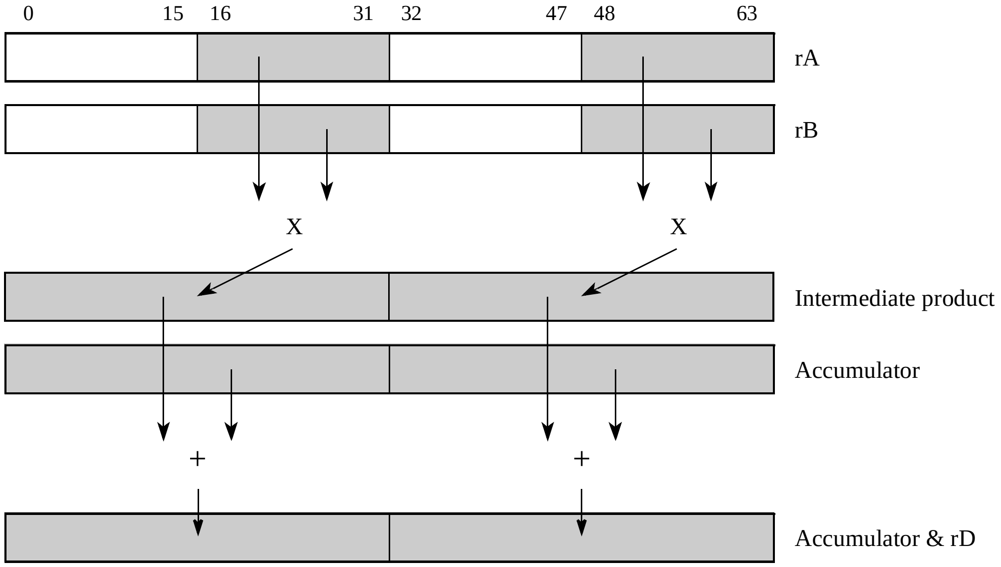


Figure 7-57. evmhoumiaaw

evmhoumianw evmhoumianw

Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer, and Accumulate Negative into Words

evmhoumianw **r**D,**r**A,**r**B (M=1, O=1, F=0, S=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 1000 1100

$temp1_{0:31} = rA_{16:31} *ui\ rB_{16:31}$
$temp2_{0:31} = rA_{48:63} *ui\ rB_{48:63}$

$temp3_{0:32} = ACC_{0:31} - temp1_{0:31}$
$temp4_{0:32} = ACC_{32:63} - temp2_{0:31}$

$ACC_{0:31} = rD_{0:31} = temp3_{1:32}$
$ACC_{32:63} = rD_{32:63} = temp4_{1:32}$

For each word element in the accumulator the following operations are performed in the order shown:

1. Each odd-numbered unsigned integer half word element in **r**A is multiplied by the corresponding unsigned integer half word element in **r**B.
2. The intermediate 32-bit product is subtracted from the contents of the accumulator word to form a 33-bit intermediate difference. The low 32 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word.

Other registers altered: ACC

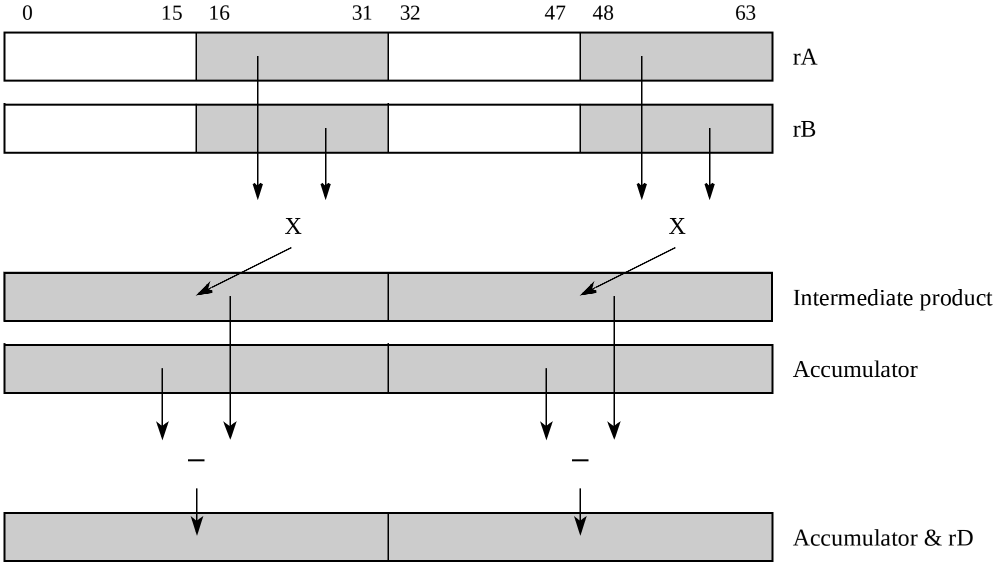


Figure 7-58. evmhoumianw

evmhousiaaw evmhousiaaw

Vector Multiply Half Words, Odd, Unsigned, Saturate, Integer and Accumulate into Words

evmhousiaaw **r**D,**r**A,**r**B (M=0, O=1, F=0, S=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 0000 0100

```
$temp1_{0:31}$ = $rA_{16:31}$ *ui $rB_{16:31}$
$temp2_{0:31}$ = $rA_{48:63}$ *ui $rB_{48:63}$

$temp3_{0:32}$ = $ACC_{0:31}$ + $temp1_{0:31}$
$temp4_{0:32}$ = $ACC_{32:63}$ + $temp2_{0:31}$

ovh = $temp3_{0}$
ovl = $temp4_{0}$

$rD_{0:31}$ = SATURATE_ACC(ovh, 0xFFFFFFFF, $temp3_{1:32}$)
$rD_{32:63}$ = SATURATE_ACC(ovl, 0xFFFFFFFF, $temp4_{1:32}$)

$ACC_{0:31}$ = $rD_{0:31}$
$ACC_{32:63}$ = $rD_{32:63}$

$SPEFSCR_{OVH}$ = ovh
$SPEFSCR_{OV}$ = ovl
$SPEFSCR_{SOVH}$ = $SPEFSCR_{SOVH}$ | ovh
$SPEFSCR_{SOV}$ = $SPEFSCR_{SOV}$ | ovl
```

For each word element in the accumulator the following operations are performed in the order shown:

1. Each odd-numbered unsigned integer half word element in **r**A is multiplied by the corresponding unsigned integer half word element in **r**B.
2. The intermediate 32-bit product is added to the contents of the accumulator word to form a 33-bit intermediate sum.
3. If the intermediate sum has overflowed, 0xFFFF_FFFF is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word.
4. If there is an overflow from the addition, the overflow information is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR, ACC

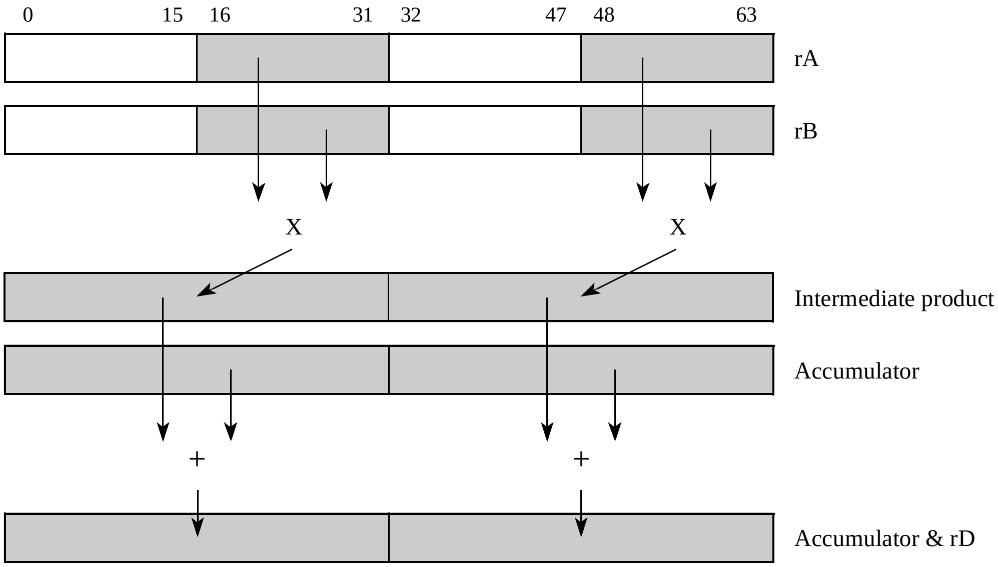


Figure 7-59. evmhousiaaw

evmhousianw evmhousianw

Vector Multiply Half Words, Odd, Unsigned, Saturate, Integer and Accumulate Negative into Words

evmhousianw **r**D,**r**A,**r**B (M=0, O=1, F=0, S=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 1000 0100

```
$temp1_{0:31} = rA_{16:31}$ *ui $rB_{16:31}$
$temp2_{0:31} = rA_{48:63}$ *ui $rB_{48:63}$

$temp3_{0:32} = ACC_{0:31} - temp1_{0:31}$
$temp4_{0:32} = ACC_{32:63} - temp2_{0:31}$

$ovh = temp3_0$
$ovl = temp4_0$

$rD_{0:31}$ = SATURATE_ACC(ovh, 0x00000000, $temp3_{1:32}$)
$rD_{32:63}$ = SATURATE_ACC(ovl, 0x00000000, $temp4_{1:32}$)

$ACC_{0:31} = rD_{0:31}$
$ACC_{32:63} = rD_{32:63}$

$SPEFSCR_{OVH}$ = ovh
$SPEFSCR_{OV}$ = ovl
$SPEFSCR_{SOVH} = SPEFSCR_{SOVH}$ | ovh
$SPEFSCR_{SOV} = SPEFSCR_{SOV}$ | ovl
```

For each word element in the accumulator the following operations are performed in the order shown:

1. Each odd-numbered unsigned integer half word element in **r**A is multiplied by the corresponding unsigned integer half word element in **r**B.
2. The intermediate 32-bit product is subtracted from the contents of the accumulator word to form a 33-bit intermediate difference.
3. If the intermediate difference has underflowed (is negative), 0x0000_0000 is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word.
4. If there is an underflow from either subtraction, the underflow information is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR, ACC

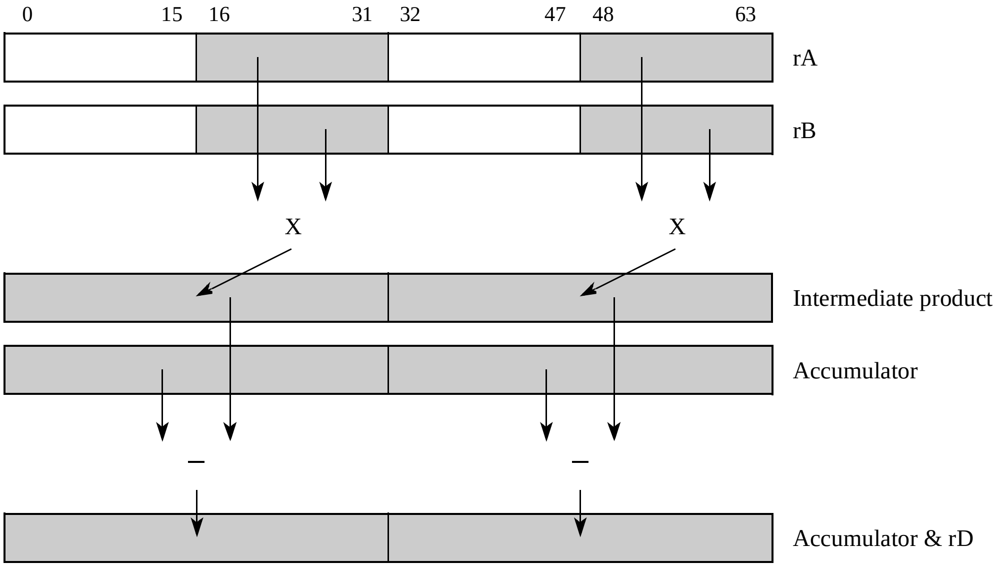


Figure 7-60. evmhousianw

7.4.2 Multiply Words Instructions

The following instructions perform 32 × 32 multiplies with optional saturation, returning either the higher or lower portion of the product, with and without accumulates, using signed or unsigned integer or fractional operands.

Table 7-5. Multiply Words Instructions

Vector Multiply Word High Signed, Modulo, Fractional (**evmwhsmf**)	Vector Multiply Word Low Signed, Saturate, Integer and Accumulate in Words **(evmwlssiaaw)**
Vector Multiply Word High Signed, Modulo, Fractional, to Accumulator (**evmwhsmfa)**	Vector Multiply Word Low Signed, Saturate, Integer and Accumulate Negative in Words **(evmwlssianw)**
Vector Multiply Word High Signed, Modulo, Integer **(evmwhsmi)**	Vector Multiply Word Low Unsigned, Modulo, Integer **(evmwlumi)**
Vector Multiply Word High Signed, Modulo, Integer to Accumulator **(evmwhsmia)**	Vector Multiply Word Low Unsigned, Modulo, Integer, to Accumulator **(evmwlumia)**
Vector Multiply Word High Signed, Saturate, Fractional, to Accumulator **(evmwhssfa)**	Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate in Words **(evmwlumiaaw)**
Vector Multiply Word High Unsigned, Modulo, Integer **(evmwhumi)**	Vector Multiply Word Low Unsigned, Modulo, Integer, and Accumulate Negative in Words **(evmwlumianw)**
Vector Multiply Word High Unsigned, Modulo, Integer, to Accumulator **(evmwhumia)**	Vector Multiply Word Low Unsigned, Saturate, Integer, and Accumulate in Words **(evmwlusiaaw)**
Vector Multiply Word Low Signed, Modulo, Integer and Accumulate in Words **(evmwlsmiaaw)**	Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate Negative in Words **(evmwlusianw)**
Vector Multiply Word Low Signed, Modulo, Integer, and Accumulate Negative in Words **(evmwlsmianw)**	Vector Multiply Word Signed, Modulo, Fractional **(evmwsmf)**

Table 7-5. Multiply Words Instructions (Continued)

Vector Multiply Word Signed, Modulo, Fractional, to Accumulator **(evmwsmfa)**	Vector Multiply Word Unsigned, Modulo, Integer, to Accumulator **(evmwumia)**
Vector Multiply Word Signed, Modulo, Fractional, and Accumulate **(evmwsmfaa)**	Vector Multiply Word Unsigned, Modulo, Integer and Accumulate **(evmwumiaa)**
Vector Multiply Word Signed, Modulo, Fractional and Accumulate Negative **(evmwsmfan)**	Vector Multiply Word Unsigned, Modulo, Integer and Accumulate Negative **(evmwumian)**
Vector Multiply Word Signed, Modulo, Integer **(evmwsmi)**	Vector Add Signed, Modulo, Integer to Accumulator Word (**evaddsmiaaw**)
Vector Multiply Word Signed, Modulo, Integer, to Accumulator **(evmwsmia)**	Vector Add Signed, Saturate, Integer to Accumulator Word **(evaddssiaaw)**
Vector Multiply Word Signed, Modulo, Integer, and Accumulate **(evmwsmiaa)**	Vector Add Unsigned, Modulo, Integer to Accumulator Word **(evaddumiaaw)**
Vector Multiply Word Signed, Modulo, Integer and Accumulate Negative **(evmwsmian**)	Vector Add Unsigned, Saturate, Integer to Accumulator Word (**evaddusiaaw**)
Vector Multiply Word Signed, Saturate, Fractional **(evmwssf)**	Vector Subtract Signed, Modulo, Integer to Accumulator Word **(evsubfsmiaaw)**
Vector Multiply Word Signed, Saturate, Fractional, to Accumulator **(evmwssfa)**	Vector Subtract Signed, Saturate, Integer to Accumulator Word **(evsubfssiaaw)**
Vector Multiply Word Signed, Saturate, Fractional, and Accumulate **(evmwssfaa)**	Vector Subtract Unsigned, Modulo, Integer to Accumulator Word **(evsubfumiaaw)**
Vector Multiply Word Signed, Saturate, Fractional and Accumulate Negative **(evmwssfan)**	Vector Subtract Unsigned, Saturate, Integer to Accumulator Word **(evsubfusiaaw)**
Vector Multiply Word Unsigned, Modulo, Integer **(evmwumi)**	—

evmwhsmf evmwhsmf

Vector Multiply Word High Signed, Modulo, Fractional

evmwhsmf **r**D,**r**A,**r**B (M=1, F=1, S=1,A=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 0100 1111

```
$temp1_{0:63} = rA_{0:31} * rB_{0:31}$
$temp2_{0:63} = rA_{32:63} * rB_{32:63}$
$rD_{0:31} = temp1_{1:32}$
$rD_{32:63} = temp2_{1:32}$
```

Each signed fractional word element in **r**A is multiplied by the corresponding signed fractional word element in **r**B. Bits 1–32 of the two 64-bit signed fractional products (eliminating the redundant sign bit) are placed into the two word elements of **r**D.

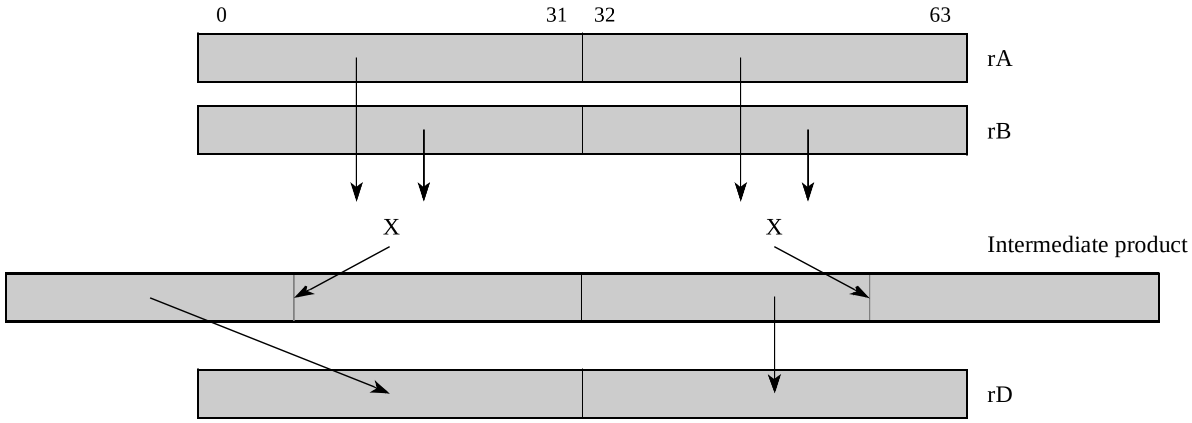


Figure 7-61. evmwhsmf

evmwhsmfa

Vector Multiply Word High Signed, Modulo, Fractional, to Accumulator

evmwhsmfa **r**D,**r**A,**r**B (M=1, F=1, S=1,A=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 0110 1111

```
$temp1_{0:64} = rA_{0:31} * rB_{0:31}$
$temp2_{0:64} = rA_{32:63} * rB_{32:63}$

$rD_{0:31} = temp1_{1:32}$
$rD_{32:63} = temp2_{1:32}$

$ACC_{0:63} = rD_{0:63}$
```

Each signed fractional word element in **r**A is multiplied by the corresponding signed fractional word element in **r**B. Bits 1–32 of the two 64-bit signed fractional products (eliminating the redundant sign bit) are placed into the two word elements of **r**D. The result in **r**D is also placed in the accumulator.

Other registers altered: ACC

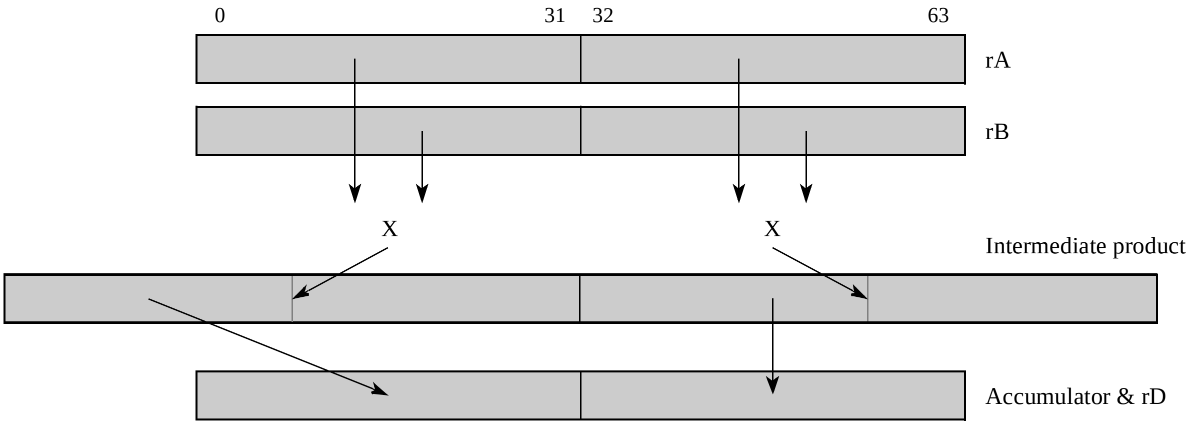


Figure 7-62. evmwhsmfa

evmwhsmi evmwhsmi

Vector Multiply Word High Signed, Modulo, Integer

evmwhsmi **r**D,**r**A,**r**B (M=1, F=0, S=1,A=0)

0 … 5	6 … 10	11 … 15	16 … 20	21 … 31
4	RD	RA	RB	100 0100 1101

```
$temp1_{0:63} = rA_{0:31}$ *si $rB_{0:31}$
$temp2_{0:63} = rA_{32:63}$ *si $rB_{32:63}$

$rD_{0:31} = temp1_{0:31}$
$rD_{32:63} = temp2_{0:31}$
```

Each signed integer word element in **r**A is multiplied by the corresponding signed integer word element in **r**B. The upper 32-bits of the two 64-bit signed integer products are placed into the two word elements of **r**D.

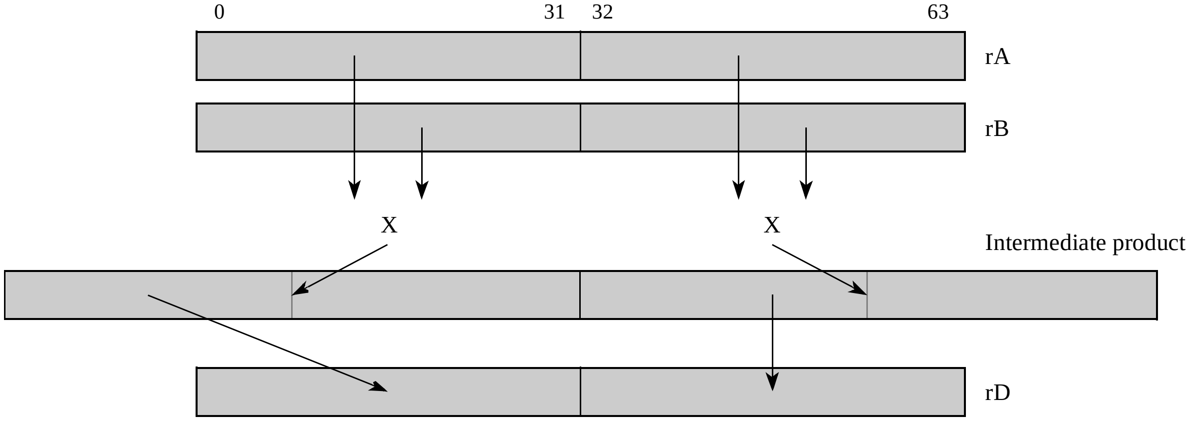


Figure 7-63. evmwhsmi

evmwhsmia evmwhsmia

Vector Multiply Word High Signed, Modulo, Integer to Accumulator

evmwhsmia **r**D,**r**A,**r**B (M=1, F=0, S=1,A=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 0110 1101

$temp1_{0:63} = rA_{0:31} \text{ *si } rB_{0:31}$
$temp2_{0:63} = rA_{32:63} \text{ *si } rB_{32:63}$

$rD_{0:31} = temp1_{0:31}$
$rD_{32:63} = temp2_{0:31}$

$ACC_{0:63} = rD_{0:63}$

Each signed integer word element in **r**A is multiplied by the corresponding signed integer word element in **r**B. The upper 32-bits of the two 64-bit signed integer products are placed into the two word elements of **r**D. The result in **r**D is also placed in the accumulator.

Other registers altered: ACC

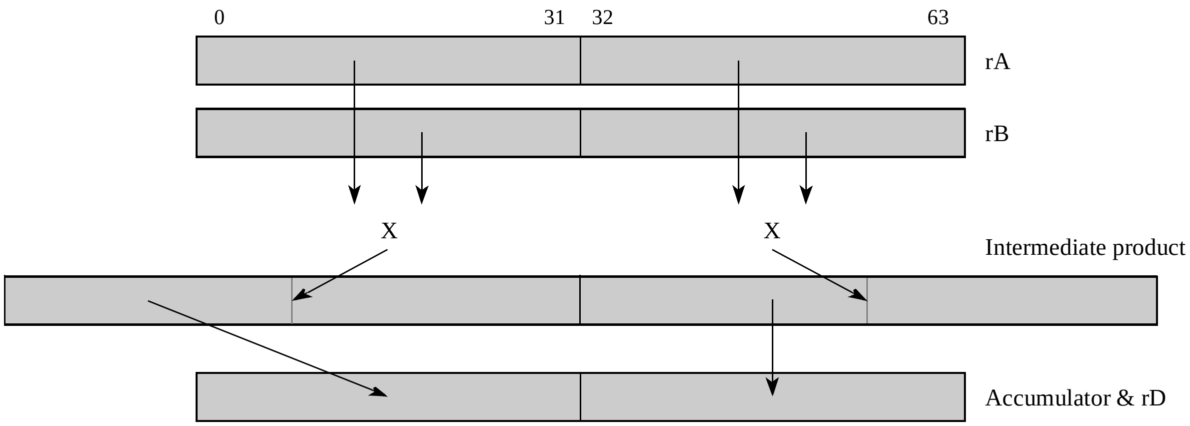


Figure 7-64. evmwhsmia

evmwhssf evmwhssf

Vector Multiply Word High Signed, Saturate, Fractional

evmwhssf **r**D,**r**A,**r**B (M=0, F=1, S=1,A=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 0100 0111

```
$temp1_{0:63} = rA_{0:31} * rB_{0:31}$
$temp2_{0:63} = rA_{32:63} * rB_{32:63}$

$movh = temp1_0 \oplus temp1_1$
$movl = temp2_0 \oplus temp2_1$

$rD_{0:31}$ = SATURATE(movh, 0x7FFFFFFF, $temp1_{1:32}$)
$rD_{32:63}$ = SATURATE(movl, 0x7FFFFFFF, $temp2_{1:32}$)

$SPEFSCR_{OVH} = movh$
$SPEFSCR_{OV} = movl$
$SPEFSCR_{SOVH} = SPEFSCR_{SOVH} | movh$
$SPEFSCR_{SOV} = SPEFSCR_{SOV} | movl$
```

Each signed fractional word element in **r**A is multiplied by the corresponding signed fractional word element in **r**B. Bits$_{1:32}$ of the two 64-bit signed fractional products (eliminating the redundant sign bit) are placed into the two word elements of **r**D. If the inputs are -1.0 and -1.0 the result is saturated to the most positive signed fraction (0x7FFFFFFF). If saturation occurs the overflow and summary overflow bits are recorded.

Other registers altered: SPEFSCR

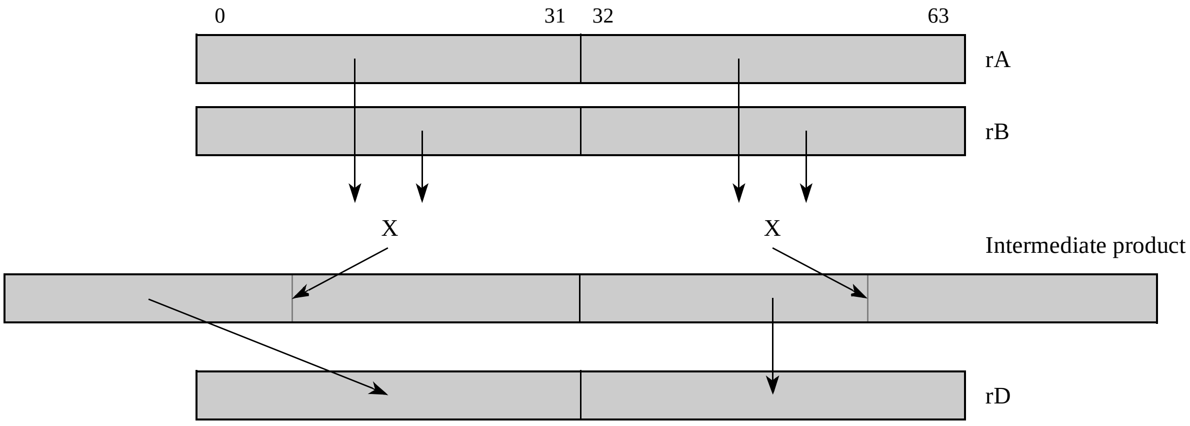


Figure 7-65. evmwhssf

evmwhssfa evmwhssfa

Vector Multiply Word High Signed, Saturate, Fractional, to Accumulator

evmwhssfa **r**D,**r**A,**r**B (M=0, F=1, S=1,A=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 0110 0111

```
$temp1_{0:63} = rA_{0:31} * rB_{0:31}$
$temp2_{0:63} = rA_{32:63} * rB_{32:63}$

$movh = temp1_0 \oplus temp1_1$
$movl = temp2_0 \oplus temp2_1$

$rD_{0:31} = SATURATE(movh, 0x7FFFFFFF, temp1_{1:32})$
$rD_{32:63} = SATURATE(movl, 0x7FFFFFFF, temp2_{1:32})$

$ACC_{0:63} = rD_{0:63}$

$SPEFSCR_{OVH} = movh$
$SPEFSCR_{OV} = movl$
$SPEFSCR_{SOVH} = SPEFSCR_{SOVH} | movh$
$SPEFSCR_{SOV} = SPEFSCR_{SOV} | movl$
```

Each signed fractional word element in **r**A is multiplied by the corresponding signed fractional word element in **r**B. Bits 1–32 of the two 64-bit signed fractional products (eliminating the redundant sign bit) are placed into the two word elements of **r**D. If the inputs are –1.0 and –1.0 the result is saturated to the most positive signed fraction (0x7FFF_FFFF). If saturation occurs, the overflow and summary overflow bits are recorded. The result in **r**D is also placed in the accumulator.

Other registers altered: SPEFSCR, ACC

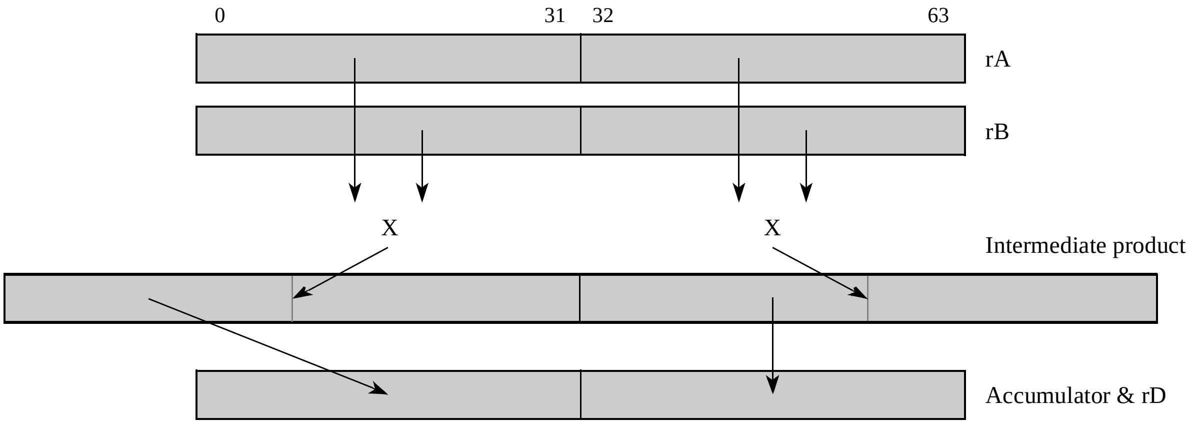


Figure 7-66. evmwhssfa

evmwhumi

Vector Multiply Word High Unsigned, Modulo, Integer

evmwhumi **r**D,**r**A,**r**B (M=1, F=0, S=0,A=0)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	100 0100 1100

$temp1_{0:63} = rA_{0:31} \text{ *ui } rB_{0:31}$
$temp2_{0:63} = rA_{32:63} \text{ *ui } rB_{32:63}$

$rD_{0:31} = temp1_{0:31}$
$rD_{32:63} = temp2_{0:31}$

Each unsigned integer word element in **r**A is multiplied by the corresponding unsigned integer word element in **r**B. The upper 32-bits of the two 64-bit unsigned integer products are placed into the two word elements of **r**D.

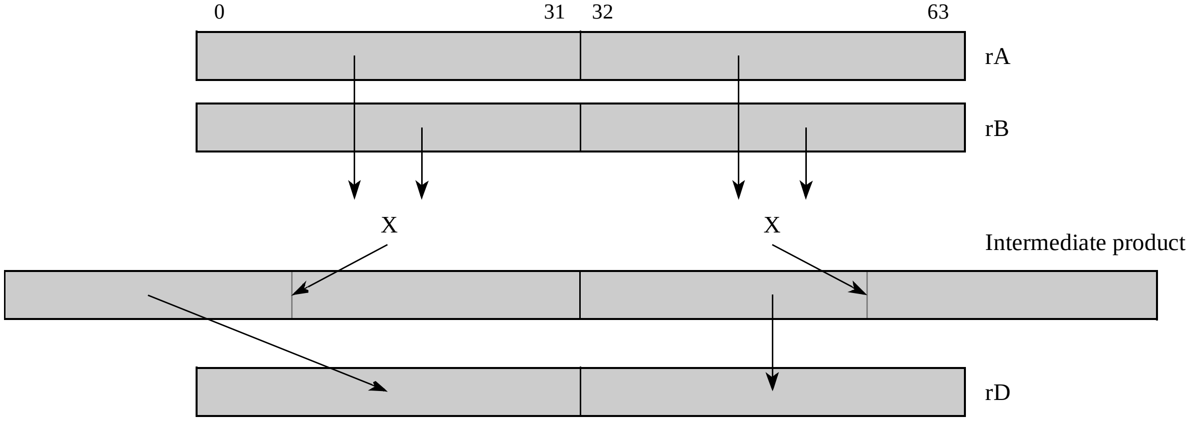


Figure 7-67. evmwhumi

evmwhumia evmwhumia

Vector Multiply Word High Unsigned, Modulo, Integer, to Accumulator

evmwhumia **r**D,**r**A,**r**B (M=1, F=0, S=0,A=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 0110 1100

```
$temp1_{0:63} = rA_{0:31}$ *ui $rB_{0:31}$
$temp2_{0:63} = rA_{32:63}$ *ui $rB_{32:63}$

$rD_{0:31} = temp1_{0:31}$
$rD_{32:63} = temp2_{0:31}$

$ACC_{0:63} = rD_{0:63}$
```

Each unsigned integer word element in **r**A is multiplied by the corresponding unsigned integer word element in **r**B. The upper 32-bits of the two 64-bit unsigned integer products are placed into the two word elements of **r**D. The result in **r**D is also placed in the accumulator.

Other registers altered: ACC

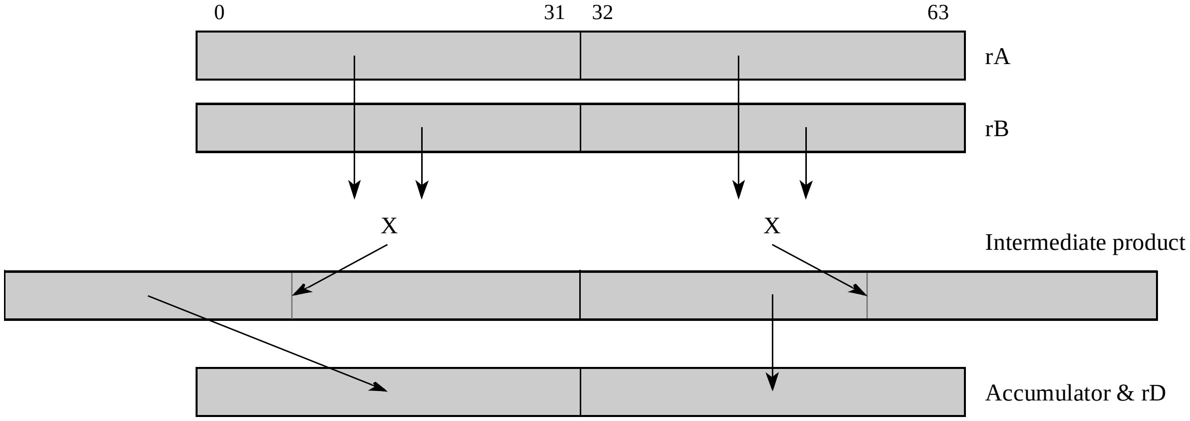


Figure 7-68. evmwhumia

evmwlsmiaaw evmwlsmiaaw

Vector Multiply Word Low Signed, Modulo, Integer and Accumulate in Words

evmwlsmiaaw **r**D,**r**A,**r**B (M=1, F=0, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 0100 1001

$temp1_{0:63} = rA_{0:31} *si\ rB_{0:31}$
$temp2_{0:63} = rA_{32:63} *si\ rB_{32:63}$

$rD_{0:31} = ACC_{0:31} + temp1_{32:63}$
$rD_{32:63} = ACC_{32:63} + temp2_{32:63}$

$ACC_{0:63} = rD_{0:63}$

For each word element in the accumulator the following operations are performed in the order shown:

1. Each signed integer word element in **r**A is multiplied by the corresponding signed integer word element in **r**B.
2. The low 32 bits of the 64-bit intermediate product are added to the contents of the accumulator word and placed into the corresponding **r**D word.
3. The result in **r**D is also placed in the accumulator.

NOTE

This instruction produces a valid result only if the intermediate product can be represented in the lower 32 bits.

Other registers altered: ACC

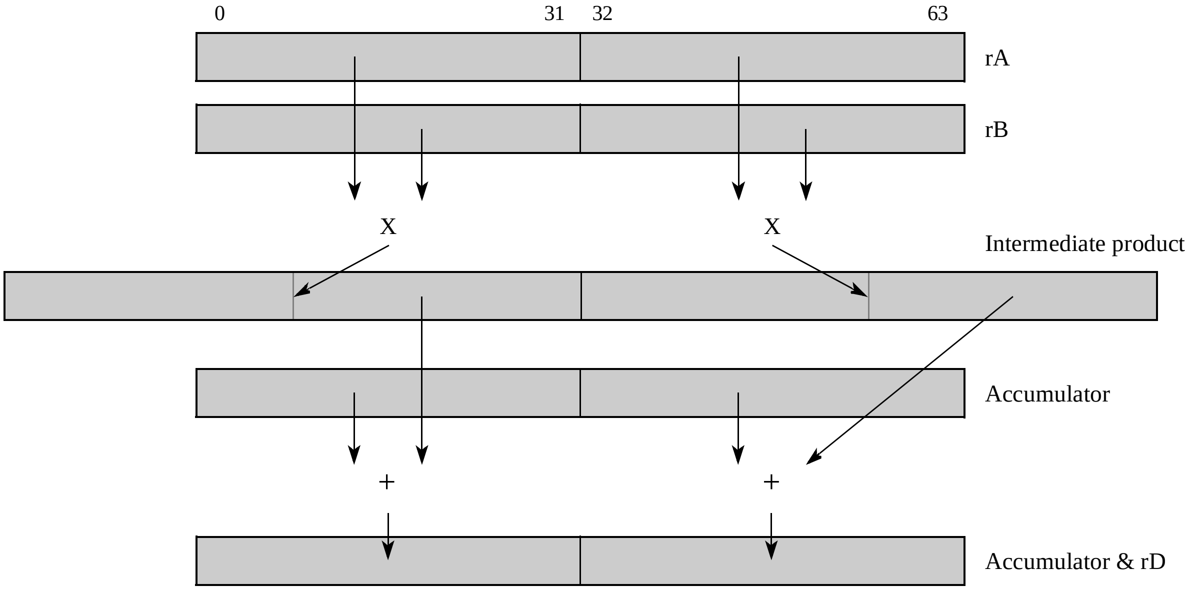


Figure 7-69. evmwlsmiaaw

evmwlsmianw evmwlsmianw

Vector Multiply Word Low Signed, Modulo, Integer, and Accumulate Negative in Words

evmwlsmianw **r**D,**r**A,**r**B (M=1, F=0, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 1100 1001

$temp1_{0:63} = rA_{0:31}$ *si $rB_{0:31}$
$temp2_{0:63} = rA_{32:63}$ *si $rB_{32:63}$

$rD_{0:31} = ACC_{0:31} - temp1_{32:63}$
$rD_{32:63} = ACC_{32:63} - temp2_{32:63}$

$ACC_{0:63} = rD_{0:63}$

For each word element in the accumulator the following operations are performed in the order shown:

1. Each signed integer word element in **r**A is multiplied by the corresponding signed integer word element in **r**B.
2. The low 32 bits of the 64-bit intermediate product are subtracted from the contents of the accumulator word and placed into the corresponding **r**D word.
3. The result in **r**D is also placed in the accumulator.

NOTE

This instruction produces a valid result only if the intermediate product can be represented in the lower 32 bits.

Other registers altered: ACC

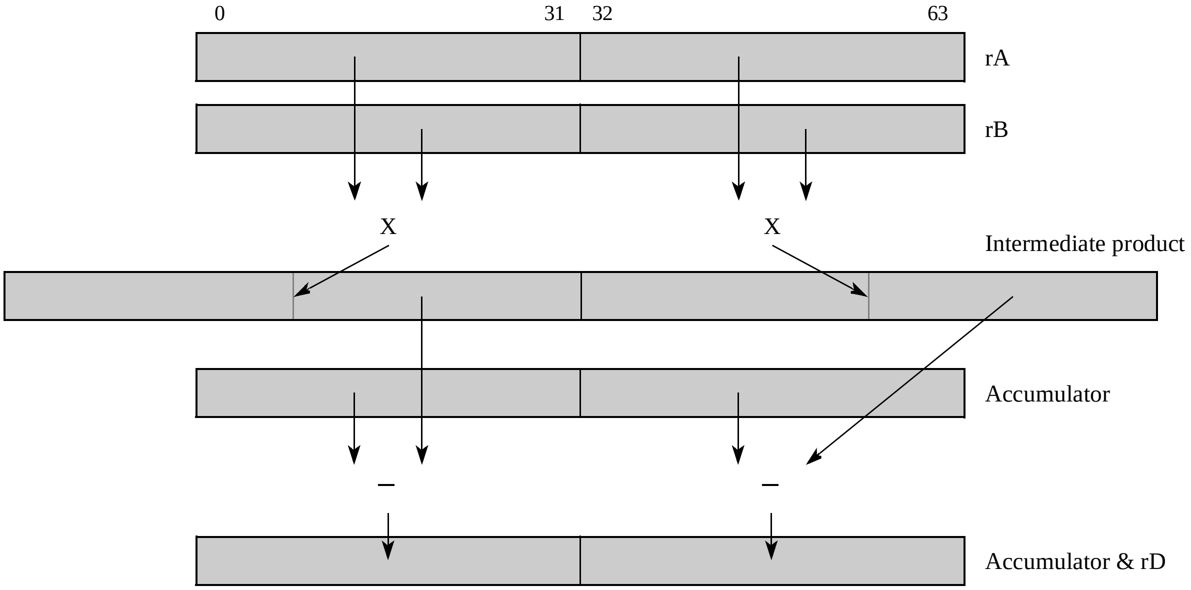


Figure 7-70. evmwlsmianw

evmwlssiaaw evmwlssiaaw

Vector Multiply Word Low Signed, Saturate, Integer and Accumulate in Words

evmwlssiaaw **r**D,**r**A,**r**B (M=0, F=0, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 0100 0001

```
$temp1_{0:63} = rA_{0:31}$ *si $rB_{0:31}$
$temp2_{0:63} = rA_{32:63}$ *si $rB_{32:63}$

$temp3_{0:32} = \{ACC_0,ACC_{0:31}\} + \{temp1_{32},temp1_{32:63}\}$
$temp4_{0:32} = \{ACC_{32},ACC_{32:63}\} + \{temp2_{32},temp2_{32:63}\}$

$ovh = temp3_0 \oplus temp3_1$
$ovl = temp4_0 \oplus temp4_1$

$rD_{0:31}$ = SATURATE_ACC(ovh, $temp3_0$, 0x80000000, 0x7FFFFFFF, $temp3_{1:32}$)
$rD_{32:63}$ = SATURATE_ACC(ovl, $temp4_0$, 0x80000000, 0x7FFFFFFF, $temp4_{1:32}$)

$ACC_{0:31} = rD_{0:31}$
$ACC_{32:63} = rD_{32:63}$

$SPEFSCR_{OVH} = ovh$
$SPEFSCR_{OV} = ovl$
$SPEFSCR_{SOVH} = SPEFSCR_{SOVH} | ovh$
$SPEFSCR_{SOV} = SPEFSCR_{SOV} | ovl$
```

For each word element in the accumulator the following operations are performed in the order shown:

1. Each signed integer word element in **r**A is multiplied by the corresponding signed integer word element in **r**B.
2. The low 32 bits of the 64-bit intermediate product are added to the contents of the accumulator word to form an intermediate sum.
3. If the intermediate sum has overflowed, the appropriate saturation value (0x7FFF_FFFF if positive overflow or 0x8000_0000 if negative overflow) is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word.
4. If there is an overflow from the addition, the overflow information is recorded in the SPEFSCR overflow and summary overflow bits.

NOTE

This instruction produces a valid result only if the intermediate product can be represented in the lower 32 bits.

Other registers altered: SPEFSCR, ACC

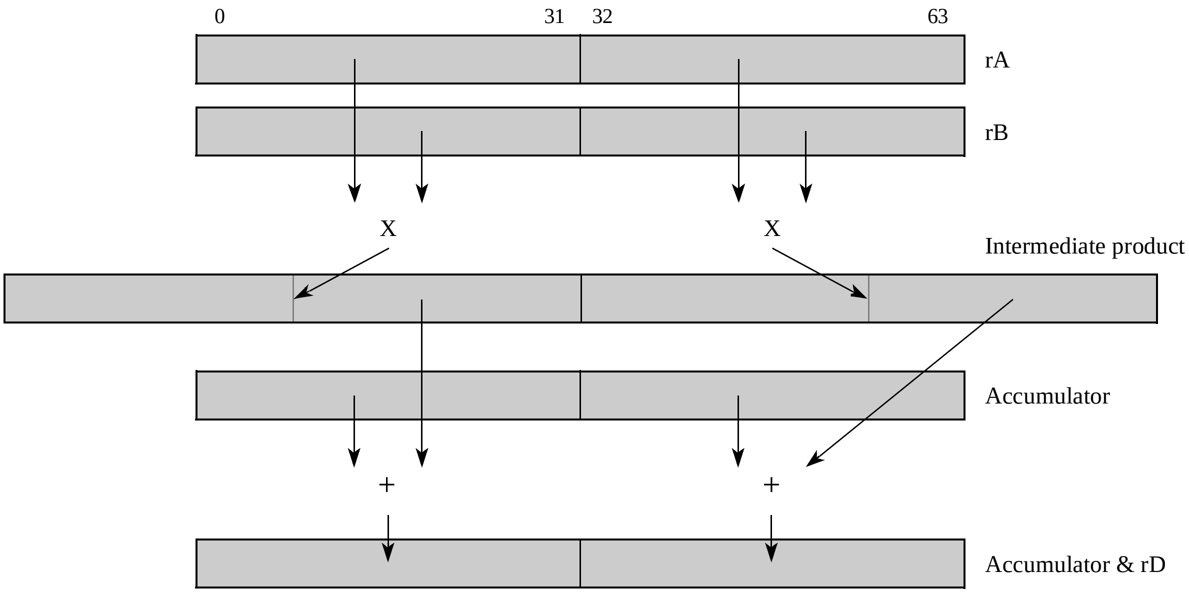


Figure 7-71. evmwlssiaaw

evmwlssianw evmwlssianw

Vector Multiply Word Low Signed, Saturate, Integer and Accumulate Negative in Words

evmwlssianw **r**D,**r**A,**r**B (M=0, F=0, S=1)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	101 1100 0001

```
$temp1_{0:63} = rA_{0:31}$ *si $rB_{0:31}$
$temp2_{0:63} = rA_{32:63}$ *si $rB_{32:63}$

$temp3_{0:32} = \{ACC_0,ACC_{0:31}\} - \{temp1_{32},temp1_{32:63}\}$
$temp4_{0:32} = \{ACC_{32},ACC_{32:63}\} - \{temp2_{32},temp2_{32:63}\}$

$ovh = temp3_0 \oplus temp3_1$
$ovl = temp4_0 \oplus temp4_1$

$rD_{0:31}$ = SATURATE_ACC(ovh, $temp3_0$, 0x80000000, 0x7FFFFFFF, $temp3_{1:32}$)
$rD_{32:63}$ = SATURATE_ACC(ovl, $temp4_0$, 0x80000000, 0x7FFFFFFF, $temp4_{1:32}$)

$ACC_{0:31} = rD_{0:31}$
$ACC_{32:63} = rD_{32:63}$

$SPEFSCR_{OVH} = ovh$
$SPEFSCR_{OV} = ovl$
$SPEFSCR_{SOVH} = SPEFSCR_{SOVH} | ovh$
$SPEFSCR_{SOV} = SPEFSCR_{SOV} | ovl$
```

For each word element in the accumulator the following operations are performed in the order shown:

1. Each signed integer word element in **r**A is multiplied by the corresponding signed integer word element in **r**B.
2. The low 32 bits of the 64-bit intermediate product are subtracted from the contents of the accumulator word to form an intermediate difference.
3. If the intermediate difference has overflowed, the appropriate saturation value (0x7FFFFFFF if positive overflow or 0x80000000 if negative overflow) is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word.
4. If there is an overflow from the difference, the overflow information is recorded in the SPEFSCR overflow and summary overflow bits.

NOTE

This instruction produces a valid result only if the intermediate product can be represented in the lower 32 bits.

Other registers altered: SPEFSCR, ACC

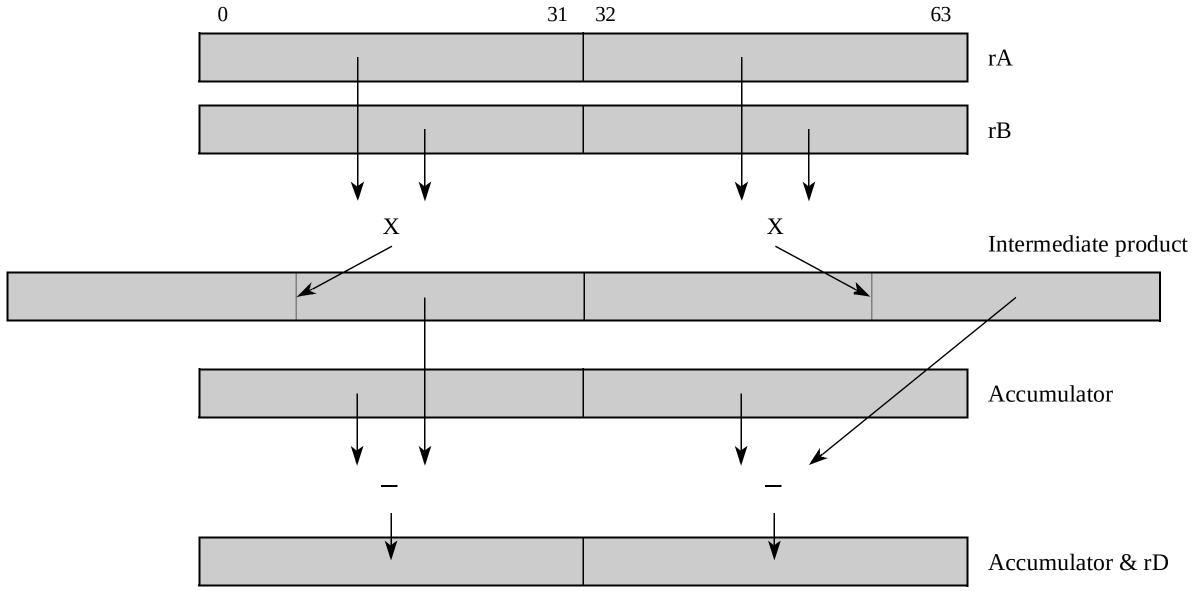


Figure 7-72. evmwlssianw

evmwlumi evmwlumi

Vector Multiply Word Low Unsigned, Modulo, Integer

evmwlumi **r**D,**r**A,**r**B (M=1, F=0, S=0,A=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 0100 1000

```
$temp1_{0:63} = rA_{0:31}$ *ui $rB_{0:31}$
$temp2_{0:63} = rA_{32:63}$ *ui $rB_{32:63}$

$rD_{0:31} = temp1_{32:63}$
$rD_{32:63} = temp2_{32:63}$
```

Each unsigned integer word element in **r**A is multiplied by the corresponding unsigned integer word element in **r**B. The lower 32-bits of the two 64-bit unsigned integer products are placed into the two word elements of **r**D.

NOTE

The low-order 32 bits of the product are independent of whether the word elements in **r**A and **r**B are treated as signed or unsigned 32-bit integers.

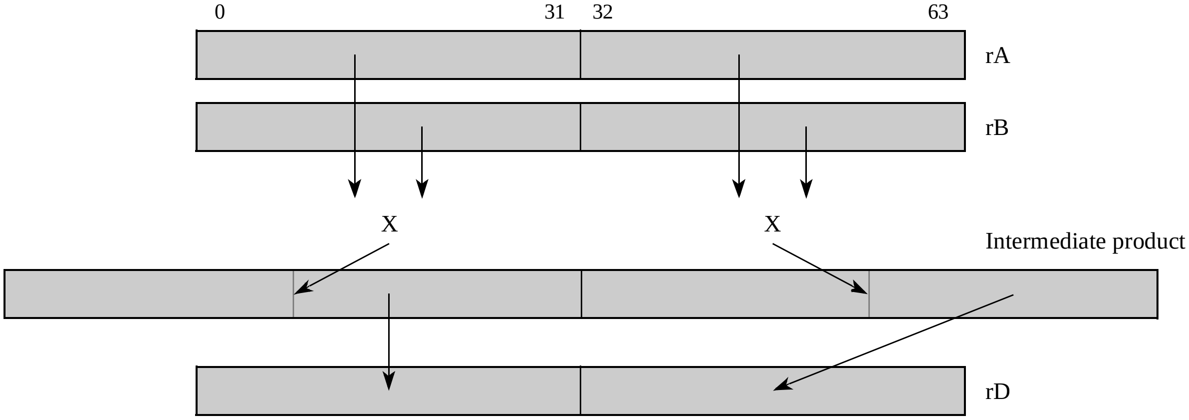


Figure 7-73. evmwlumi

evmwlumia evmwlumia

Vector Multiply Word Low Unsigned, Modulo, Integer, to Accumulator

evmwlumia **r**D,**r**A,**r**B (M=1, F=0, S=0,A=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 0110 1000

$temp1_{0:63} = rA_{0:31} *ui\ rB_{0:31}$
$temp2_{0:63} = rA_{32:63} *ui\ rB_{32:63}$

$rD_{0:31} = temp1_{32:63}$
$rD_{32:63} = temp2_{32:63}$

$ACC_{0:63} = rD_{0:63}$

Each unsigned integer word element in **r**A is multiplied by the corresponding unsigned integer word element in **r**B. The lower 32-bits of the two 64-bit unsigned integer products are placed into the two word elements of **r**D. The result in **r**D is also placed in the accumulator.

NOTE

The low-order 32bits of the product are independent of whether the word elements in **r**A and **r**B are treated as signed or unsigned 32-bit integers.

Other registers altered: ACC

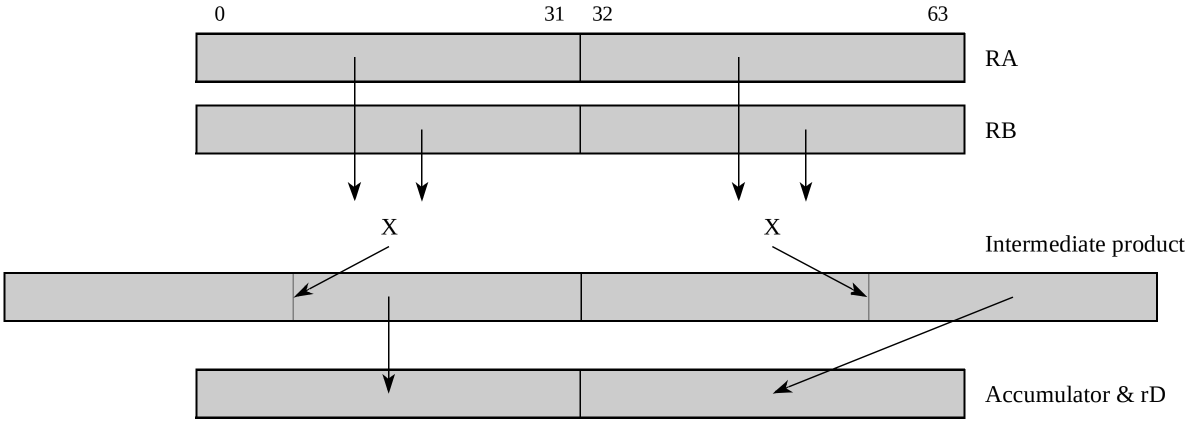


Figure 7-74. evmwlumia

evmwlumiaaw evmwlumiaaw

Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate in Words

evmwlumiaaw **r**D,**r**A,**r**B (M=1, F=0, S=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 0100 1000

$temp1_{0:63} = rA_{0:31} \text{ *ui } rB_{0:31}$
$temp2_{0:63} = rA_{32:63} \text{ *ui } rB_{32:63}$

$rD_{0:31} = ACC_{0:31} + temp1_{32:63}$
$rD_{32:63} = ACC_{32:63} + temp2_{32:63}$

$ACC_{0:63} = rD_{0:63}$

For each word element in the accumulator the following operations are performed in the order shown:

1. Each unsigned integer word element in **r**A is multiplied by the corresponding unsigned integer word element in **r**B.
2. The low 32 bits of the 64-bit intermediate product are added to the contents of the accumulator word and placed into the corresponding **r**D word. The result in **r**D is also placed in the accumulator.

NOTE

This instruction produces a valid result only if the intermediate product can be represented in the lower 32 bits.

Other registers altered: ACC

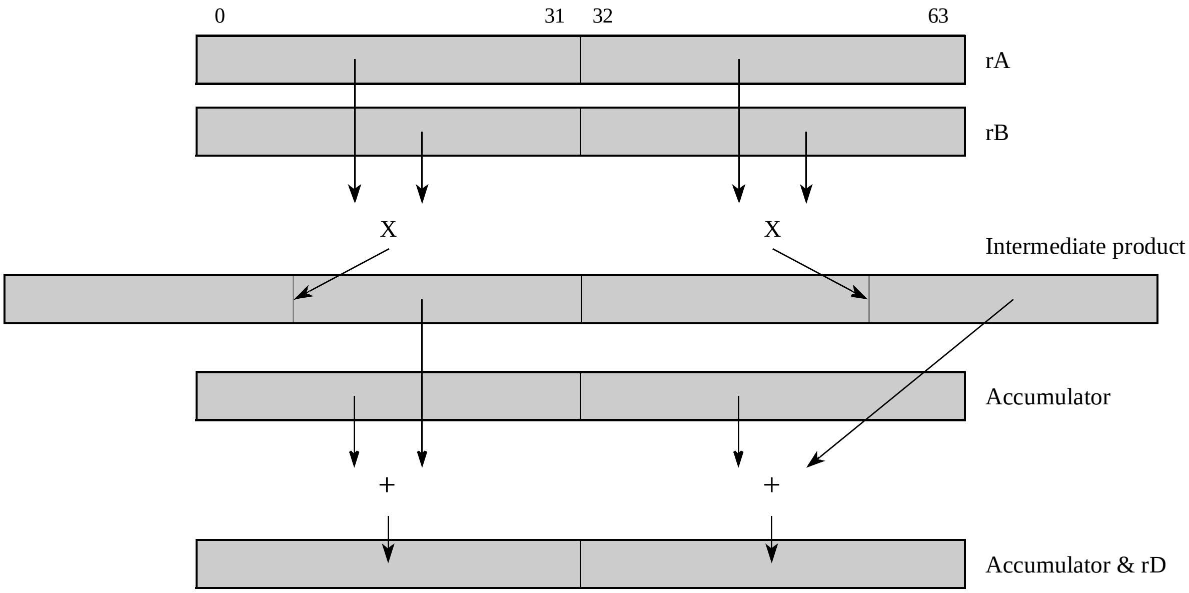


Figure 7-75. evmwlumiaaw

evmwlumianw evmwlumianw

Vector Multiply Word Low Unsigned, Modulo, Integer, and Accumulate Negative in Words

evmwlumianw **r**D,**r**A,**r**B (M=1, F=0, S=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 1100 1000

```
$temp1_{0:63} = rA_{0:31}$ *ui $rB_{0:31}$
$temp2_{0:63} = rA_{32:63}$ *ui $rB_{32:63}$

$rD_{0:31} = ACC_{0:31} - temp1_{32:63}$
$rD_{32:63} = ACC_{32:63} - temp2_{32:63}$

$ACC_{0:63} = rD_{0:63}$
```

For each word element in the accumulator the following operations are performed in the order shown:

1. Each unsigned integer word element in **r**A is multiplied by the corresponding unsigned integer word element in **r**B.
2. The low 32 bits of the 64-bit intermediate product are subtracted from the contents of the accumulator word and placed into the corresponding **r**D word. The result in **r**D is also placed in the accumulator.

NOTE

This instruction produces a valid result only if the intermediate product can be represented in the lower 32 bits.

Other registers altered: ACC

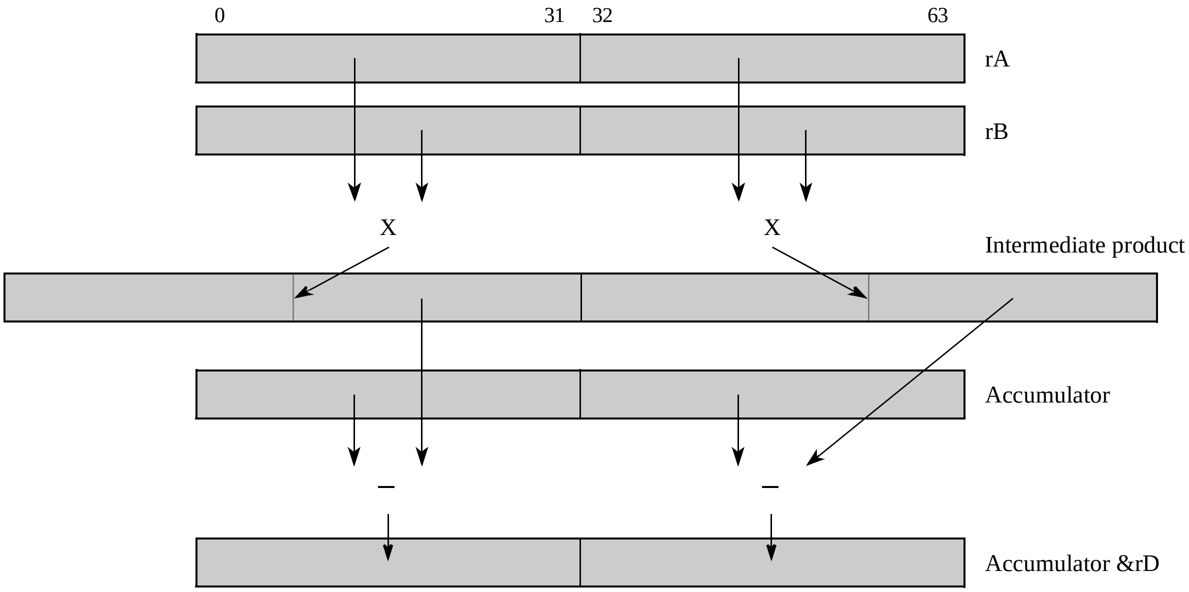


Figure 7-76. evmwlumianw

evmwlusiaaw evmwlusiaaw

Vector Multiply Word Low Unsigned, Saturate, Integer, and Accumulate in Words

evmwlusiaaw **r**D,**r**A,**r**B (M=0, F=0, S=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 0100 0000

```
$temp1_{0:63} = rA_{0:31}$ *ui $rB_{0:31}$
$temp2_{0:63} = rA_{32:63}$ *ui $rB_{32:63}$

$temp3_{0:32} = ACC_{0:31} + temp1_{32:63}$
$temp4_{0:32} = ACC_{32:63} + temp2_{32:63}$

$ovh = temp3_0$
$ovl = temp4_0$

$rD_{0:31}$ = SATURATE_ACC(ovh, 0xFFFFFFFF, $temp3_{1:32}$)
$rD_{32:63}$ = SATURATE_ACC(ovl, 0xFFFFFFFF, $temp4_{1:32}$)

$ACC_{0:31} = rD_{0:31}$
$ACC_{32:63} = rD_{32:63}$

$SPEFSCR_{OVH} = ovh$
$SPEFSCR_{OV} = ovl$
$SPEFSCR_{SOVH} = SPEFSCR_{SOVH} \mid ovh$
$SPEFSCR_{SOV} = SPEFSCR_{SOV} \mid ovl$
```

For each word element in the accumulator the following operations are performed in the order shown:

1. Each unsigned integer word element in **r**A is multiplied by the corresponding unsigned integer word element in **r**B.
2. The low 32 bits of the 64-bit intermediate product are added to the contents of the accumulator word to form a 33-bit intermediate sum.
3. If the intermediate sum has overflowed, 0xFFFF_FFFF is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word.
4. If there is an overflow from the addition, the overflow information is recorded in the SPEFSCR overflow and summary overflow bits.

NOTE

This instruction produces a valid result only if the intermediate product can be represented in the lower 32 bits.

Other registers altered: SPEFSCR, ACC

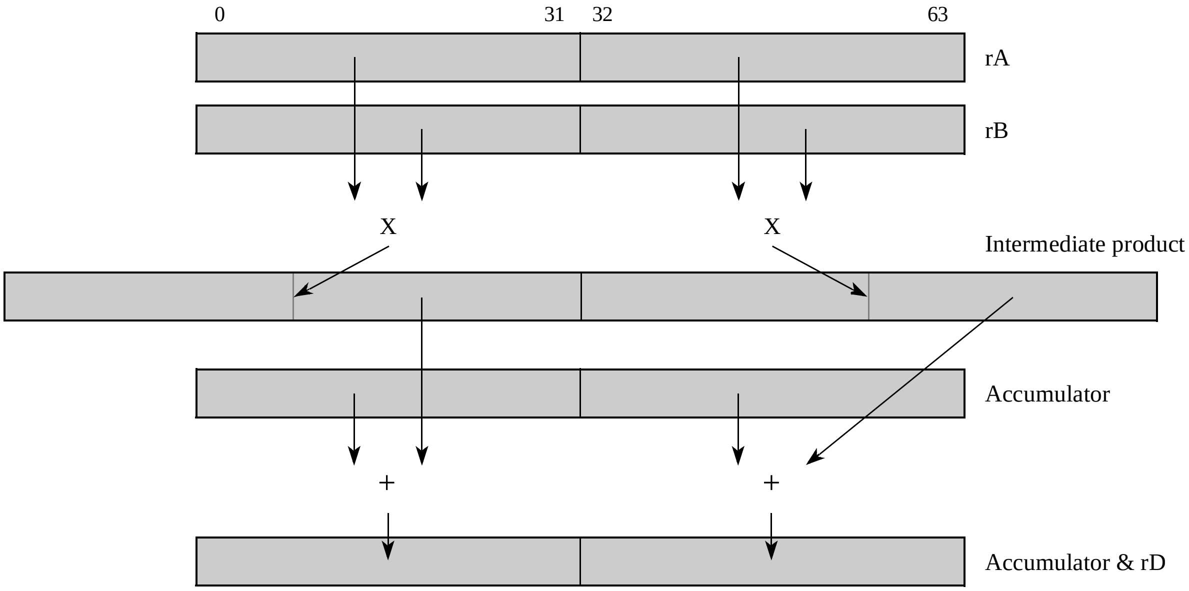


Figure 7-77. evmwlusiaaw

evmwlusianw evmwlusianw

Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate Negative in Words

evmwlusianw **r**D,**r**A,**r**B (M=0, F=0, S=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 1100 0000

```
$temp1_{0:63} = rA_{0:31}$ *ui $rB_{0:31}$
$temp2_{0:63} = rA_{32:63}$ *ui $rB_{32:63}$

$temp3_{0:32} = ACC_{0:31} - temp1_{32:63}$
$temp4_{0:32} = ACC_{32:63} - temp2_{32:63}$

$ovh = temp3_0$
$ovl = temp4_0$

$rD_{0:31}$ = SATURATE_ACC(ovh, 0x00000000, $temp3_{1:32}$)
$rD_{32:63}$ = SATURATE_ACC(ovl, 0x00000000, $temp4_{1:32}$)

$ACC_{0:31} = rD_{0:31}$
$ACC_{32:63} = rD_{32:63}$

$SPEFSCR_{OVH}$ = ovh
$SPEFSCR_{OV}$ = ovl
$SPEFSCR_{SOVH} = SPEFSCR_{SOVH}$ | ovh
$SPEFSCR_{SOV} = SPEFSCR_{SOV}$ | ovl
```

For each word element in the accumulator the following operations are performed in the order shown:

1. Each unsigned integer word element in **r**A is multiplied by the corresponding unsigned integer word element in **r**B.
2. The low 32 bits of the 64-bit intermediate product are subtracted from the contents of the accumulator word to form a 33-bit intermediate difference.
3. If the intermediate difference has underflowed, 0x00000000 is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word.
4. If there is an underflow from the difference, the underflow information is recorded in the SPEFSCR overflow and summary overflow bits.

NOTE

This instruction produces a valid result only if the intermediate product can be represented in the lower 32 bits.

Other registers altered: SPEFSCR, ACC

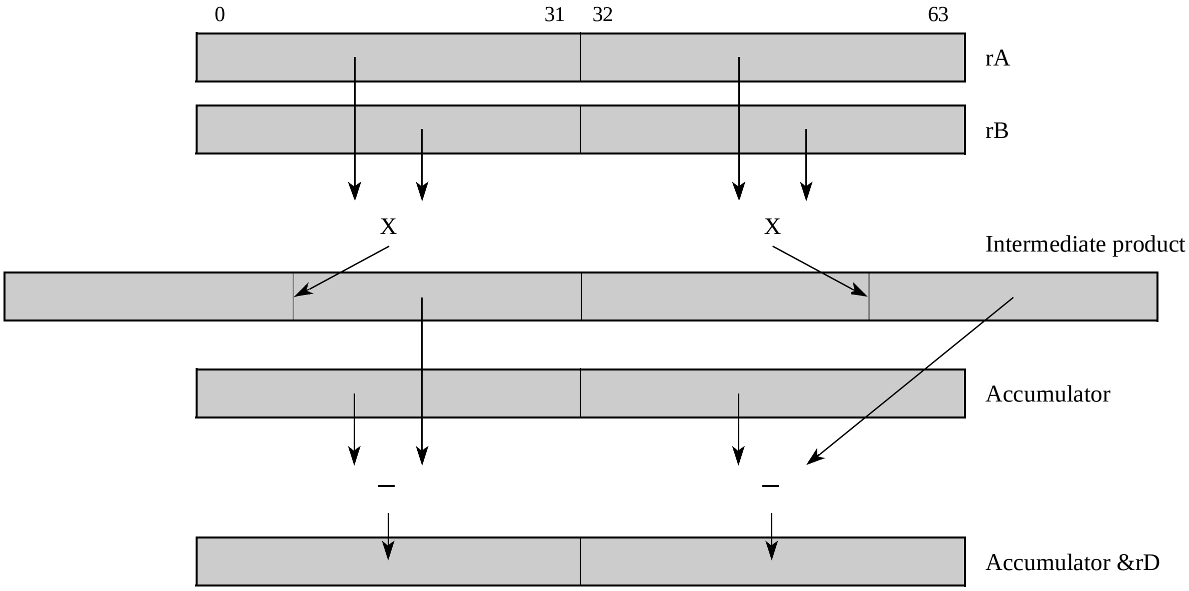


Figure 7-78. evmwlusianw

evmwsmf evmwsmf

Vector Multiply Word Signed, Modulo, Fractional

evmwsmf **r**D,**r**A,**r**B (M=1, F=1, S=1, A=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 0101 1011

$temp1_{0:64} = (rA_{32:63} * rB_{32:63}) \| 0$

$rD_{0:63} = temp1_{1:64}$

The low signed fractional word element in **r**A is multiplied by the corresponding low signed fractional word element in **r**B. Bits 1–63 of the 64-bit signed fractional product are padded on the right with a ‘0’. This result is placed in **r**D.

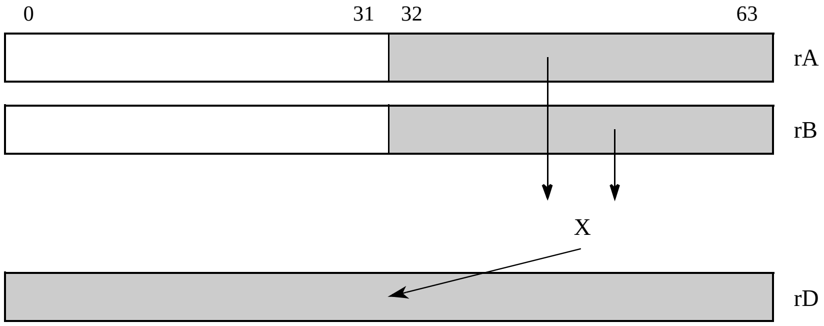


Figure 7-79. evmwsmf

evmwsmfa evmwsmfa

Vector Multiply Word Signed, Modulo, Fractional, to Accumulator

evmwsmfa **r**D,**r**A,**r**B (M=1, F=1, S=1, A=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 0111 1011

$\text{temp1}_{0:64} = (\text{rA}_{32:63} * \text{rB}_{32:63}) \parallel 0$

$\text{ACC}_{0:63} = \text{rD}_{0:63} = \text{temp1}_{1:64}$

The low signed fractional word element in **r**A is multiplied by the corresponding low signed fractional word element in **r**B. Bits 1–63 of the 64-bit signed fractional product are padded on the right with a ‘0’. This result is placed in **r**D. The result in **r**D is also placed in the accumulator.

Other registers altered: ACC

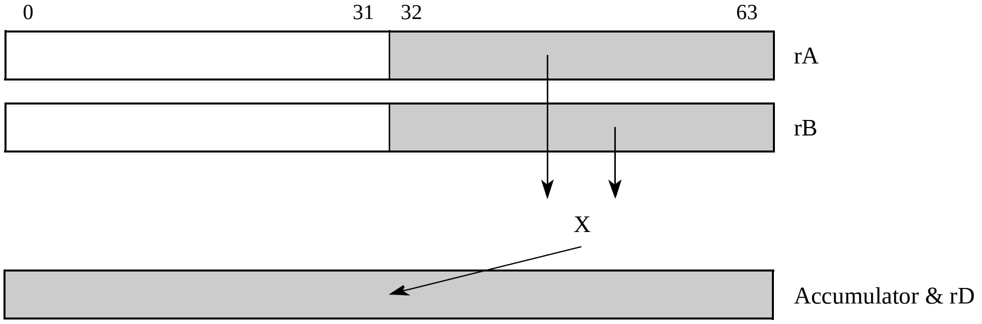


Figure 7-80. evmwsmfa

evmwsmfaa evmwsmfaa

Vector Multiply Word Signed, Modulo, Fractional, and Accumulate

evmwsmfaa **r**D,**r**A,**r**B (M=1, F=1, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 0101 1011

$\mathrm{temp1}_{0:64} = (\mathrm{rA}_{32:63} * \mathrm{rB}_{32:63}) \,\|\, 0$

$\mathrm{temp2}_{0:64} = \mathrm{ACC}_{0:63} + \mathrm{temp1}_{1:64}$

$\mathrm{ACC}_{0:63} = \mathrm{rD}_{0:63} = \mathrm{temp2}_{1:64}$

The low signed fractional word element in **r**A is multiplied by the corresponding low signed fractional word element in **r**B. Bits 1–63 of the 64-bit signed fractional product are padded on the right with a ‘0’, and this result is added to the contents of the 64-bit accumulator to form a 65-bit intermediate sum. The lower 64 bits of the intermediate sum are placed back into the accumulator and also written into **r**D.

Other registers altered: ACC

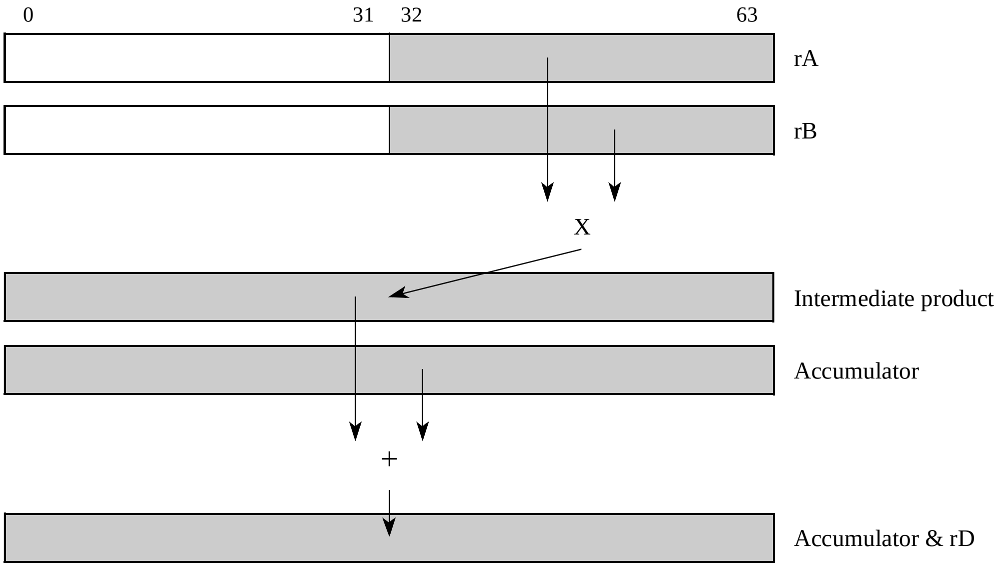


Figure 7-81. evmwsmfaa

evmwsmfan evmwsmfan

Vector Multiply Word Signed, Modulo, Fractional and Accumulate Negative

evmwsmfan **r**D,**r**A,**r**B (M=1, F=1, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 1101 1011

$temp1_{0:64} = (rA_{32:63} * rB_{32:63}) \| 0$

$temp2_{0:64} = ACC_{0:63} - temp1_{1:64}$

$ACC_{0:63} = rD_{0:63} = temp2_{1:64}$

The low signed fractional word element in **r**A is multiplied by the corresponding low signed fractional word element in **r**B. Bits 1–63 of the 64-bit signed fractional product are padded on the right with a ‘0’. This result is subtracted from the contents of the 64-bit accumulator to form a 65-bit intermediate difference. The lower 64 bits of the intermediate difference is placed back into the accumulator and also written into **r**D.

Other registers altered: ACC

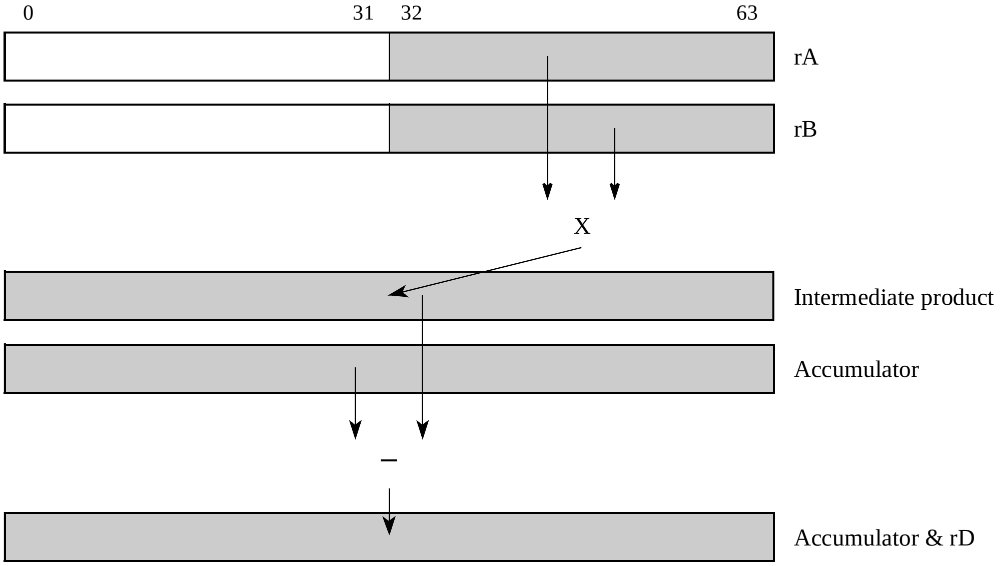


Figure 7-82. evmwsmfan

evmwsmi evmwsmi

Vector Multiply Word Signed, Modulo, Integer

evmwsmi **r**D,**r**A,**r**B (M=1, F=0, S=1, A=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 0101 1001

$temp_{0:63} = rA_{32:63} *si\ rB_{32:63}$

$ACC_{0:63} = rD_{0:63} = temp_{0:63}$

The low signed integer word element in **r**A is multiplied by the corresponding low signed integer word element in **r**B. The 64-bit signed integer product is placed in **r**D.

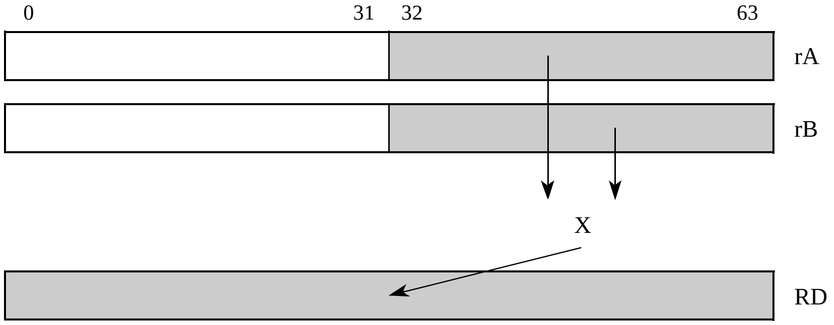


Figure 7-83. evmwsmi

evmwsmia

evmwsmia

Vector Multiply Word Signed, Modulo, Integer, to Accumulator

evmwsmia **r**D,**r**A,**r**B (M=1, F=0, S=1, A=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 0111 1001

$\mathrm{temp}_{0:63} = \mathrm{rA}_{32:63} *\mathrm{si}\ \mathrm{rB}_{32:63}$

$\mathrm{ACC}_{0:63} = \mathrm{rD}_{0:63} = \mathrm{temp}_{0:63}$

The low signed integer word element in **r**A is multiplied by the corresponding low signed integer word element in **r**B. The 64-bit signed integer product is placed in **r**D. The result in **r**D is also placed in the accumulator.

Other registers altered: ACC

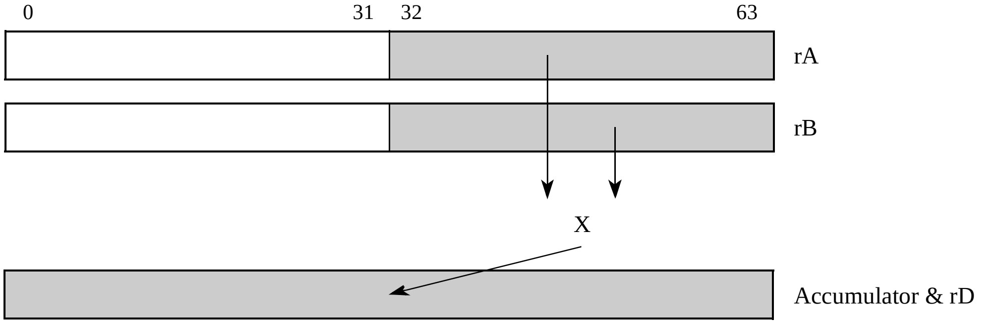


Figure 7-84. evmwsmia

evmwsmiaa evmwsmiaa

Vector Multiply Word Signed, Modulo, Integer, and Accumulate

evmwsmiaa **r**D,**r**A,**r**B (M=1, F=0, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 0101 1001

$temp1_{0:63} = rA_{32:63}$ *si $rB_{32:63}$

$temp2_{0:64} = ACC_{0:63} + temp1_{0:63}$

$ACC_{0:63} = rD_{0:63} = temp2_{1:64}$

The low signed integer word element in **r**A is multiplied by the corresponding low signed integer word element in **r**B. The intermediate product is added to the contents of the 64-bit accumulator to form a 65-bit intermediate sum. The lower 64 bits of the intermediate sum is placed back into the accumulator and also written into **r**D.

Other registers altered: ACC

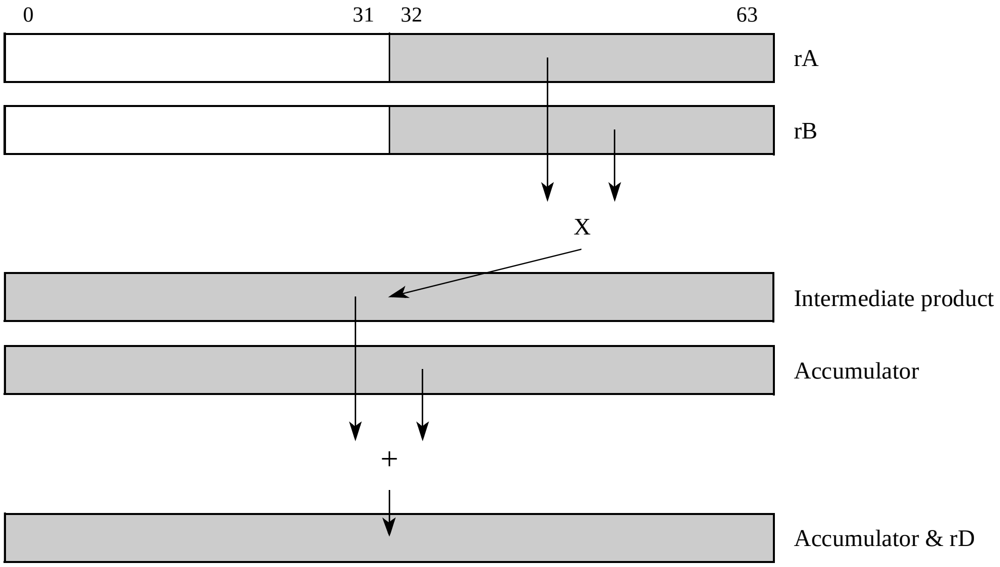


Figure 7-85. evmwsmiaa

evmwsmian evmwsmian

Vector Multiply Word Signed, Modulo, Integer and Accumulate Negative

evmwsmian **r**D,**r**A,**r**B (M=1, F=0, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 1101 1001

$temp1_{0:63} = rA_{32:63}$ *si $rB_{32:63}$

$temp2_{0:64} = ACC_{0:63} - temp1_{0:63}$

$ACC_{0:63} = rD_{0:63} = temp2_{1:64}$

The low signed integer word element in **r**A is multiplied by the corresponding low signed integer word element in **r**B. The intermediate product is subtracted from the contents of the 64-bit accumulator to form a 65-bit intermediate difference. The lower 64 bits of the intermediate difference is placed back into the accumulator and also written into **r**D.

Other registers altered: ACC

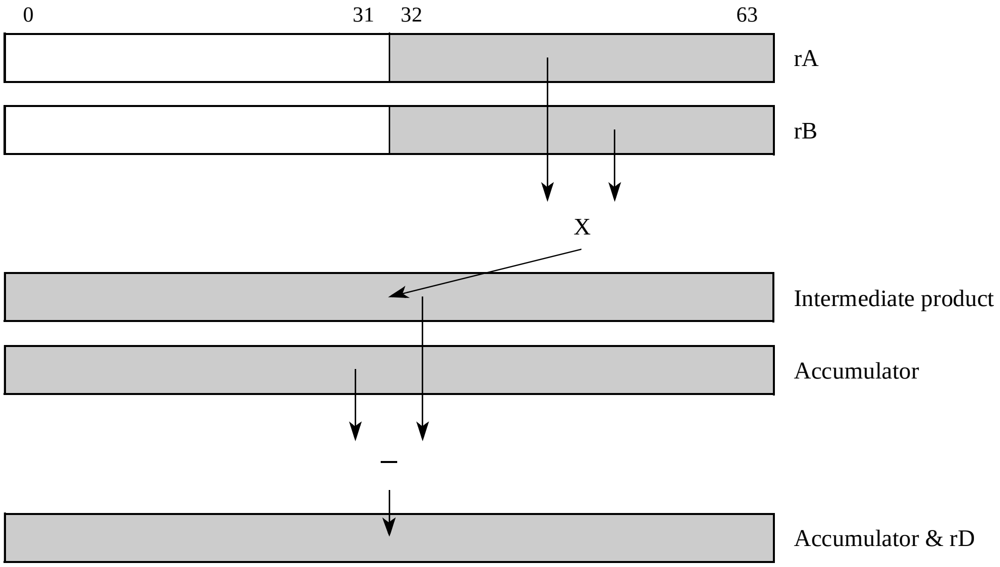


Figure 7-86. evmwsmian

evmwssf evmwssf

Vector Multiply Word Signed, Saturate, Fractional

evmwssf **r**D,**r**A,**r**B (M=0, F=1, S=1, A=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 0101 0011

```
$temp_{0:64} = (rA_{32:63} * rB_{32:63}) || 0$
$movl = temp_0 \oplus temp_1$
$rD_{0:63} = SATURATE(movh, 0x7FFFFFFFFFFFFFFF, temp_{1:64})$
$SPEFSCR_{OVH} = 0$
$SPEFSCR_{OV} = movl$
$SPEFSCR_{SOV} = SPEFSCR_{SOV} | movl$
```

The low signed fractional word element in **r**A is multiplied by the corresponding low signed fractional word element in **r**B. The 64-bit signed fractional product is placed in **r**D. If the inputs are –1.0 and –1.0 the result is saturated to the most positive signed fraction (0x7FFF_FFFF_FFFF_FFFF). If saturation occurs, the overflow and summary overflow bits are recorded.

Other registers altered: SPEFSCR

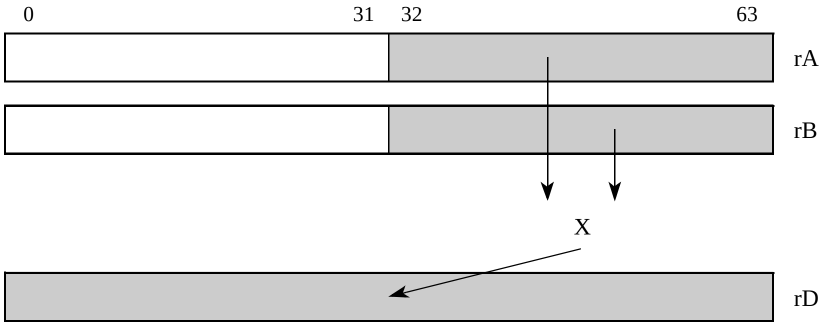


Figure 7-87. evmwssf

evmwssfa

Vector Multiply Word Signed, Saturate, Fractional, to Accumulator

evmwssfa **r**D,**r**A,**r**B (M=0, F=1, S=1, A=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 0111 0011

```
$temp_{0:64} = (rA_{32:63} * rB_{32:63}) || 0$
$movl = temp_0 \oplus temp_1$
$ACC_{0:63} = rD_{0:63} = SATURATE(movh, 0x7FFFFFFFFFFFFFFF, temp_{1:64})$
$SPEFSCR_{OVH} = 0$
$SPEFSCR_{OV} = movl$
$SPEFSCR_{SOV} = SPEFSCR_{SOV} | movl$
```

The low signed fractional word element in **r**A is multiplied by the corresponding low signed fractional word element in **r**B. The 64-bit signed fractional product is placed in **r**D. If the inputs are –1.0 and –1.0, the result is saturated to the most positive signed fraction (0x7FFF_FFFF_FFFF_FFFF). If saturation occurs the overflow and summary overflow bits are recorded. The result in **r**D is also placed in the accumulator.

Other registers altered: SPEFSCR, ACC

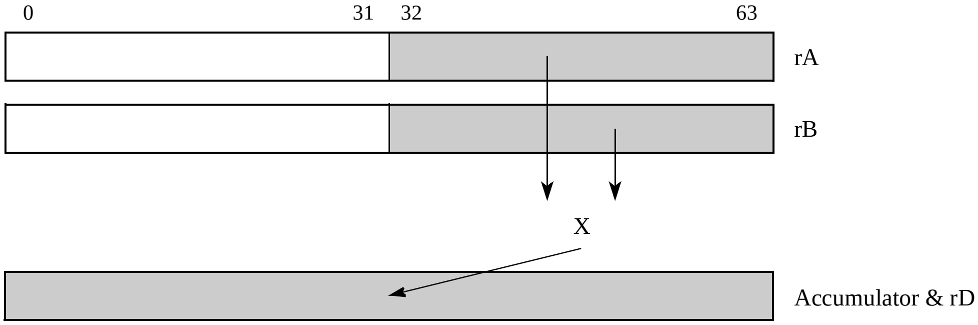


Figure 7-88. evmwssfa

evmwssfaa evmwssfaa

Vector Multiply Word Signed, Saturate, Fractional, and Accumulate

evmwssfaa **r**D,**r**A,**r**B (M=0, F=1, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 0101 0011

```
$temp1_{0:64} = (rA_{32:63} * rB_{32:63}) || 0$
$mov = temp1_0 \oplus temp1_1$

$temp2_{0:63} = SATURATE(mov, 0x7FFFFFFFFFFFFFFF, temp1_{1:64})$
$temp3_{0:64} = \{ACC_0,ACC_{0:63}\} + \{temp2_0,temp2_{0:63}\}$
$ov = temp3_0 \oplus temp3_1$

$rD_{0:63} = SATURATE\_ACC(ov, temp3_0, 0x8000000000000000, 0x7FFFFFFFFFFFFFFF, temp3_{1:64})$
$ACC_{0:63} = rD_{0:63}$

$SPEFSCR_{OV} = mov | ov$
$SPEFSCR_{OVH} = 0$
$SPEFSCR_{SOV} = SPEFSCR_{SOV} | mov | ov$
```

The low signed fractional word element in **r**A is multiplied by the corresponding low signed fractional word element in **r**B. If the inputs are –1.0 and –1.0, the product is saturated to the most positive signed fraction (0x7FFF_FFFF_FFFF_FFFF). The 64-bit intermediate product is shifted left by one bit (to eliminate the redundant sign bit) and padded on the right with a ‘0’, and this value is then added to the contents of the 64-bit accumulator to form an intermediate sum.

If the intermediate sum has overflowed, the appropriate saturation value (0x7FFF_FFFF_FFFF_FFFF if positive overflow or 0x8000_0000_0000_0000 if negative overflow) is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 64 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word. The overflow and summary overflow bits are recorded to indicate occurrence of saturation on either the multiply or the addition.

Other registers altered: SPEFSCR, ACC

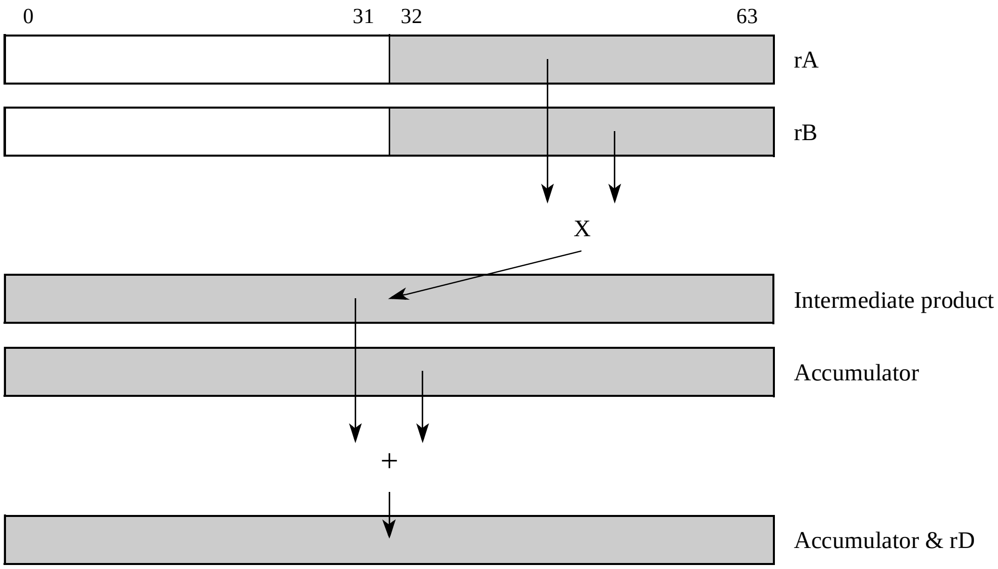


Figure 7-89. evmwssfaa

evmwssfan

Vector Multiply Word Signed, Saturate, Fractional and Accumulate Negative

evmwssfan **r**D,**r**A,**r**B (M=0, F=1, S=1)

0 … 5	6 … 10	11 … 15	16 … 20	21 … 31
4	RD	RA	RB	101 1101 0011

```
$temp1_{0:64} = (rA_{32:63} * rB_{32:63}) \| 0$
$mov = temp1_0 \oplus temp1_1$

$temp2_{0:63} = SATURATE(mov, 0x7FFFFFFFFFFFFFFF, temp1_{1:64})$
$temp3_{0:64} = \{ACC_0,ACC_{0:63}\} - \{temp2_0,temp2_{0:63}\}$
$ov = temp3_0 \oplus temp3_1$

$rD_{0:63} = SATURATE\_ACC(ov, temp3_0, 0x8000000000000000, 0x7FFFFFFFFFFFFFFF, temp3_{1:64})$
$ACC_{0:63} = rD_{0:63}$

$SPEFSCR_{OV} = mov | ov$
$SPEFSCR_{OVH} = 0$
$SPEFSCR_{SOV} = SPEFSCR_{SOV} | mov | ov$
```

The low signed fractional word element in **r**A is multiplied by the corresponding low signed fractional word element in **r**B. If the inputs are –1.0 and –1.0 the product is saturated to the most positive signed fraction (0x7FFF_FFFF_FFFF_FFFF). The 64-bit intermediate product is shifted left by one bit (to eliminate the redundant sign bit) and padded on the right with a ‘0’, and this value is then subtracted from the contents of the 64-bit accumulator to form an intermediate sum. If the intermediate difference has overflowed, the appropriate saturation value (0x7FFF_FFFF_FFFF_FFFF if positive overflow or 0x8000_0000_0000_0000 if negative overflow) is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 64 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word. The overflow and summary overflow bits are recorded to indicate occurrence of saturation either the multiply or the subtraction.

Other registers altered: SPEFSCR, ACC

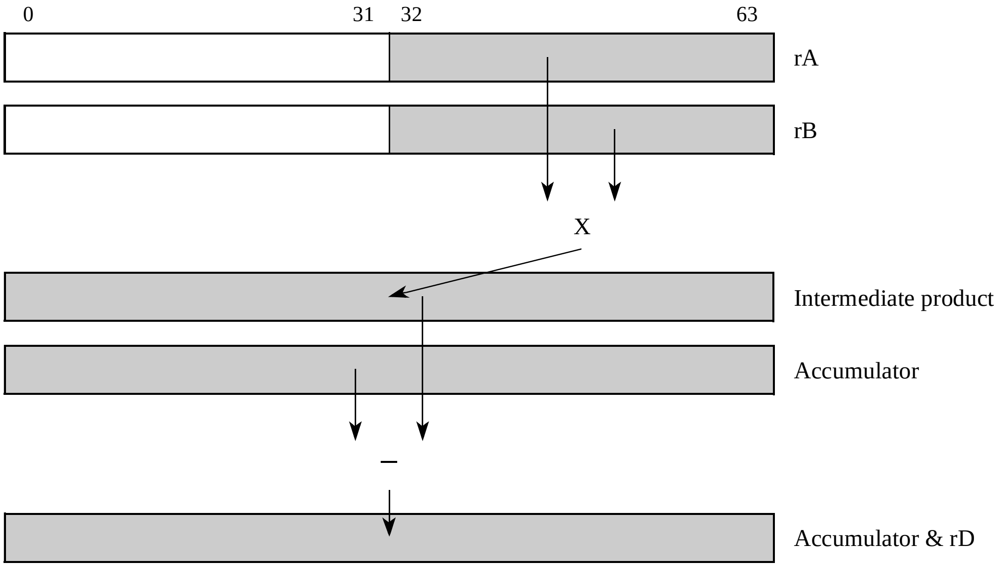


Figure 7-90. evmwssfan

evmwumi evmwumi

Vector Multiply Word Unsigned, Modulo, Integer

evmwumi **r**D,**r**A,**r**B (M=1, F=0, S=0, A=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 0101 1000

$temp_{0:63} = rA_{32:63} *ui\ rB_{32:63}$

$ACC_{0:63} = rD_{0:63} = temp_{0:63}$

The low unsigned integer word element in **r**A is multiplied by the corresponding low unsigned integer word element in **r**B. The 64-bit unsigned integer product is placed in **r**D.

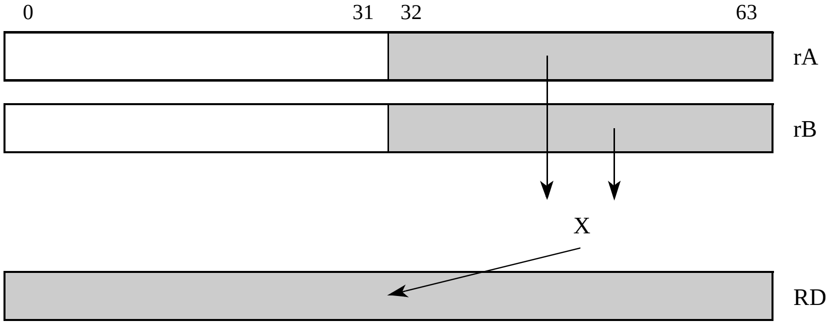


Figure 7-91. evmwumi

evmwumia evmwumia

Vector Multiply Word Unsigned, Modulo, Integer, to Accumulator

evmwumia **r**D,**r**A,**r**B (M=1, F=0, S=0, A=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 0111 1000

$temp_{0:63} = rA_{32:63}$ *ui $rB_{32:63}$

$ACC_{0:63} = rD_{0:63} = temp_{0:63}$

The low unsigned integer word element in **r**A is multiplied by the corresponding low unsigned integer word element in **r**B. The 64-bit unsigned integer product is placed in **r**D. The result in **r**D is also placed in the accumulator.

Other registers altered: ACC

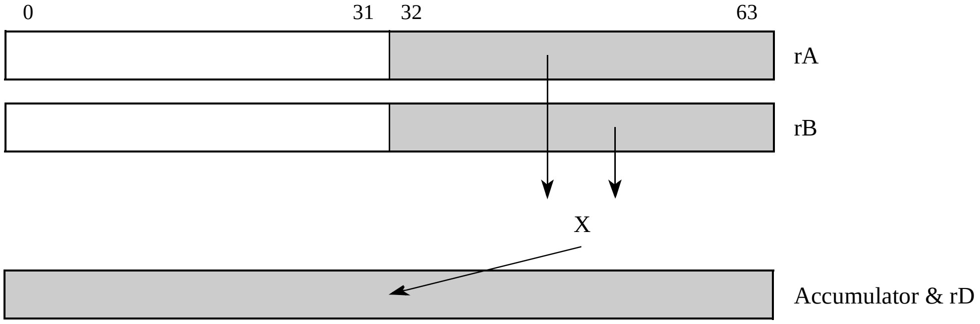


Figure 7-92. evmwumia

evmwumiaa evmwumiaa

Vector Multiply Word Unsigned, Modulo, Integer and Accumulate

evmwumiaa **r**D,**r**A,**r**B (M=1, F=0, S=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 0101 1000

$temp1_{0:63} = rA_{32:63} \text{ *ui } rB_{32:63}$

$temp2_{0:64} = ACC_{0:63} + temp1_{0:63}$

$ACC_{0:63} = rD_{0:63} = temp2_{1:64}$

The low unsigned integer word element in **r**A is multiplied by the corresponding low unsigned integer word element in **r**B. The intermediate product is added to the contents of the 64-bit accumulator to form a 65-bit intermediate sum. The lower 64 bits of the intermediate sum is placed back into the accumulator and also written into **r**D.

Other registers altered: ACC

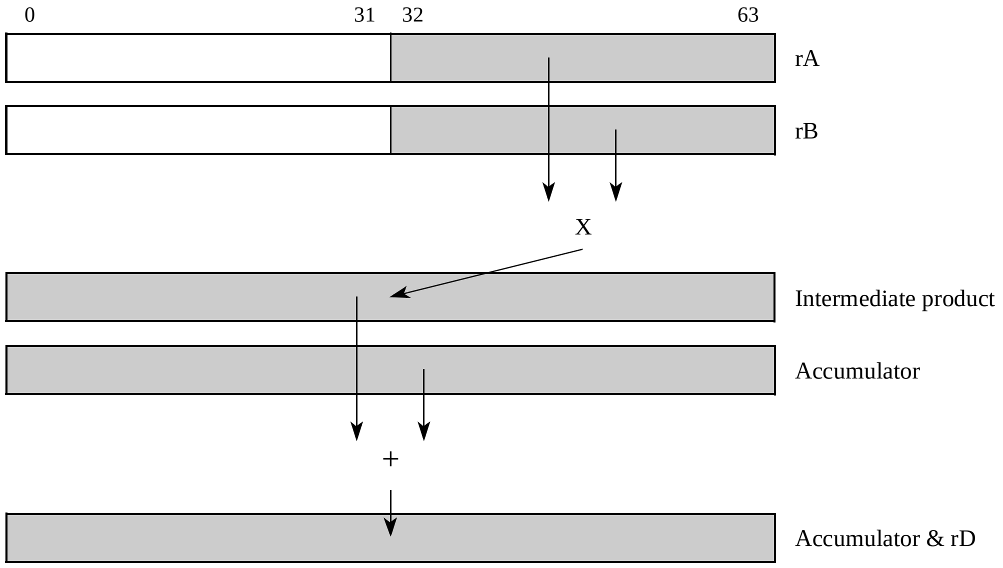


Figure 7-93. evmwumiaa

evmwumian evmwumian

Vector Multiply Word Unsigned, Modulo, Integer and Accumulate Negative

evmwumian **r**D,**r**A,**r**B (M=1, F=0, S=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 1101 1000

$temp1_{0:63} = rA_{32:63} *ui\ rB_{32:63}$

$temp2_{0:64} = ACC_{0:63} - temp1_{0:63}$

$ACC_{0:63} = rD_{0:63} = temp2_{1:64}$

The low unsigned integer word element in **r**A is multiplied by the corresponding low unsigned integer word element in **r**B. The intermediate product is subtracted from the contents of the 64-bit accumulator to form a 65-bit intermediate difference. The lower 64 bits of the intermediate difference is placed back into the accumulator and also written into **r**D.

Other registers altered: ACC

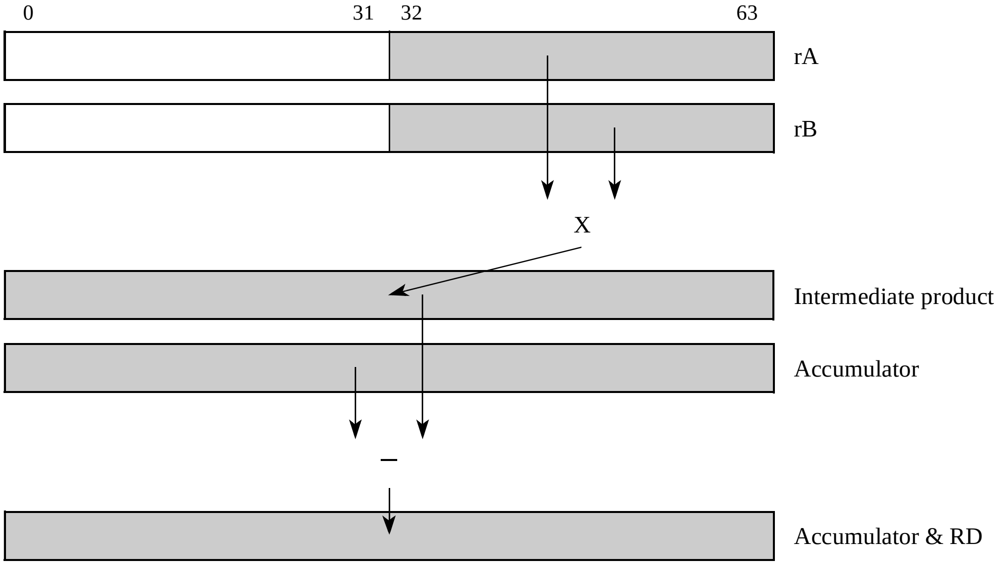


Figure 7-94. evmwumian

7.4.3 Add/Subtract Word to Accumulator Instructions

The following instructions perform addition and subtraction, with and without accumulates, using signed or unsigned integer or fractional operands, with optional saturation.

evaddsmiaaw evaddsmiaaw

Vector Add Signed, Modulo, Integer to Accumulator Word

evaddsmiaaw **r**D,**r**A (M=1, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	0000 0	100 1100 1001

$rD_{0:31} = ACC_{0:31} + rA_{0:31}$
$rD_{32:63} = ACC_{32:63} + rA_{32:63}$

$ACC_{0:63} = rD_{0:63}$

Each word element in **r**A is added to the corresponding word element in the accumulator and placed into the corresponding **r**D word. The result in **r**D is also placed in the accumulator.

Other registers altered: ACC

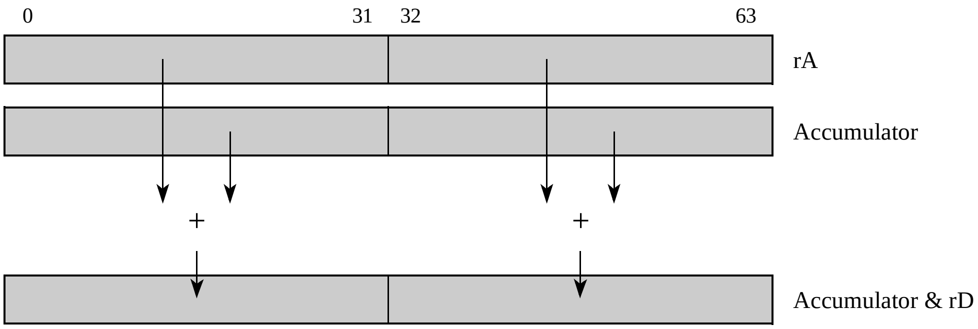


Figure 7-95. evaddsmiaaw

evaddssiaaw evaddssiaaw

Vector Add Signed, Saturate, Integer to Accumulator Word

evaddssiaaw **r**D,**r**A (M=0, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	0000 0	100 1100 0001

```
$temp1_{0:32} = EXTS(ACC_{0:31}) + EXTS(rA_{0:31})$
$temp2_{0:32} = EXTS(ACC_{32:63}) + EXTS(rA_{32:63})$

$ovh = temp1_0 \oplus temp1_1$
$ovl = temp2_0 \oplus temp2_1$

$rD_{0:31} = SATURATE\_ACC(ovh, temp1_0, 0x80000000, 0x7FFFFFFF, temp1_{1:32})$
$rD_{32:63} = SATURATE\_ACC(ovl, temp2_0, 0x80000000, 0x7FFFFFFF, temp2_{1:32})$

$ACC_{0:31} = rD_{0:31}$
$ACC_{32:63} = rD_{32:63}$

$SPEFSCR_{OVH} = ovh$
$SPEFSCR_{OV} = ovl$
$SPEFSCR_{SOVH} = SPEFSCR_{SOVH} | ovh$
$SPEFSCR_{SOV} = SPEFSCR_{SOV} | ovl$
```

Each word element in **r**A is added to the corresponding word element in the accumulator to form 33-bit intermediate sum. If the intermediate sum has overflowed, the appropriate saturation value (0x7FFF_FFFF if positive overflow or 0x8000_0000 if negative overflow) is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word.

If there is an overflow from the addition, the overflow information is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR, ACC

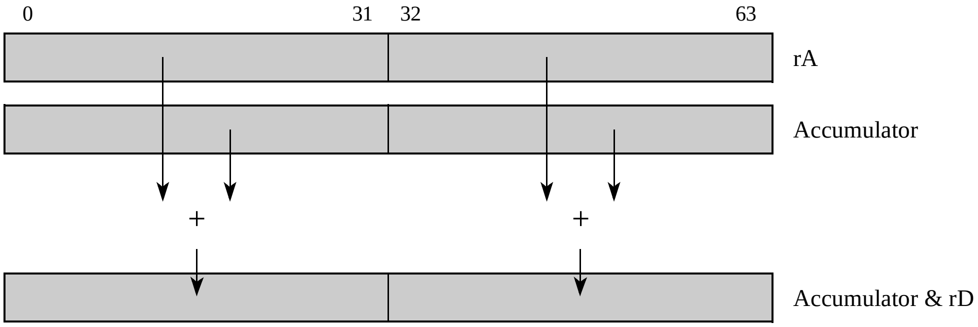


Figure 7-96. evaddssiaaw

evaddumiaaw evaddumiaaw

Vector Add Unsigned, Modulo, Integer to Accumulator Word

evaddumiaaw **r**D,**r**A (M=1, S=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	0000 0	100 1100 1000

$rD_{0:31} = ACC_{0:31} + rA_{0:31}$
$rD_{32:63} = ACC_{32:63} + rA_{32:63}$

$ACC_{0:63} = rD_{0:63}$

Each word element in **r**A is added to the corresponding word element in the accumulator and placed into the corresponding **r**D word. The result in **r**D is also placed in the accumulator.

Other registers altered: ACC

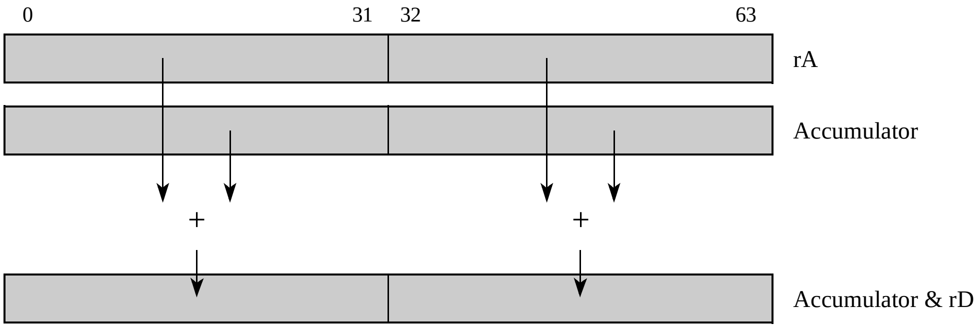


Figure 7-97. evaddumiaaw

evaddusiaaw evaddusiaaw

Vector Add Unsigned, Saturate, Integer to Accumulator Word

evaddusiaaw **r**D,**r**A (M=0, S=0)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	0000 0	100 1100 0000

```
$temp1_{0:32} = EXTZ(ACC_{0:31}) + EXTZ(rA_{0:31})$
$temp2_{0:32} = EXTZ(ACC_{32:63}) + EXTZ(rA_{32:63})$

$ovh = temp1_0$
$ovl = temp2_0$

$rD_{0:31} = SATURATE(ovh, 0xFFFFFFFF, temp1_{1:32})$
$rD_{32:63} = SATURATE(ovl, 0xFFFFFFFF, temp2_{1:32})$

$ACC_{0:31} = rD_{0:31}$
$ACC_{32:63} = rD_{32:63}$

$SPEFSCR_{OVH} = ovh$
$SPEFSCR_{OV} = ovl$
$SPEFSCR_{SOVH} = SPEFSCR_{SOVH} | ovh$
$SPEFSCR_{SOV} = SPEFSCR_{SOV} | ovl$
```

Each word element in **r**A is added to the corresponding word element in the accumulator to form 33-bit intermediate sum. If the intermediate sum has overflowed, 0xFFFF_FFFF is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word.

If there is an overflow from the addition, the overflow information is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR, ACC

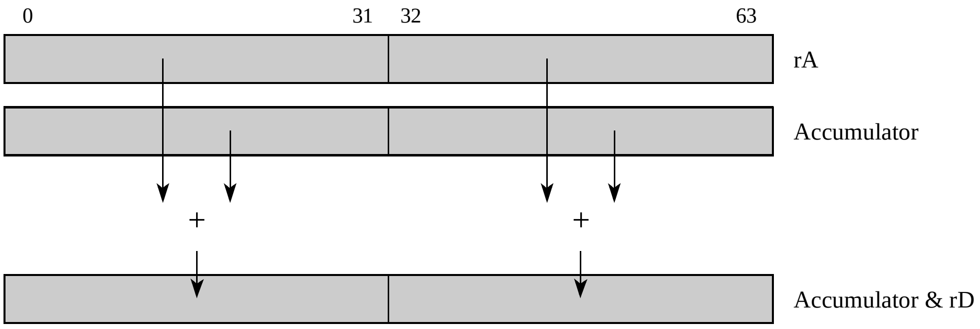


Figure 7-98. evaddusiaaw

evsubfsmiaaw evsubfsmiaaw

Vector Subtract Signed, Modulo, Integer to Accumulator Word

evsubfsmiaaw **r**D,**r**A (M=1, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	0000 0	100 1100 1011

$rD_{0:31} = ACC_{0:31} - rA_{0:31}$
$rD_{32:63} = ACC_{32:63} - rA_{32:63}$

$ACC_{0:63} = rD_{0:63}$

Each word element in **r**A is subtracted from the corresponding word element in the accumulator and placed into the corresponding **r**D word. The result in **r**D is also placed in the accumulator.

Other registers altered: ACC

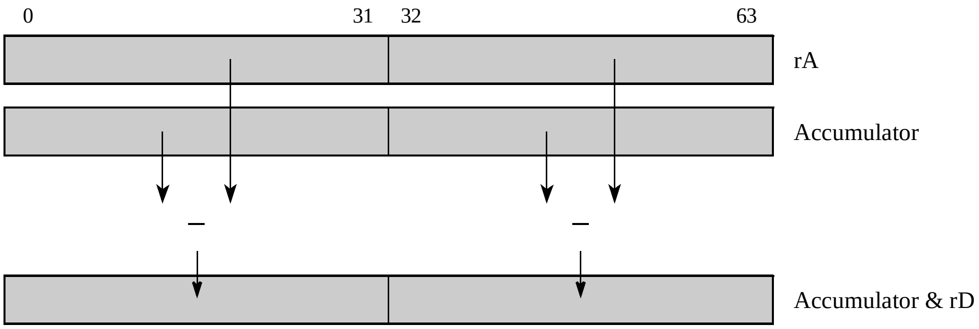


Figure 7-99. evsubfsmiaaw

evsubfssiaaw evsubfssiaaw

Vector Subtract Signed, Saturate, Integer to Accumulator Word

evsubfssiaaw **r**D,**r**A (M=0, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	0000 0	100 1100 0011

```
$temp1_{0:32} = EXTS(ACC_{0:31}) - EXTS(rA_{0:31})$
$temp2_{0:32} = EXTS(ACC_{32:63}) - EXTS(rA_{32:63})$

$ovh = temp1_0 \oplus temp1_1$
$ovl = temp2_0 \oplus temp2_1$

$rD_{0:31} = SATURATE\_ACC(ovh, temp1_0, 0x80000000, 0x7FFFFFFF, temp1_{1:32})$
$rD_{32:63} = SATURATE\_ACC(ovl, temp2_0, 0x80000000, 0x7FFFFFFF, temp2_{1:32})$

$ACC_{0:31} = rD_{0:31}$
$ACC_{32:63} = rD_{32:63}$

$SPEFSCR_{OVH} = ovh$
$SPEFSCR_{OV} = ovl$
$SPEFSCR_{SOVH} = SPEFSCR_{SOVH} | ovh$
$SPEFSCR_{SOV} = SPEFSCR_{SOV} | ovl$
```

Each word element in **r**A is subtracted from the corresponding word element in the accumulator to form 33-bit intermediate difference. If the intermediate difference has overflowed, the appropriate saturation value (0x7FFF_FFFF if positive overflow or 0x8000_0000 if negative overflow) is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word.

If there is an overflow from the subtraction, the overflow information is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR, ACC

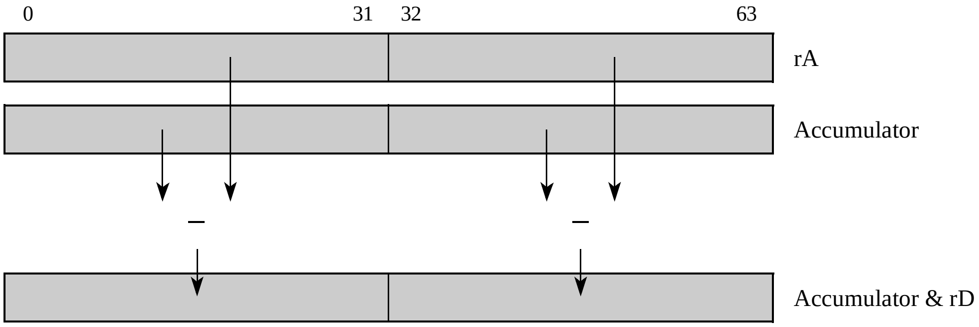


Figure 7-100. evsubfssiaaw

evsubfumiaaw evsubfumiaaw

Vector Subtract Unsigned, Modulo, Integer to Accumulator Word

evsubfumiaaw **r**D,**r**A (M=1, S=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	0000 0	100 1100 1010

$rD_{0:31} = ACC_{0:31} - rA_{0:31}$
$rD_{32:63} = ACC_{32:63} - rA_{32:63}$

$ACC_{0:63} = rD_{0:63}$

Each word element in **r**A is subtracted from the corresponding word element in the accumulator and placed into the corresponding **r**D word. The result in **r**D is also placed in the accumulator.

Other registers altered: ACC

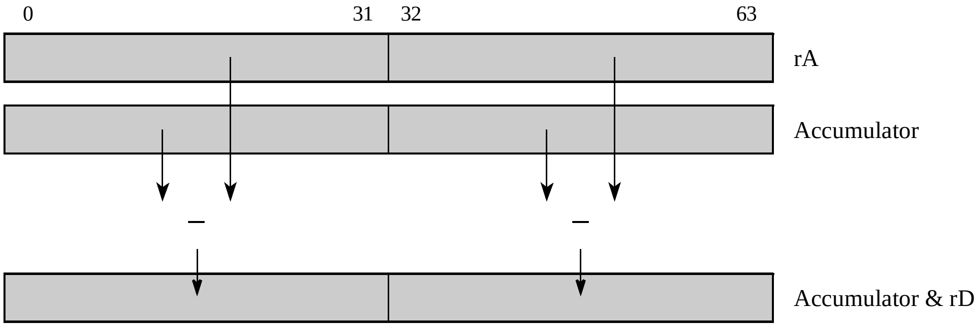


Figure 7-101. evsubfumiaaw

evsubfusiaaw evsubfusiaaw

Vector Subtract Unsigned, Saturate, Integer to Accumulator Word

evsubfusiaaw **r**D,**r**A (M=0, S=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	0000 0	100 1100 0010

```
$temp1_{0:32} = EXTZ(ACC_{0:31}) - EXTZ(rA_{0:31})$
$temp2_{0:32} = EXTZ(ACC_{32:63}) - EXTZ(rA_{32:63})$

$ovh = temp1_0$
$ovl = temp2_0$

$rD_{0:31} = SATURATE(ovh, 0x00000000, temp1_{1:32})$
$rD_{32:63} = SATURATE(ovl, 0x00000000, temp2_{1:32})$

$ACC_{0:31} = rD_{0:31}$
$ACC_{32:63} = rD_{32:63}$

$SPEFSCR_{OVH} = ovh$
$SPEFSCR_{OV} = ovl$
$SPEFSCR_{SOVH} = SPEFSCR_{SOVH} | ovh$
$SPEFSCR_{SOV} = SPEFSCR_{SOV} | ovl$
```

Each word element in **r**A is subtracted from the corresponding word element in the accumulator to form 33-bit intermediate difference. If the intermediate difference has underflowed, 0x0000_0000 is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word.

If there is an underflow from the subtraction, the underflow information is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR, ACC

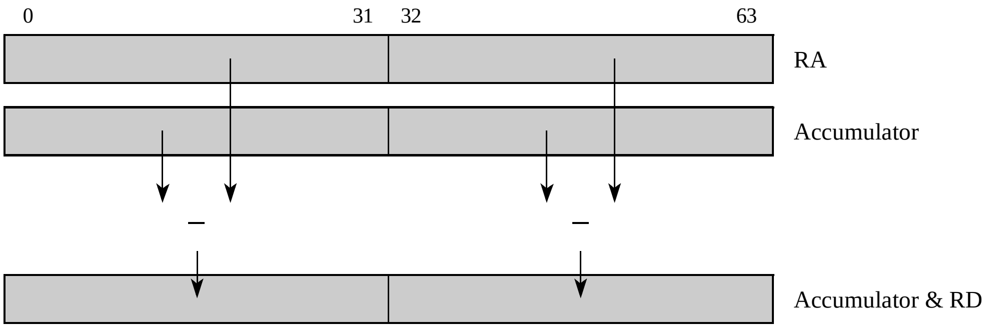


Figure 7-102. evsubfusiaaw

7.4.4 Initializing and Reading the Accumulator

To read the accumulator contents into a register, a multiply-accumulate instruction where one of its operands is a zero should be used, as the following sequence shows:

```
evxor RD, RD, RD      // Zero the contents of RD, not necessary if
                      // a zero is available in some register.
evmwumiaa RD, RD, RD // Multiply 0 with 0, add the 0 result to
                      // accumulator and store back the value in acc and RD
```

To initialize the accumulator, the **evmra** instruction is used, as shown in Figure 7-103.

evmra **r**D,**r**A

0 5	6 10	11 15	16 20	21 31
4	RD	RA	0000 0	100 1100 0100

Figure 7-103. Move Register to Accumulator (evmra)

$RD_{0:63} = acc_{0:63} = RA_{0:63}$

The contents of **r**A are written into the accumulator and copied into **r**D. This is the method for initializing the accumulator.

7.5 SPE Vector Load/Store Instructions

SPE Vector load and store instructions are provided with a variety of options. The mnemonics are formed as follows:

ev{l,st}<X><Y>[Z]x

- X specifies the size of the load
- Y specifies the size of data packed into the value being loaded. Thus **evldhx** specified a load that brings in a double-word composed of four half words.
- Z specifies the operation to be performed such as unpack or splat.

All load and store instructions are specified as indexed forms. A specification of a 0 in the **r**A field of the instruction results in the non-indexed form of the instruction. For all loads and stores, only the lower 32 bits of registers **r**A and **r**B are used and the effective address is 32 bits.

Power ISA embedded category load instructions are implemented such that the upper half of all registers are left unchanged for a load.

Table 7-6 lists the SPE vector load/store instructions discussed in this section.

Table 7-6. SPE Vector Load/Store Instructions

Vector Load Double into Double **(evldd)**	Vector Load Double into Half Words **(evldh)**
Vector Load Double into Double Indexed **(evlddx)**	Vector Load Double into Half Words Indexed **(evldhx)**
Vector Load Double into Words **(evldw)**	Vector Load Word into Half Words Even **(evlwhe)**
Vector Load Double into Words Indexed **(evldwx)**	Vector Load Word into Half Words Even Indexed **(evlwhex)**

Table 7-6. SPE Vector Load/Store Instructions (Continued)

Vector Load Word into Half Words Odd Unsigned (Zero-Extended) **(evlwhou)**	Vector Store Double of Double **(evstdd)**
Vector Load Word into Half Words Odd Unsigned Indexed (Zero-Extended) **(evlwhoux)**	Vector Store Double of Double Indexed **(evstddx)**
Vector Load Word into Half Words Odd Signed (With Sign Extension) **(evlwhos)**	Vector Store Double of Two Words **(evstdw)**
Vector Load Word into Half Words Odd Signed Indexed (With Sign Extension) **(evlwhosx)**	Vector Store Double of Two Words Indexed **(evstdwx)**
Vector Load Word into Word and Splat **(evlwhosx)**	Vector Store Double of Four Half Words **(evstdh)**
Vector Load Word into Word and Splat Indexed **(evlwwsplatx)**	Vector Store Double of Four Half Words Indexed **(evstdhx)**
Vector Load Word into Half Words and Splat **(evlwhsplat)**	Vector Store Word of Word from Even **(evstwwe)**
Vector Load Word into Half Words and Splat Indexed **(evlwhsplatx)**	Vector Store Word of Word from Even Indexed **(evstwwex)**
Vector Load Half Word into Half Word Even and Splat **(evlhhesplat)**	Vector Store Word of Word from Odd **(evstwwo)**
Vector Load Half Word into Half Word Even and Splat Indexed **(evlhhesplatx)**	Vector Store Word of Word from Odd Indexed **(evstwwox)**
Vector Load Half Word into Half Word Odd Unsigned and Splat (**evlhhesplatx**)	Vector Store Word of Two Half Words from Even **(evstwhe)**
Vector Load Half Word into Half Word Odd Unsigned and Splat Indexed **(evlhhousplatx)**	Vector Store Word of Two Half Words from Even Indexed **(evstwhex)**
Vector Load Half Word into Half Word Odd Signed and Splat **(evlhhossplat)**	Vector Store Word of Two Half Words from Odd **(evstwho)**
Vector Load Half Word into Half Word Odd Signed and Splat Indexed **(evlhhossplatx)**	Vector Store Word of Two Half Words from Odd Indexed **(evstwhox)**

evldd

Vector Load Double into Double

evldd **r**D,**d**(**r**A)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	UIMM[1]	0 1 1 0 0 0 0 0 0 0 1

[1] **d** = UIMM<<3

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*8)
RD = MEM(EA,8)
```

Figure 7-104 shows how bytes are loaded into **r**D as determined by the endian mode.

Byte addr	0	1	2	3	4	5	6	7
Memory	a	b	c	d	e	f	g	h
GPR in big endian	a	b	c	d	e	f	g	h
GPR in little endian	h	g	f	e	d	c	b	a

Figure 7-104. evldd Results in Big- and Little-Endian Modes

NOTE

If the EA of **evldd** is not word aligned, an Alignment interrupt is generated.

evlddx evlddx

Vector Load Double into Double Indexed

evlddx **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	011 0000 0000

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
RD = MEM(EA,8)
```

Figure 7-105 shows how bytes are loaded into **r**D as determined by the endian mode.

Byte addr	0	1	2	3	4	5	6	7
Memory	a	b	c	d	e	f	g	h
GPR in big endian	a	b	c	d	e	f	g	h
GPR in little endian	h	g	f	e	d	c	b	a

Figure 7-105. evlddx Results in Big- and Little-Endian Modes

NOTE

If the EA of **evlddx** is not word aligned, an Alignment interrupt is generated.

evldw evldw

Vector Load Double into Words

evldw **r**D,**d**(**r**A)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	UIMM[1]	0 1 1 0 0 0 0 0 0 1 1

[1] **d** = UIMM<<3

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*8)
RD$_{0:31}$ = MEM(EA,4)
RD$_{32:63}$ = MEM(EA+4,4)
```

Figure 7-106 shows how bytes are loaded into **r**D as determined by the endian mode.

Byte addr	0	1	2	3	4	5	6	7
Memory	a	b	c	d	e	f	g	h
GPR in big endian	a	b	c	d	e	f	g	h
GPR in little endian	d	c	b	a	h	g	f	e

Figure 7-106. evldw Results in Big- and Little-Endian Modes

NOTE

If the EA of **evldw** is not word aligned, an Alignment interrupt is generated.

evldwx evldwx

Vector Load Double into Words Indexed

evldwx **r**D,**r**A,**r**B

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	011 0000 0010

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
$RD_{0:31}$ = MEM(EA,4)
$RD_{32:63}$ = MEM(EA+4,4)
```

Figure 7-107 shows how bytes are loaded into **r**D as determined by the endian mode.

Byte addr	0	1	2	3	4	5	6	7
Memory	a	b	c	d	e	f	g	h
GPR in big endian	a	b	c	d	e	f	g	h
GPR in little endian	d	c	b	a	h	g	f	e

Figure 7-107. evldwx Results in Big- and Little-Endian Modes

NOTE

If the EA of **evldwx** is not word aligned, an Alignment interrupt is generated.

evldh evldh

Vector Load Double into Half Words

evldh **r**D,**d**(**r**A)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	UIMM[1]	011 0000 0101

[1] **d** = UIMM<<3

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*8)
$RD_{0:15}$ = MEM(EA,2)
$RD_{16:31}$ = MEM(EA+2,2)
$RD_{32:47}$ = MEM(EA+4,2)
$RD_{48:63}$ = MEM(EA+6,2)
```

Figure 7-108 shows how bytes are loaded into **r**D as determined by the endian mode.

Byte addr	0	1	2	3	4	5	6	7
Memory	a	b	c	d	e	f	g	h
GPR in big endian	a	b	c	d	e	f	g	h
GPR in little endian	b	a	d	c	f	e	h	g

Figure 7-108. evldh Results in Big- and Little-Endian Modes

NOTE

If the EA of **evldh** is not word aligned, an Alignment interrupt is generated.

evldhx evldhx

Vector Load Double into Half Words Indexed

evldhx **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	011 0000 0100

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
$RD_{0:15}$ = MEM(EA,2)
$RD_{16:31}$ = MEM(EA+2,2)
$RD_{32:47}$ = MEM(EA+4,2)
$RD_{48:63}$ = MEM(EA+6,2)
```

Figure 7-109 shows how bytes are loaded into **r**D as determined by the endian mode.

Byte addr	0	1	2	3	4	5	6	7
Memory	a	b	c	d	e	f	g	h
GPR in big endian	a	b	c	d	e	f	g	h
GPR in little endian	b	a	d	c	f	e	h	g

Figure 7-109. evldhx Results in Big- and Little-Endian Modes

NOTE

If the EA of **evldhx** is not word aligned, an Alignment interrupt is generated.

evlwhe evlwhe

Vector Load Word into Half Words Even

evlwhe **r**D,**d**(**r**A)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	UIMM[1]	0 1 1 0 0 0 1 0 0 0 1

[1] **d** = UIMM<<2

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*4)
RD$_{0:15}$ = MEM(EA,2)
RD$_{16:31}$ = 0x0000
RD$_{32:47}$ = MEM(EA+2,2)
RD$_{48:63}$ = 0x0000
```

Figure 7-110 shows how bytes are loaded into **r**D as determined by the endian mode.

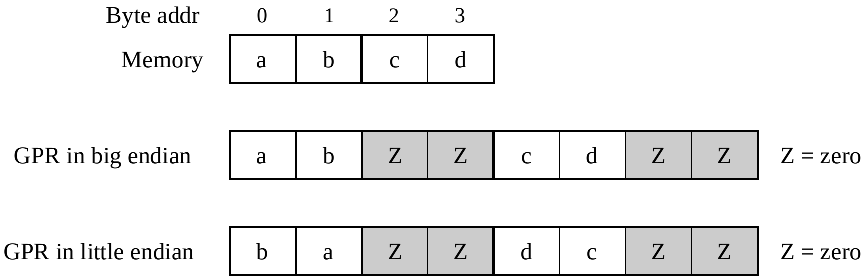


Figure 7-110. evlwhe Results in Big- and Little-Endian Modes

evlwhex

evlwhex

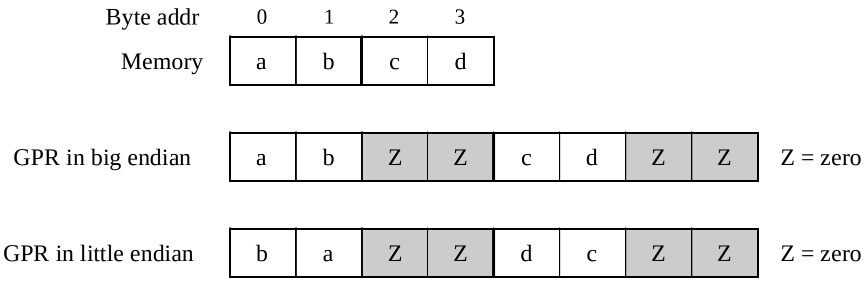
Vector Load Word into Half Words Even Indexed

evlwhex **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	011 0001 0000

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
$RD_{0:15}$ = MEM(EA,2)
$RD_{16:31}$ = 0x0000
$RD_{32:47}$ = MEM(EA+2,2)
$RD_{48:63}$ = 0x0000
```

Figure 7-111 shows how bytes are loaded into **r**D as determined by the endian mode.



Byte addr	0	1	2	3					
Memory	a	b	c	d					
GPR in big endian	a	b	Z	Z	c	d	Z	Z	Z = zero
GPR in little endian	b	a	Z	Z	d	c	Z	Z	Z = zero

Figure 7-111. evlwhex Results in Big- and Little-Endian Modes

evlwhou evlwhou

Vector Load Word into Half Words Odd Unsigned (Zero-Extended)

evlwhou **r**D,**d**(**r**A)

0 … 5	6 … 10	11 … 15	16 … 20	21 … 31
4	RD	RA	UIMM[1]	0 1 1 0 0 0 1 0 1 0 1

[1] **d** = UIMM<<2

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*4)
$RD_{0:15}$ = 0x0000
$RD_{16:31}$ = MEM(EA,2)
$RD_{32:47}$ = 0x0000
$RD_{48:63}$ = MEM(EA+2,2)
```

Figure 7-112 shows how bytes are loaded into **r**D as determined by the endian mode.

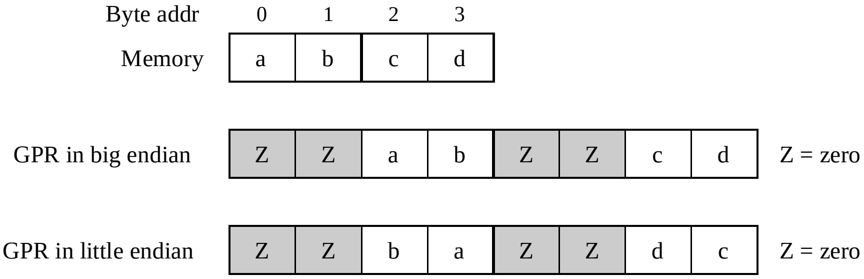


Figure 7-112. evlwhou Results in Big- and Little-Endian Modes

evlwhoux

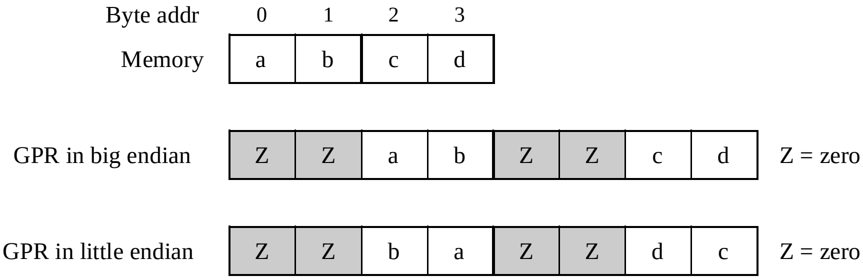
Vector Load Word into Half Words Odd Unsigned Indexed (Zero-Extended)

evlwhoux **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	011 0001 0100

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
$RD_{0:15}$ = 0x0000
$RD_{16:31}$ = MEM(EA,2)
$RD_{32:47}$ = 0x0000
$RD_{48:63}$ = MEM(EA+2,2)
```

Figure 7-113 shows how bytes are loaded into **r**D as determined by the endian mode.



Byte addr	0	1	2	3					
Memory	a	b	c	d					
GPR in big endian	Z	Z	a	b	Z	Z	c	d	Z = zero
GPR in little endian	Z	Z	b	a	Z	Z	d	c	Z = zero

Figure 7-113. evlwhoux Results in Big- and Little-Endian Modes

evlwhos

Vector Load Word into Half Words Odd Signed (With Sign Extension)

evlwhos **r**D,**d**(**r**A)

0 — 5	6 — 10	11 — 15	16 — 20	21 — 31
4	RD	RA	UIMM[1]	0 1 1 0 0 0 1 0 1 1 1

[1] **d** = UIMM<<2

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*4)
$RD_{0:31}$ = EXTS(MEM(EA,2))
$RD_{32:63}$ = EXTS(MEM(EA+2,2))
```

Figure 7-114 shows how bytes are loaded into **r**D as determined by the endian mode.

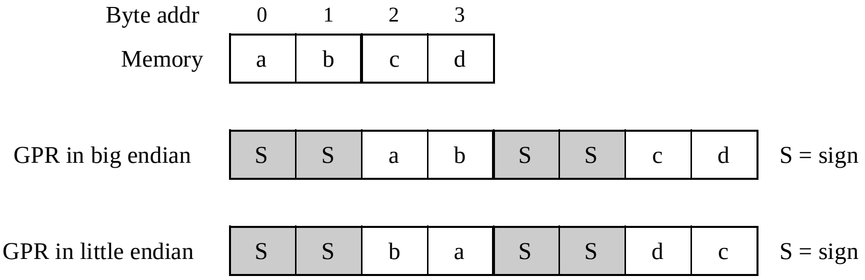


Figure 7-114. evlwhos Results in Big- and Little-Endian Modes

In the big-endian memory, the msb of a and c are sign-extended. In the little-endian memory, the msb of b and d are sign-extended.

evlwhosx evlwhosx

Vector Load Word into Half Words Odd Signed Indexed (With Sign Extension)

evlwhosx **r**D,**r**A,**r**B

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	011 0001 0110

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
$RD_{0:31}$ = EXTS(MEM(EA,2))
$RD_{32:63}$ = EXTS(MEM(EA+2,2))
```

Figure 7-115 shows how bytes are loaded into **r**D as determined by the endian mode.

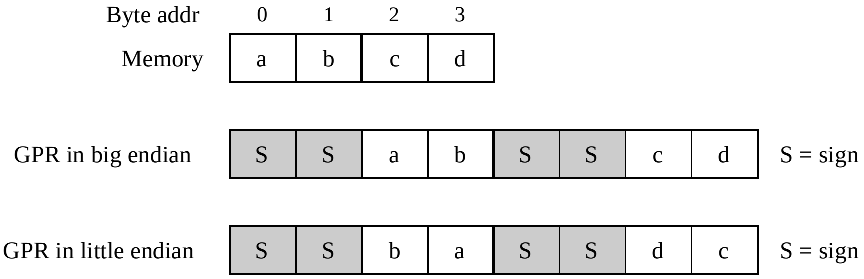


Figure 7-115. evlwhosx Results in Big- and Little-Endian Modes

In the big-endian memory, the msbs of a and c are sign-extended. In the little-endian memory, the msbs of b and d are sign-extended.

evlwwsplat evlwwsplat

Vector Load Word into Word and Splat

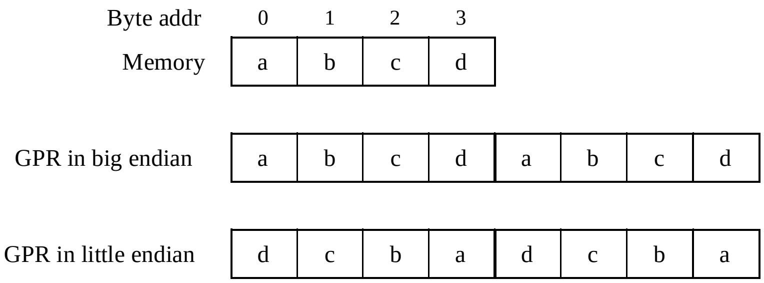
evlwwsplat **r**D,**d**(**r**A)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	UIMM[1]	011 0001 1001

[1] **d** = UIMM<<2

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*4)
$RD_{0:31}$ = MEM(EA,4)
$RD_{32:63}$ = MEM(EA,4)
```

Figure 7-116 shows how bytes are loaded into **r**D as determined by the endian mode.



Byte addr	0	1	2	3				
Memory	a	b	c	d				
GPR in big endian	a	b	c	d	a	b	c	d
GPR in little endian	d	c	b	a	d	c	b	a

Figure 7-116. evlwwsplat Results in Big- and Little-Endian Modes

evlwwsplatx evlwwsplatx

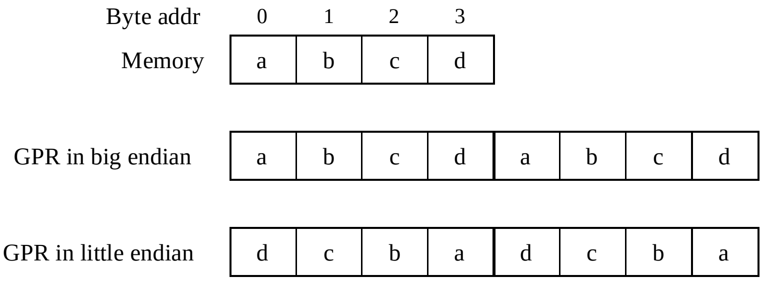
Vector Load Word into Word and Splat Indexed

evlwwsplatx **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	011 0001 1000

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
$RD_{0:31}$ = MEM(EA,4)
$RD_{32:63}$ = MEM(EA,4)
```

Figure 7-117 shows how bytes are loaded into **r**D as determined by the endian mode.



Byte addr	0	1	2	3				
Memory	a	b	c	d				
GPR in big endian	a	b	c	d	a	b	c	d
GPR in little endian	d	c	b	a	d	c	b	a

Figure 7-117. evlwwsplatx Results in Big- and Little-Endian Modes

evlwhsplat evlwhsplat

Vector Load Word into Half Words and Splat

evlwhsplat **r**D,**d**(**r**A)

0 … 5	6 … 10	11 … 15	16 … 20	21 … 31
4	RD	RA	UIMM[1]	0 1 1 0 0 0 1 1 1 0 1

[1] **d** = UIMM<<2

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*4)
$RD_{0:15}$ = MEM(EA,2)
$RD_{16:31}$ = MEM(EA,2)
$RD_{32:47}$ = MEM(EA+2,2)
$RD_{48:63}$ = MEM(EA+2,2)
```

Figure 7-118 shows how bytes are loaded into **r**D as determined by the endian mode.

Byte addr	0	1	2	3
Memory	a	b	c	d

GPR in big endian	a	b	a	b	c	d	c	d
GPR in little endian	b	a	b	a	d	c	d	c

Figure 7-118. evlwhsplat Results in Big- and Little-Endian Modes

evlwhsplatx evlwhsplatx

Vector Load Word into Half Words and Splat Indexed

evlwhsplatx **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	011 0001 1100

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
$RD_{0:15}$ = MEM(EA,2)
$RD_{16:31}$ = MEM(EA,2)
$RD_{32:47}$ = MEM(EA+2,2)
$RD_{48:63}$ = MEM(EA+2,2)
```

Figure 7-119 shows how bytes are loaded into **r**D as determined by the endian mode.

Byte addr	0	1	2	3				
Memory	a	b	c	d				
GPR in big endian	a	b	a	b	c	d	c	d
GPR in little endian	b	a	b	a	d	c	d	c

Figure 7-119. evlwhsplatx Results in Big- and Little-Endian Modes

evlhhesplat

evlhhesplat

Vector Load Half Word into Half Word Even and Splat

evlhhesplat **r**D,**d**(**r**A)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	UIMM[1]	0 1 1 0 0 0 0 1 0 0 1

[1] **d** = UIMM<<1

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*2)
RD$_{0:15}$ = MEM(EA,2)
RD$_{16:31}$ = 0x0000
RD$_{32:47}$ = MEM(EA,2)
RD$_{48:63}$ = 0x0000
```

Figure 7-120 shows how bytes are loaded into **r**D as determined by the endian mode.

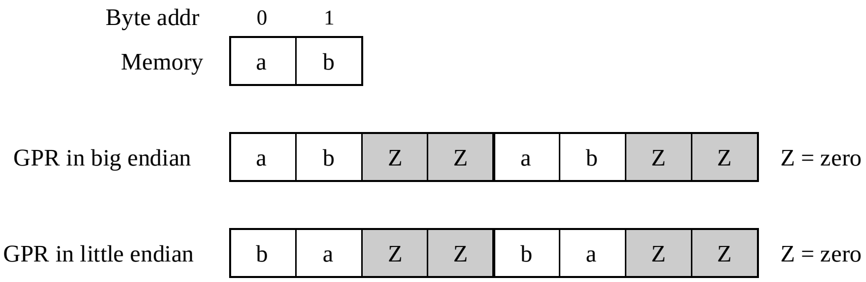


Figure 7-120. evlhhesplat Results in Big- and Little-Endian Modes

evlhhesplatx evlhhesplatx

Vector Load Half Word into Half Word Even and Splat Indexed

evlhhesplatx **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	011 0000 1000

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
$RD_{0:15}$ = MEM(EA,2)
$RD_{16:31}$ = 0x0000
$RD_{32:47}$ = MEM(EA,2)
$RD_{48:63}$ = 0x0000
```

Figure 7-121 shows how bytes are loaded into **r**D as determined by the endian mode.

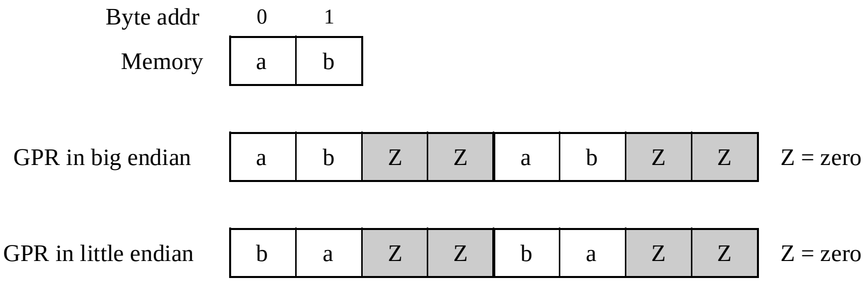


Figure 7-121. evlhhesplatx Results in Big- and Little-Endian Modes

evlhhousplat evlhhousplat

Vector Load Half Word into Half Word Odd Unsigned and Splat

evlhhousplat **r**D,**d**(**r**A)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	UIMM[1]	011 0000 1101

[1] **d** = UIMM<<1

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*2)
$RD_{0:15}$ = 0x0000
$RD_{16:31}$ = MEM(EA,2)
$RD_{32:47}$ = 0x0000
$RD_{48:63}$ = MEM(EA,2)
```

Figure 7-122 shows how bytes are loaded into **r**D as determined by the endian mode.

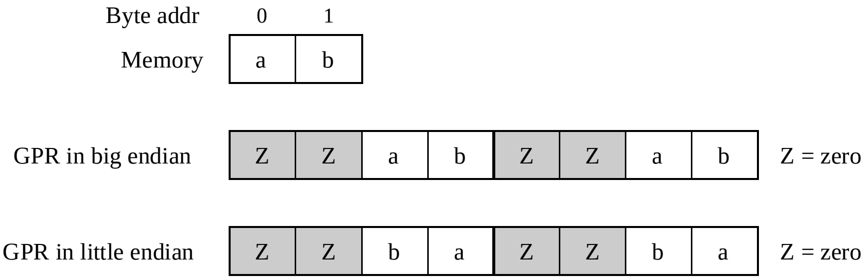


Figure 7-122. evlhhousplat Results in Big- and Little-Endian Modes

evlhhousplatx evlhhousplatx

Vector Load Half Word into Half Word Odd Unsigned and Splat Indexed

evlhhousplatx **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	011 0000 1100

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
$RD_{0:15}$ = 0x0000
$RD_{16:31}$ = MEM(EA,2)
$RD_{32:47}$ = 0x0000
$RD_{48:63}$ = MEM(EA,2)
```

Figure 7-123 shows how bytes are loaded into **r**D as determined by the endian mode.

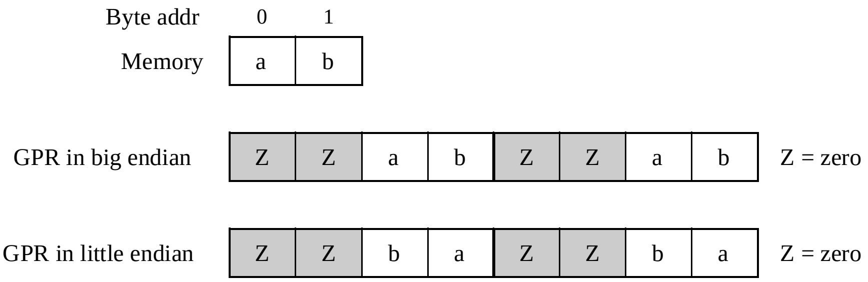


Figure 7-123. evlhhousplatx Results in Big- and Little-Endian Modes

evlhhossplat evlhhossplat

Vector Load Half Word into Half Word Odd Signed and Splat

evlhhossplat **r**D,**d**(**r**A)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	UIMM[1]	011 0000 1111

[1] **d** = UIMM<<1

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*2)
$RD_{0:31}$ = EXTS(MEM(EA,2))
$RD_{32:63}$ = EXTS(MEM(EA,2))
```

Figure 7-124 shows how bytes are loaded into **r**D as determined by the endian mode.

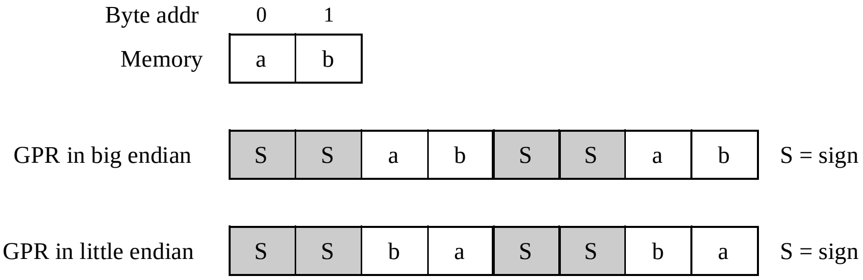


Figure 7-124. evlhhossplat Results in Big- and Little-Endian Modes

In big-endian memory, the msb of a is sign-extended. In the little-endian memory, the msb of b is sign-extended.

evlhhossplatx evlhhossplatx

Vector Load Half Word into Half Word Odd Signed and Splat Indexed

evlhhossplatx **r**D,**r**A,**r**B

0 … 5	6 … 10	11 … 15	16 … 20	21 … 31
4	RD	RA	RB	011 0000 1110

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
$RD_{0:31}$ = EXTS(MEM(EA,2))
$RD_{32:63}$ = EXTS(MEM(EA,2))
```

Figure 7-125 shows how bytes are loaded into **r**D as determined by the endian mode.

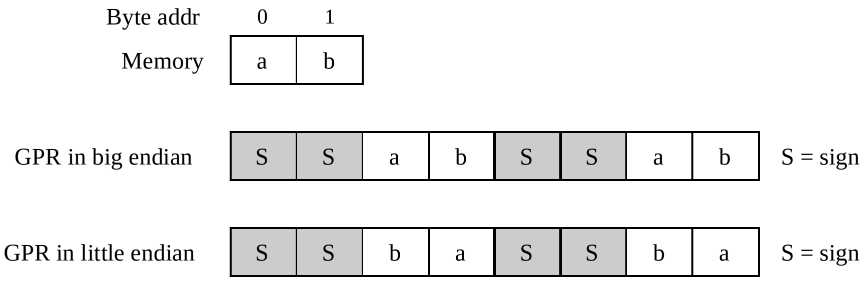


Figure 7-125. evlhhossplatx Results in Big- and Little-Endian Modes

In big-endian memory, the msb of a is sign-extended. In the little-endian memory, the msb of b is sign-extended.

evstdd evstdd

Vector Store Double of Double

evstdd **r**S,**d**(**r**A)

0 5	6 10	11 15	16 20	21 31
4	RS	RA	UIMM[1]	011 0010 0001

[1] **d** = UIMM<<3

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*8)
MEM(EA,8) = $RS_{0:63}$
```

Figure 7-126 shows how bytes are stored in memory as determined by the endian mode.

GPR	a	b	c	d	e	f	g	h
Byte addr	0	1	2	3	4	5	6	7
Memory in big endian	a	b	c	d	e	f	g	h
Memory in little endian	h	g	f	e	d	c	b	a

Figure 7-126. evstdd Results in Big- and Little-Endian Modes

NOTE

If the EA of **evstdd** is not word aligned, an Alignment interrupt is generated.

evstddx evstddx

Vector Store Double of Double Indexed

evstddx **r**S,**r**A,**r**B

0 – 5	6 – 10	11 – 15	16 – 20	21 – 31
4	RS	RA	RB	011 0010 0000

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
MEM(EA,8) = RS$_{0:63}$
```

Figure 7-127 shows how bytes are stored in memory as determined by the endian mode.

GPR	a	b	c	d	e	f	g	h
Byte addr	0	1	2	3	4	5	6	7
Memory in big endian	a	b	c	d	e	f	g	h
Memory in little endian	h	g	f	e	d	c	b	a

Figure 7-127. evstddx Results in Big- and Little-Endian Modes

NOTE

If the EA of **evstddx** is not word aligned, an Alignment interrupt is generated.

evstdw

Vector Store Double of Two Words

evstdw **r**S,**d**(**r**A)

0 5	6 10	11 15	16 20	21 31
4	RS	RA	UIMM[1]	0 1 1 0 0 1 0 0 0 1 1

[1] **d** = UIMM<<3

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*8)
MEM(EA,4) = $RS_{0:31}$
MEM(EA+4,4) = $RS_{32:63}$
```

Figure 7-128 shows how bytes are stored in memory as determined by the endian mode.

GPR	a	b	c	d	e	f	g	h
Byte addr	0	1	2	3	4	5	6	7
Memory in big endian	a	b	c	d	e	f	g	h
Memory in little endian	d	c	b	a	h	g	f	e

Figure 7-128. evstdw Results in Big- and Little-Endian Modes

NOTE

If the EA of **evstdw** is not word aligned, an Alignment interrupt is generated.

evstdwx evstdwx

Vector Store Double of Two Words Indexed

evstdwx **r**S,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RS	RA	RB	011 0010 0010

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
MEM(EA,4) = $RS_{0:31}$
MEM(EA+4,4) = $RS_{32:63}$
```

Figure 7-129 shows how bytes are stored in memory as determined by the endian mode.

GPR	a	b	c	d	e	f	g	h
Byte addr	0	1	2	3	4	5	6	7
Memory in big endian	a	b	c	d	e	f	g	h
Memory in little endian	d	c	b	a	h	g	f	e

Figure 7-129. evstdwx Results in Big- and Little-Endian Modes

NOTE

If the EA of **evstdwx** is not word aligned, an Alignment interrupt is generated.

evstdh evstdh

Vector Store Double of Four Half Words

evstdh **r**S,**d**(**r**A)

0 5	6 10	11 15	16 20	21 31
4	RS	RA	UIMM[1]	0 1 1 0 0 1 0 0 1 0 1

[1] **d** = UIMM<<3

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*8)
MEM(EA,2) = RS$_{0:15}$
MEM(EA+2,2) = RS$_{16:31}$
MEM(EA+4,2) = RS$_{32:47}$
MEM(EA+6,2) = RS$_{48:63}$
```

Figure 7-130 shows how bytes are stored in memory as determined by the endian mode.

GPR	a	b	c	d	e	f	g	h
Byte addr	0	1	2	3	4	5	6	7
Memory in big endian	a	b	c	d	e	f	g	h
Memory in little endian	b	a	d	c	f	e	h	g

Figure 7-130. evstdh Results in Big- and Little-Endian Modes

NOTE

If the EA of **evstdh** is not word aligned, an Alignment interrupt is generated.

evstdhx

evstdhx

Vector Store Double of Four Half Words Indexed

evstdhx rS,rA,rB

0 5	6 10	11 15	16 20	21 31
4	RS	RA	RB	011 0010 0100

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
MEM(EA,2) = $RS_{0:15}$
MEM(EA+2,2) = $RS_{16:31}$
MEM(EA+4,2) = $RS_{32:47}$
MEM(EA+6,2) = $RS_{48:63}$
```

Figure 7-131 shows how bytes are stored in memory as determined by the endian mode.

GPR	a	b	c	d	e	f	g	h
Byte addr	0	1	2	3	4	5	6	7
Memory in big endian	a	b	c	d	e	f	g	h
Memory in little endian	b	a	d	c	f	e	h	g

Figure 7-131. evstdhx Results in Big- and Little-Endian Modes

NOTE

If the EA of **evstdhx** is not word aligned, an Alignment interrupt is generated.

evstwwe evstwwe

Vector Store Word of Word from Even

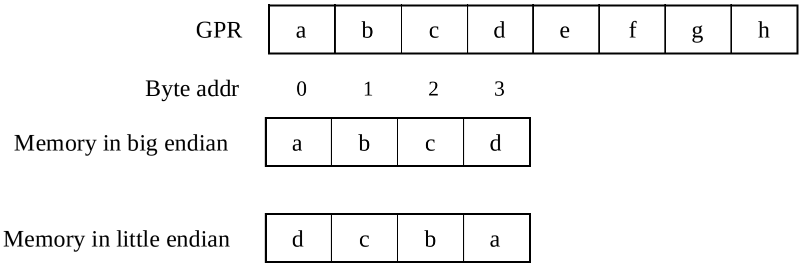
evstwwe **rS,d(rA)**

0–5	6–10	11–15	16–20	21–31
4	RS	RA	UIMM[1]	011 0011 1001

[1] **d** = UIMM<<2

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*4)
MEM(EA,4) = $RS_{0:31}$
```

Figure 7-132 shows how bytes are stored in memory as determined by the endian mode.



GPR	a	b	c	d	e	f	g	h
Byte addr	0	1	2	3				
Memory in big endian	a	b	c	d				
Memory in little endian	d	c	b	a				

Figure 7-132. evstwwe Results in Big- and Little-Endian Modes

evstwwex

Vector Store Word of Word from Even Indexed

evstwwex **r**S,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RS	RA	RB	011 0011 1000

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
MEM(EA,4) = $RS_{0:31}$
```

Figure 7-133 shows how bytes are stored in memory as determined by the endian mode.

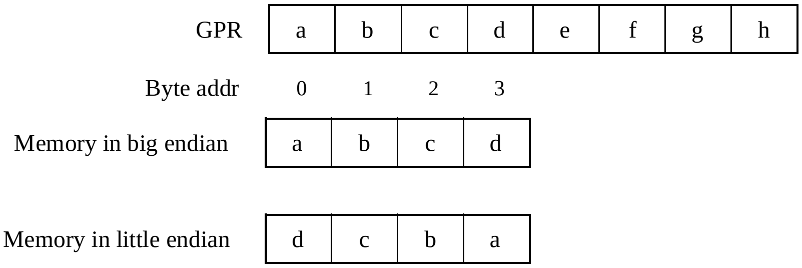


Figure 7-133. evstwwex Results in Big- and Little-Endian Modes

evstwwo evstwwo

Vector Store Word of Word from Odd

evstwwo **rS,d(rA)**

0 5	6 10	11 15	16 20	21 31
4	RS	RA	UIMM[1]	0 1 1 0 0 1 1 1 1 0 1

[1] **d** = UIMM<<2

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*4)
MEM(EA,4) = $rS_{32:63}$
```

Figure 7-134 shows how bytes are stored in memory as determined by the endian mode.

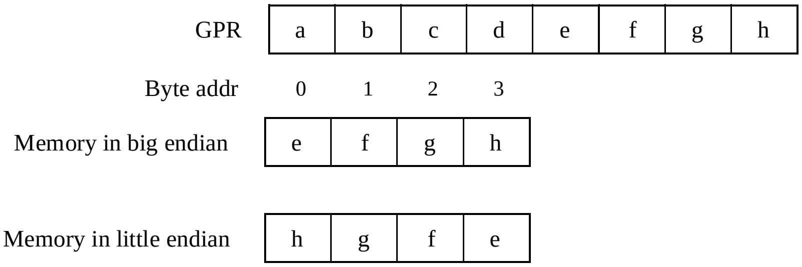


Figure 7-134. evstwwo Results in Big- and Little-Endian Modes

evstwwox evstwwox

Vector Store Word of Word from Odd Indexed

evstwwox **r**S,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RS	RA	RB	011 0011 1100

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
MEM(EA,4) = $rS_{32:63}$
```

Figure 7-135 shows how bytes are stored in memory as determined by the endian mode.

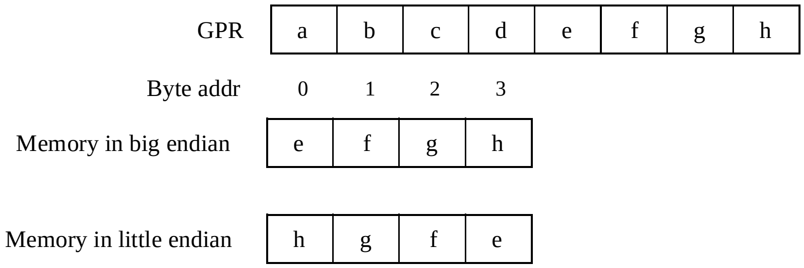


Figure 7-135. evstwwox Results in Big- and Little-Endian Modes

evstwhe   evstwhe

Vector Store Word of Two Half Words from Even

evstwhe **r**S,**d**(**r**A)

0 5	6 10	11 15	16 20	21 31
4	RS	RA	UIMM[1]	011 0011 0001

[1] **d** = UIMM<<2

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*4)
MEM(EA,2) = RS$_{0:15}$
MEM(EA+2,2) = RS$_{32:47}$
```

Figure 7-136 shows how bytes are stored in memory as determined by the endian mode.

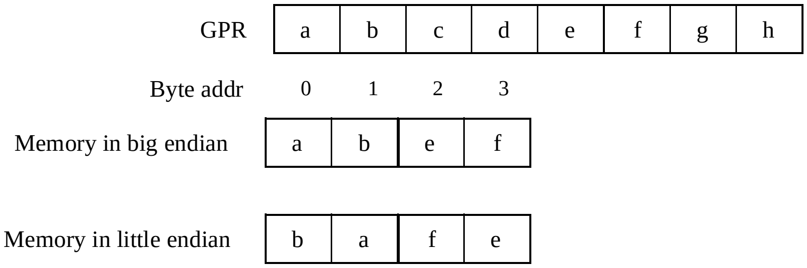


Figure 7-136. evstwhe Results in Big- and Little-Endian Modes

evstwhex evstwhex

Vector Store Word of Two Half Words from Even Indexed

evstwhex **r**S,**r**A,**r**B

0–5	6–10	11–15	16–20	21–31
4	RS	RA	RB	011 0011 0000

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
MEM(EA,2) = $RS_{0:15}$
MEM(EA+2,2) = $RS_{32:47}$
```

Figure 7-137 shows how bytes are stored in memory as determined by the endian mode.

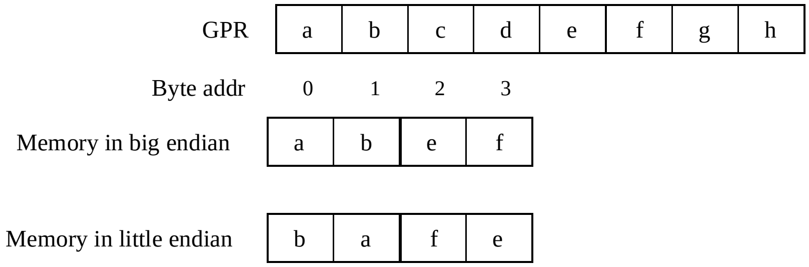


Figure 7-137. evstwhex Results in Big- and Little-Endian Modes

evstwho evstwho

Vector Store Word of Two Half Words from Odd

evstwho **r**S,**d**(**r**A)

0 5	6 10	11 15	16 20	21 31
4	RS	RA	UIMM[1]	011 0011 0101

[1] **d** = UIMM<<2

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*4)
MEM(EA,2) = $RS_{16:31}$
MEM(EA+2,2) = $RS_{48:63}$
```

Figure 7-138 shows how bytes are stored in memory as determined by the endian mode.

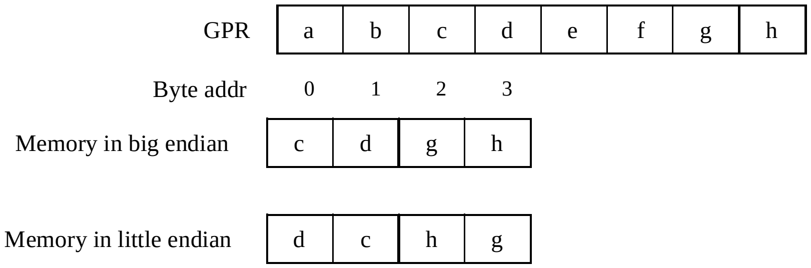


Figure 7-138. evstwho Results in Big- and Little-Endian Modes

evstwhox evstwhox

Vector Store Word of Two Half Words from Odd Indexed

evstwhox **r**S,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RS	RA	RB	011 0011 0100

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
MEM(EA,2) = $RS_{16:31}$
MEM(EA+2,2) = $RS_{48:63}$
```

Figure 7-139 shows how bytes are stored in memory as determined by the endian mode.

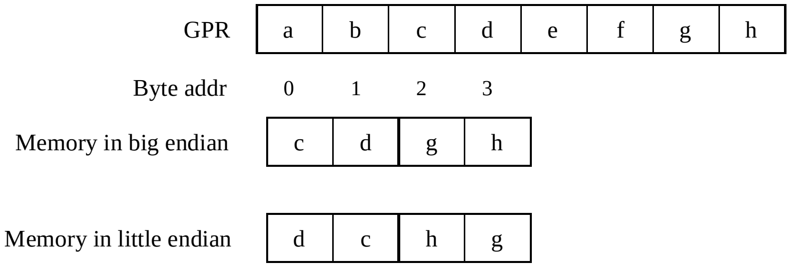


Figure 7-139. evstwhox Results in Big- and Little-Endian Modes

7.6 SPE Instruction Timing

Table 7-7, Table 7-8, and Table 7-9 show SPE instruction timing in number of processor clock cycles. 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.

7.6.1 SPE Integer Simple Instructions Timing

Table 7-7 shows instruction timing for SPE integer simple instructions sorted by opcode. These instructions are issued as a pair of operations.

Table 7-7. Timing for Integer Simple Instructions

Instruction	Latency	Throughput	Comments
brinc	1	1	—
evabs	1	1	—
evaddiw	1	1	—
evaddw	1	1	—
evand	1	1	—

Table 7-7. Timing for Integer Simple Instructions (Continued)

Instruction	Latency	Throughput	Comments
evandc	1	1	—
evcmpeq	1	1	—
evcmpgts	1	1	—
evcmpgtu	1	1	—
evcmplts	1	1	—
evcmpltu	1	1	—
evcntlsw	1	1	—
evcntlzw	1	1	—
eveqv	1	1	—
evextsb	1	1	—
evextsh	1	1	—
evmergehi	1	1	—
evmergehilo	1	1	—
evmergelo	1	1	—
evmergelohi	1	1	—
evnand	1	1	—
evneg	1	1	—
evnor	1	1	—
evor	1	1	—
evorc	1	1	—
evrlw	1	1	—
evrlwi	1	1	—
evrndw	1	1	—
evsel	1	1	—
evslw	1	1	—
evslwi	1	1	—
evsplatfi	1	1	—
evsplati	1	1	—
evsrwis	1	1	—
evsrwiu	1	1	—
evsrws	1	1	—
evsrwu	1	1	—
evsubfw	1	1	—

Table 7-7. Timing for Integer Simple Instructions (Continued)

Instruction	Latency	Throughput	Comments
evsubifw	1	1	—
evxor	1	1	—

7.6.2 SPE Load and Store Instruction Timing

Table 7-7 shows instruction timing for SPE load and store instructions sorted by opcode. Actual timing depends on alignment; the table indicates timing for aligned operands.

Table 7-8. SPE Load and Store Instruction Timing

Instruction	Latency	Throughput	Comments
evldd	2	1	—
evlddx	2	1	—
evldh	2	1	—
evldhx	2	1	—
evldw	2	1	—
evldwx	2	1	—
evlhhesplat	2	1	—
evlhhesplatx	2	1	—
evlhhossplat	2	1	—
evlhhossplatx	2	1	—
evlhhousplat	2	1	—
evlhhousplatx	2	1	—
evlwhe	2	1	—
evlwhex	2	1	—
evlwhos	2	1	—
evlwhosx	2	1	—
evlwhou	2	1	—
evlwhoux	2	1	—
evlwhsplat	2	1	—
evlwhsplatx	2	1	—
evlwwsplat	2	1	—
evlwwsplatx	2	1	—
evstdd	2	1	—
evstddx	2	1	—

Table 7-8. SPE Load and Store Instruction Timing (Continued)

Instruction	Latency	Throughput	Comments
evstdh	2	1	—
evstdhx	2	1	—
evstdw	2	1	—
evstdwx	2	1	—
evstwhe	2	1	—
evstwhex	2	1	—
evstwho	2	1	—
evstwhox	2	1	—
evstwwe	2	1	—
evstwwex	2	1	—
evstwwo	2	1	—
evstwwox	2	1	—

7.6.3 SPE Complex Integer Instruction Timing

Table 7-9 shows instruction timing for SPE complex integer instructions sorted by opcode. For the divide instructions, the number of stall cycles is (latency) for following instructions.

Table 7-9. SPE Complex Integer Instruction Timing

Instruction	Latency	Through put	Comments
evaddsmiaaw	1	1	—
evaddssiaaw	1	1	—
evaddumiaaw	1	1	—
evaddusiaaw	1	1	—
evdivws	12–32[1]	12–32[1]	—
evdivwu	12–32[1]	12–32[1]	—
evmhegsmfaa	2	1	—
evmhegsmfan	2	1	—
evmhegsmiaa	2	1	—
evmhegsmian	2	1	—
evmhegumiaa	2	1	—
evmhegumian	2	1	—
evmhesmf	2	1	—
evmhesmfa	2	1	—

Table 7-9. SPE Complex Integer Instruction Timing (Continued)

Instruction	Latency	Through put	Comments
evmhesmfaaw	2	1	—
evmhesmfanw	2	1	—
evmhesmi	2	1	—
evmhesmia	2	1	—
evmhesmiaaw	2	1	—
evmhesmianw	2	1	—
evmhessf	2	1	—
evmhessfa	2	1	—
evmhessfaaw	2	1	—
evmhessfanw	2	1	—
evmhessiaaw	2	1	—
evmhessianw	2	1	—
evmheumi	2	1	—
evmheumia	2	1	—
evmheumiaaw	2	1	—
evmheumianw	2	1	—
evmheusiaaw	2	1	—
evmheusianw	2	1	—
evmhogsmfaa	2	1	—
evmhogsmfan	2	1	—
evmhogsmiaa	2	1	—
evmhogsmian	2	1	—
evmhogumiaa	2	1	—
evmhogumian	2	1	—
evmhosmf	2	1	—
evmhosmfa	2	1	—
evmhosmfaaw	2	1	—
evmhosmfanw	2	1	—
evmhosmi	2	1	—
evmhosmia	2	1	—
evmhosmiaaw	2	1	—
evmhosmianw	2	1	—

Table 7-9. SPE Complex Integer Instruction Timing (Continued)

Instruction	Latency	Through put	Comments
evmhossf	2	1	—
evmhossfa	2	1	—
evmhossfaaw	2	1	—
evmhossfanw	2	1	—
evmhossiaaw	2	1	—
evmhossianw	2	1	—
evmhoumi	2	1	—
evmhoumia	2	1	—
evmhoumiaaw	2	1	—
evmhoumianw	2	1	—
evmhousiaaw	2	1	—
evmhousianw	2	1	—
evmra	2	1	—
evmwhsmf	2	1	—
evmwhsmfa	2	1	—
evmwhsmi	2	1	—
evmwhsmia	2	1	—
evmwhssf	2	1	—
evmwhssfa	2	1	—
evmwhumi	2	1	—
evmwhumia	2	1	—
evmwlsmiaaw	2	1	—
evmwlsmianw	2	1	—
evmwlssiaaw	2	1	—
evmwlssianw	2	1	—
evmwlumi	2	1	—
evmwlumia	2	1	—
evmwlumiaaw	2	1	—
evmwlumianw	2	1	—
evmwlusiaaw	2	1	—
evmwlusianw	2	1	—
evmwsmf	2	1	—

Table 7-9. SPE Complex Integer Instruction Timing (Continued)

Instruction	Latency	Through put	Comments
evmwsmfa	2	1	—
evmwsmfaa	2	1	—
evmwsmfan	2	1	—
evmwsmi	2	1	—
evmwsmia	2	1	—
evmwsmiaa	2	1	—
evmwsmian	2	1	—
evmwssf	2	1	—
evmwssfa	2	1	—
evmwssfaa	2	1	—
evmwssfan	2	1	—
evmwumi	2	1	—
evmwumia	2	1	—
evmwumiaa	2	1	—
evmwumian	2	1	—
evsubfsmiaaw	1	1	—
evsubfssiaaw	1	1	—
evsubfumiaaw	1	1	—
evsubfusiaaw	1	1	—

[1] Timing is data dependent

7.7 Instruction Forms and Opcodes

Table 7-10 provides the division of the opcode space for the new SPE instructions.

Table 7-10. Opcode Space Division

Opcode Bits		Instruction Class
0–5	**21–25**	
4	0100*	SPE APU integer simple instructions
4	01010	EFPU floating-point instructions
4	01011	Embedded floating-point APU instructions
4	01100	SPE APU load/store instructions
4	01101	SPE APU reserved for future use

Table 7-10. Opcode Space Division (Continued)

Opcode Bits		Instruction Class
0–5	21–25	
4	0111*	SPE APU reserved for future use
4	10***	SPE APU integer complex instructions
4	11***	SPE APU integer complex instructions: reserved for future use

7.7.1 SPE Vector Integer Simple Instructions

For instructions that have signed and unsigned forms, bit 31 is 1 for the signed form and 0 for the unsigned form. For instructions that have immediate forms, bit 30 is 1 for immediate forms. All instructions have the destination register specified in the bits 6–10, which differs from PowerPC ISA/Book E where some instructions have the destination in bits 11–15.

Table 7-11. Opcodes for Integer Simple Instructions

Instruction	Opcode					Comments
	0–5	6–10	11–15	16–20	21–31	
brinc	4	rD	RA	rB	010 0000 1111	—
evabs	4	RD	RA	00000	010 0000 1000	—
evaddiw	4	RD	UIMM	RB	010 0000 0010	—
evaddw	4	RD	RA	RB	010 0000 0000	—
evand	4	RD	RA	RB	010 0001 0001	RD = RA & RB
evandc	4	RD	RA	RB	010 0001 0010	RD = RA & (~RB)
evcmpeq	4	crfD 00	RA	RB	010 0011 0100	—
evcmpgts	4	crfD 00	RA	RB	010 0011 0001	—
evcmpgtu	4	crfD 00	RA	RB	010 0011 0000	—
evcmplts	4	crfD 00	RA	RB	010 0011 0011	—
evcmpltu	4	crfD 00	RA	RB	010 0011 0010	—
evcntlsw	4	RD	RA	00000	010 0000 1110	—
evcntlzw	4	RD	RA	00000	010 0000 1101	—
eveqv	4	RD	RA	RB	010 0001 1001	RD = ~(RA XOR RB)
evextsb	4	RD	RA	00000	010 0000 1010	—
evextsh	4	RD	RA	00000	010 0000 1011	—
evmergehi	4	RD	RA	RB	010 0010 1100	—
evmergehilo	4	RD	RA	RB	010 0010 1110	—
evmergelo	4	RD	RA	RB	010 0010 1101	—

Table 7-11. Opcodes for Integer Simple Instructions (Continued)

Instruction	Opcode					Comments
	0–5	6–10	11–15	16–20	21–31	
evmergelohi	4	RD	RA	RB	010 0010 1111	—
evnand	4	RD	RA	RB	010 0001 1110	RD = ~(RA & RB)
evneg	4	RD	RA	00000	010 0000 1001	—
evnor	4	RD	RA	RB	010 0001 1000	RD = ~(RA | RB)
evor	4	RD	RA	RB	010 0001 0111	RD = RA | RB
evorc	4	RD	RA	RB	010 0001 1011	RD = RA | (~RB)
evrlw	4	RD	RA	RB	010 0010 1000	—
evrlwi	4	RD	RA	UIMM	010 0010 1010	—
evrndw	4	RD	RA	00000	010 0000 1100	—
evsel	4	RD	RA	RB	010 0111 1crfS	crfS is a 3-bit field
evslw	4	RD	RA	RB	010 0010 0100	—
evslwi	4	RD	RA	UIMM	010 0010 0110	—
evsplatfi	4	RD	SIMM	00000	010 0010 1011	—
evsplati	4	RD	SIMM	00000	010 0010 1001	—
evsrwis	4	RD	RA	UIMM	010 0010 0011	—
evsrwiu	4	RD	RA	UIMM	010 0010 0010	—
evsrws	4	RD	RA	RB	010 0010 0001	—
evsrwu	4	RD	RA	RB	010 0010 0000	—
evsubfw	4	RD	RA	RB	010 0000 0100	—
evsubifw	4	RD	UIMM	RB	010 0000 0110	—
evxor	4	RD	RA	RB	010 0001 0110	RD = RA XOR RB

7.7.2 Opcodes for SPE Load and Store Instructions

Load instructions have a ‘0’ in bit 26 whereas all stores have a ‘1’ in bit 26. Bits 27 and 28 indicate the size of the data access to memory. Bit 31 indicates whether the index is immediate or the contents of a register. All store instructions have the source of the data register specified in bits 6–10 (RS).

Table 7-12. SPE Load and Store Instruction Opcodes

Instruction	Opcode Bits					Comments
	0–5	6–10	11–15	16–20	21–31	
evldd	4	RD	RA	UIMM	011 0000 0001	—
evlddx	4	RD	RA	RB	011 0000 0000	—

Table 7-12. SPE Load and Store Instruction Opcodes (Continued)

Instruction	Opcode Bits					Comments
	0–5	6–10	11–15	16–20	21–31	
evldh	4	RD	RA	UIMM	011 0000 0101	—
evldhx	4	RD	RA	RB	011 0000 0100	—
evldw	4	RD	RA	UIMM	011 0000 0011	—
evldwx	4	RD	RA	RB	011 0000 0010	—
evlhhesplat	4	RD	RA	UIMM	011 0000 1001	—
evlhhesplatx	4	RD	RA	RB	011 0000 1000	—
evlhhossplat	4	RD	RA	UIMM	011 0000 1111	—
evlhhossplatx	4	RD	RA	RB	011 0000 1110	—
evlhhousplat	4	RD	RA	UIMM	011 0000 1101	—
evlhhousplatx	4	RD	RA	RB	011 0000 1100	—
evlwhe	4	RD	RA	UIMM	011 0001 0001	—
evlwhex	4	RD	RA	RB	011 0001 0000	—
evlwhos	4	RD	RA	UIMM	011 0001 0111	—
evlwhosx	4	RD	RA	RB	011 0001 0110	—
evlwhou	4	RD	RA	UIMM	011 0001 0101	—
evlwhoux	4	RD	RA	RB	011 0001 0100	—
evlwhsplat	4	RD	RA	UIMM	011 0001 1101	—
evlwhsplatx	4	RD	RA	RB	011 0001 1100	—
evlwwsplat	4	RD	RA	UIMM	011 0001 1001	—
evlwwsplatx	4	RD	RA	RB	011 0001 1000	—
evstdd	4	RS	RA	UIMM	011 0010 0001	—
evstddx	4	RS	RA	RB	011 0010 0000	—
evstdh	4	RS	RA	UIMM	011 0010 0101	—
evstdhx	4	RS	RA	RB	011 0010 0100	—
evstdw	4	RS	RA	UIMM	011 0010 0011	—
evstdwx	4	RS	RA	RB	011 0010 0010	—
evstwhe	4	RS	RA	UIMM	011 0011 0001	—
evstwhex	4	RS	RA	RB	011 0011 0000	—
evstwho	4	RS	RA	UIMM	011 0011 0101	—
evstwhox	4	RS	RA	RB	011 0011 0100	—
evstwwe	4	RS	RA	UIMM	011 0011 1001	—

Table 7-12. SPE Load and Store Instruction Opcodes (Continued)

Instruction	Opcode Bits					Comments
	0–5	6–10	11–15	16–20	21–31	
evstwwex	4	RS	RA	RB	011 0011 1000	—
evstwwo	4	RS	RA	UIMM	011 0011 1101	—
evstwwox	4	RS	RA	RB	011 0011 1100	—

7.7.3 Opcodes for SPE Complex Integer Instructions

Figure 7-13 shows the opcodes for SPE complex integer instructions, sorted by mnemonic.

Table 7-13. Opcodes for Complex Integer Instructions, Sorted by Mnemonic

Instruction	Opcode Bits				
	0–5	6–10	11–15	16–20	21–31
evaddsmiaaw	4	RD	RA	00000	100 1100 1001
evaddssiaaw	4	RD	RA	00000	100 1100 0001
evaddumiaaw	4	RD	RA	00000	100 1100 1000
evaddusiaaw	4	RD	RA	00000	100 1100 0000
evdivws	4	RD	RA	RB	100 1100 0110
evdivwu	4	RD	RA	RB	100 1100 0111
evmhegsmfaa	4	RD	RA	RB	101 0010 1011
evmhegsmfan	4	RD	RA	RB	101 1010 1011
evmhegsmiaa	4	RD	RA	RB	101 0010 1001
evmhegsmian	4	RD	RA	RB	101 1010 1001
evmhegumiaa	4	RD	RA	RB	101 0010 1000
evmhegumian	4	RD	RA	RB	101 1010 1000
evmhesmf	4	RD	RA	RB	100 0000 1011
evmhesmfa	4	RD	RA	RB	100 0010 1011
evmhesmfaaw	4	RD	RA	RB	101 0000 1011
evmhesmfanw	4	RD	RA	RB	101 1000 1011
evmhesmi	4	RD	RA	RB	100 0000 1001
evmhesmia	4	RD	RA	RB	100 0010 1001
evmhesmiaaw	4	RD	RA	RB	101 0000 1001
evmhesmianw	4	RD	RA	RB	101 1000 1001
evmhessf	4	RD	RA	RB	100 0000 0011
evmhessfa	4	RD	RA	RB	100 0010 0011

Table 7-13. Opcodes for Complex Integer Instructions, Sorted by Mnemonic (Continued)

Instruction	Opcode Bits				
	0–5	6–10	11–15	16–20	21–31
evmhessfaaw	4	RD	RA	RB	101 0000 0011
evmhessfanw	4	RD	RA	RB	101 1000 0011
evmhessiaaw	4	RD	RA	RB	101 0000 0001
evmhessianw	4	RD	RA	RB	101 1000 0001
evmheumi	4	RD	RA	RB	100 0000 1000
evmheumia	4	RD	RA	RB	100 0010 1000
evmheumiaaw	4	RD	RA	RB	101 0000 1000
evmheumianw	4	RD	RA	RB	101 1000 1000
evmheusiaaw	4	RD	RA	RB	101 0000 0000
evmheusianw	4	RD	RA	RB	101 1000 0000
evmhogsmfaa	4	RD	RA	RB	101 0010 1111
evmhogsmfan	4	RD	RA	RB	101 1010 1111
evmhogsmiaa	4	RD	RA	RB	101 0010 1101
evmhogsmian	4	RD	RA	RB	101 1010 1101
evmhogumiaa	4	RD	RA	RB	101 0010 1100
evmhogumian	4	RD	RA	RB	101 1010 1100
evmhosmf	4	RD	RA	RB	100 0000 1111
evmhosmfa	4	RD	RA	RB	100 0010 1111
evmhosmfaaw	4	RD	RA	RB	101 0000 1111
evmhosmfanw	4	RD	RA	RB	101 1000 1111
evmhosmi	4	RD	RA	RB	100 0000 1101
evmhosmia	4	RD	RA	RB	100 0010 1101
evmhosmiaaw	4	RD	RA	RB	101 0000 1101
evmhosmianw	4	RD	RA	RB	101 1000 1101
evmhossf	4	RD	RA	RB	100 0000 0111
evmhossfa	4	RD	RA	RB	100 0010 0111
evmhossfaaw	4	RD	RA	RB	101 0000 0111
evmhossfanw	4	RD	RA	RB	101 1000 0111
evmhossiaaw	4	RD	RA	RB	101 0000 0101
evmhossianw	4	RD	RA	RB	101 1000 0101
evmhoumi	4	RD	RA	RB	100 0000 1100

Table 7-13. Opcodes for Complex Integer Instructions, Sorted by Mnemonic (Continued)

Instruction	Opcode Bits				
	0–5	6–10	11–15	16–20	21–31
evmhoumia	4	RD	RA	RB	100 0010 1100
evmhoumiaaw	4	RD	RA	RB	101 0000 1100
evmhoumianw	4	RD	RA	RB	101 1000 1100
evmhousiaaw	4	RD	RA	RB	101 0000 0100
evmhousianw	4	RD	RA	RB	101 1000 0100
evmra	4	RD	RA	00000	100 1100 0100
evmwhsmf	4	RD	RA	RB	100 0100 1111
evmwhsmfa	4	RD	RA	RB	100 0110 1111
evmwhsmi	4	RD	RA	RB	100 0100 1101
evmwhsmia	4	RD	RA	RB	100 0110 1101
evmwhssf	4	RD	RA	RB	100 0100 0111
evmwhssfa	4	RD	RA	RB	100 0110 0111
evmwhumi	4	RD	RA	RB	100 0100 1100
evmwhumia	4	RD	RA	RB	100 0110 1100
evmwlsmiaaw	4	RD	RA	RB	101 0100 1001
evmwlsmianw	4	RD	RA	RB	101 1100 1001
evmwlssiaaw	4	RD	RA	RB	101 0100 0001
evmwlssianw	4	RD	RA	RB	101 1100 0001
evmwlumi	4	RD	RA	RB	100 0100 1000
evmwlumia	4	RD	RA	RB	100 0110 1000
evmwlumiaaw	4	RD	RA	RB	101 0100 1000
evmwlumianw	4	RD	RA	RB	101 1100 1000
evmwlusiaaw	4	RD	RA	RB	101 0100 0000
evmwlusianw	4	RD	RA	RB	101 1100 0000
evmwsmf	4	RD	RA	RB	100 0101 1011
evmwsmfa	4	RD	RA	RB	100 0111 1011
evmwsmfaa	4	RD	RA	RB	101 0101 1011
evmwsmfan	4	RD	RA	RB	101 1101 1011
evmwsmi	4	RD	RA	RB	100 0101 1001
evmwsmia	4	RD	RA	RB	100 0111 1001
evmwsmiaa	4	RD	RA	RB	101 0101 1001

Table 7-13. Opcodes for Complex Integer Instructions, Sorted by Mnemonic (Continued)

Instruction	Opcode Bits				
	0–5	6–10	11–15	16–20	21–31
evmwsmian	4	RD	RA	RB	101 1101 1001
evmwssf	4	RD	RA	RB	100 0101 0011
evmwssfa	4	RD	RA	RB	100 0111 0011
evmwssfaa	4	RD	RA	RB	101 0101 0011
evmwssfan	4	RD	RA	RB	101 1101 0011
evmwumi	4	RD	RA	RB	100 0101 1000
evmwumia	4	RD	RA	RB	100 0111 1000
evmwumiaa	4	RD	RA	RB	101 0101 1000
evmwumian	4	RD	RA	RB	101 1101 1000
evsubfsmiaaw	4	RD	RA	00000	100 1100 1011
evsubfssiaaw	4	RD	RA	00000	100 1100 0011
evsubfumiaaw	4	RD	RA	00000	100 1100 1010
evsubfusiaaw	4	RD	RA	00000	100 1100 0010

Figure 7-14 shows the opcodes for SPE complex integer instructions, sorted by opcode.

Table 7-14. Opcodes for Complex Integer Instructions, Sorted by Opcode

Instruction	Opcode Bits				
	0–5	6–10	11–15	16–20	21–31
evmhessf	4	RD	RA	RB	100 0000 0011
evmhossf	4	RD	RA	RB	100 0000 0111
evmheumi	4	RD	RA	RB	100 0000 1000
evmhesmi	4	RD	RA	RB	100 0000 1001
evmhesmf	4	RD	RA	RB	100 0000 1011
evmhoumi	4	RD	RA	RB	100 0000 1100
evmhosmi	4	RD	RA	RB	100 0000 1101
evmhosmf	4	RD	RA	RB	100 0000 1111
evmhessfa	4	RD	RA	RB	100 0010 0011
evmhossfa	4	RD	RA	RB	100 0010 0111
evmheumia	4	RD	RA	RB	100 0010 1000
evmhesmia	4	RD	RA	RB	100 0010 1001
evmhesmfa	4	RD	RA	RB	100 0010 1011

Table 7-14. Opcodes for Complex Integer Instructions, Sorted by Opcode (Continued)

Instruction	Opcode Bits				
	0–5	6–10	11–15	16–20	21–31
evmhoumia	4	RD	RA	RB	100 0010 1100
evmhosmia	4	RD	RA	RB	100 0010 1101
evmhosmfa	4	RD	RA	RB	100 0010 1111
evmwhssf	4	RD	RA	RB	100 0100 0111
evmwlumi	4	RD	RA	RB	100 0100 1000
evmwhumi	4	RD	RA	RB	100 0100 1100
evmwhsmi	4	RD	RA	RB	100 0100 1101
evmwhsmf	4	RD	RA	RB	100 0100 1111
evmwssf	4	RD	RA	RB	100 0101 0011
evmwumi	4	RD	RA	RB	100 0101 1000
evmwsmi	4	RD	RA	RB	100 0101 1001
evmwsmf	4	RD	RA	RB	100 0101 1011
evmwhssfa	4	RD	RA	RB	100 0110 0111
evmwlumia	4	RD	RA	RB	100 0110 1000
evmwhumia	4	RD	RA	RB	100 0110 1100
evmwhsmia	4	RD	RA	RB	100 0110 1101
evmwhsmfa	4	RD	RA	RB	100 0110 1111
evmwssfa	4	RD	RA	RB	100 0111 0011
evmwumia	4	RD	RA	RB	100 0111 1000
evmwsmia	4	RD	RA	RB	100 0111 1001
evmwsmfa	4	RD	RA	RB	100 0111 1011
evaddusiaaw	4	RD	RA	00000	100 1100 0000
evaddssiaaw	4	RD	RA	00000	100 1100 0001
evsubfusiaaw	4	RD	RA	00000	100 1100 0010
evsubfssiaaw	4	RD	RA	00000	100 1100 0011
evmra	4	RD	RA	00000	100 1100 0100
evdivws	4	RD	RA	RB	100 1100 0110
evdivwu	4	RD	RA	RB	100 1100 0111
evaddumiaaw	4	RD	RA	00000	100 1100 1000
evaddsmiaaw	4	RD	RA	00000	100 1100 1001
evsubfumiaaw	4	RD	RA	00000	100 1100 1010

Table 7-14. Opcodes for Complex Integer Instructions, Sorted by Opcode (Continued)

Instruction	Opcode Bits				
	0–5	6–10	11–15	16–20	21–31
evsubfsmiaaw	4	RD	RA	00000	100 1100 1011
evmheusiaaw	4	RD	RA	RB	101 0000 0000
evmhessiaaw	4	RD	RA	RB	101 0000 0001
evmhessfaaw	4	RD	RA	RB	101 0000 0011
evmhousiaaw	4	RD	RA	RB	101 0000 0100
evmhossiaaw	4	RD	RA	RB	101 0000 0101
evmhossfaaw	4	RD	RA	RB	101 0000 0111
evmheumiaaw	4	RD	RA	RB	101 0000 1000
evmhesmiaaw	4	RD	RA	RB	101 0000 1001
evmhesmfaaw	4	RD	RA	RB	101 0000 1011
evmhoumiaaw	4	RD	RA	RB	101 0000 1100
evmhosmiaaw	4	RD	RA	RB	101 0000 1101
evmhosmfaaw	4	RD	RA	RB	101 0000 1111
evmhegumiaa	4	RD	RA	RB	101 0010 1000
evmhegsmiaa	4	RD	RA	RB	101 0010 1001
evmhegsmfaa	4	RD	RA	RB	101 0010 1011
evmhogumiaa	4	RD	RA	RB	101 0010 1100
evmhogsmiaa	4	RD	RA	RB	101 0010 1101
evmhogsmfaa	4	RD	RA	RB	101 0010 1111
evmwlusiaaw	4	RD	RA	RB	101 0100 0000
evmwlssiaaw	4	RD	RA	RB	101 0100 0001
evmwlumiaaw	4	RD	RA	RB	101 0100 1000
evmwlsmiaaw	4	RD	RA	RB	101 0100 1001
evmwssfaa	4	RD	RA	RB	101 0101 0011
evmwumiaa	4	RD	RA	RB	101 0101 1000
evmwsmiaa	4	RD	RA	RB	101 0101 1001
evmwsmfaa	4	RD	RA	RB	101 0101 1011
evmheusianw	4	RD	RA	RB	101 1000 0000
evmhessianw	4	RD	RA	RB	101 1000 0001
evmhessfanw	4	RD	RA	RB	101 1000 0011
evmhousianw	4	RD	RA	RB	101 1000 0100

Table 7-14. Opcodes for Complex Integer Instructions, Sorted by Opcode (Continued)

Instruction	Opcode Bits				
	0–5	6–10	11–15	16–20	21–31
evmhossianw	4	RD	RA	RB	101 1000 0101
evmhossfanw	4	RD	RA	RB	101 1000 0111
evmheumianw	4	RD	RA	RB	101 1000 1000
evmhesmianw	4	RD	RA	RB	101 1000 1001
evmhesmfanw	4	RD	RA	RB	101 1000 1011
evmhoumianw	4	RD	RA	RB	101 1000 1100
evmhosmianw	4	RD	RA	RB	101 1000 1101
evmhosmfanw	4	RD	RA	RB	101 1000 1111
evmhegumian	4	RD	RA	RB	101 1010 1000
evmhegsmian	4	RD	RA	RB	101 1010 1001
evmhegsmfan	4	RD	RA	RB	101 1010 1011
evmhogumian	4	RD	RA	RB	101 1010 1100
evmhogsmian	4	RD	RA	RB	101 1010 1101
evmhogsmfan	4	RD	RA	RB	101 1010 1111
evmwlusianw	4	RD	RA	RB	101 1100 0000
evmwlssianw	4	RD	RA	RB	101 1100 0001
evmwlumianw	4	RD	RA	RB	101 1100 1000
evmwlsmianw	4	RD	RA	RB	101 1100 1001
evmwssfan	4	RD	RA	RB	101 1101 0011
evmwumian	4	RD	RA	RB	101 1101 1000
evmwsmian	4	RD	RA	RB	101 1101 1001
evmwsmfan	4	RD	RA	RB	101 1101 1011

Chapter 8
Power Management

Power management is supported by e200 cores to minimize overall system power consumption. The e200z4 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.

8.1 Active State

The active state is the default state for the e200 core in which all of its internal units are operating 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.

8.2 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 will be completed. The **m_clk** input should remain running while in the waiting state to allow for interrupt sampling, and 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 will exit the waiting state and begin interrupt processing. The return program counter value will point 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 non-maskable 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 will return to the waiting state. While temporarily exited, the **p_waiting** output will negate, and will be re-asserted once the CPU returns to the waiting state.

8.3 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 will be completed, and the cache’s store and push buffers will be flushed. The **m_clk** input should remain running while in the Halted state to allow further transitions into the Stopped state if requested and to keep the Time Base operational if it is using **m_clk** as the clock source.

8.4 Stopped State

The stopped state is characterized as having all internal functional units of the e200 core 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.

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.

Since the Time Base unit is off during the Stopped state 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 must usually have 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 will not be 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 would allow the Time Base to be maintained during the Stopped state, and would allow a Time Base related interrupt to be generated to indicate an exit condition from the Stopped state.

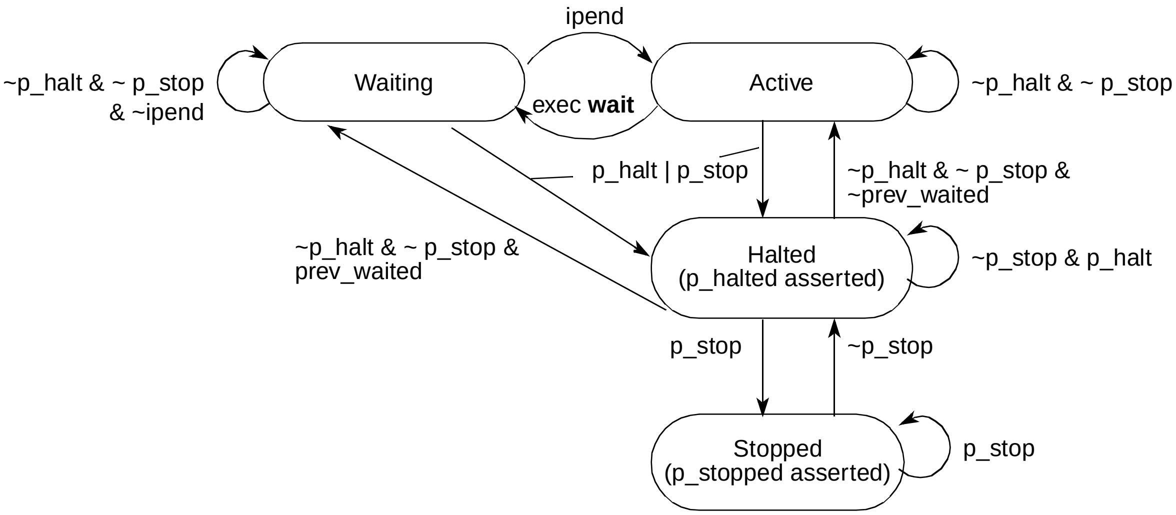


Figure 8-1. Power Management State Diagram

8.5 Power Management Pins

Table 8-1 shows how the power management pins are connected to the processor states.

Table 8-1. Power Management Pins

Pin Name	Pin Definition
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 signalled.
p_doze/p_nap/ 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

8.6 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 negated, 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 powerdown 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 powerdown state with the Time Base disabled.

8.7 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.

8.8 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 it 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.

8.9 Debug Considerations for Power Management

When a debug request is presented to the e200 core while in either the waiting, halted or stopped state, the p_wakeup signal is asserted. When m_clk is provided to the CPU, it temporarily exits 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 resamples 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 9
L1 Cache

This chapter describes the organization of the on-chip L1 instruction cache (ICache), cache control instructions, and various cache operations. It describes the interaction among the cache, the instruction unit, and the memory subsystem. This chapter also describes the replacement algorithm used for the L1 ICache.

9.1 Overview

The e200z4 processor supports a 4 Kbyte, 2 or 4-way set-associative, instruction cache (ICache) with a 32-byte line size. The ICache improves system performance by providing low-latency data to the instruction pipeline, which decouples processor performance from system memory performance. The ICache is virtually indexed and physically tagged.

Instruction addresses from the processor to the ICache 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. If the access does not match a valid cache tag entry (misses in the cache), the cache performs a line-fill transfer on the system bus.

The L1 ICache incorporates the following features:

- 4 Kbyte, 2- or 4-way configurable set-associative instruction cache
- 64-bit data, 32-bit address bus plus attributes and control
- 32-byte line size
- Cache line locking
- Way allocation
- Tag and data parity or multi-bit EDC protection with correction/auto-invalidation capability
- Virtually indexed, physically tagged
- Pseudo round-robin replacement algorithm
- Line-fill buffer
- Hit under fill

9.2 4 Kbyte ICache Organization

The e200z446n3 4 Kbyte cache is organized as either two ways of 64 sets or four ways of 32 sets with each line containing 32 bytes (four double words) of storage. Figure 9-1 shows the block diagram.

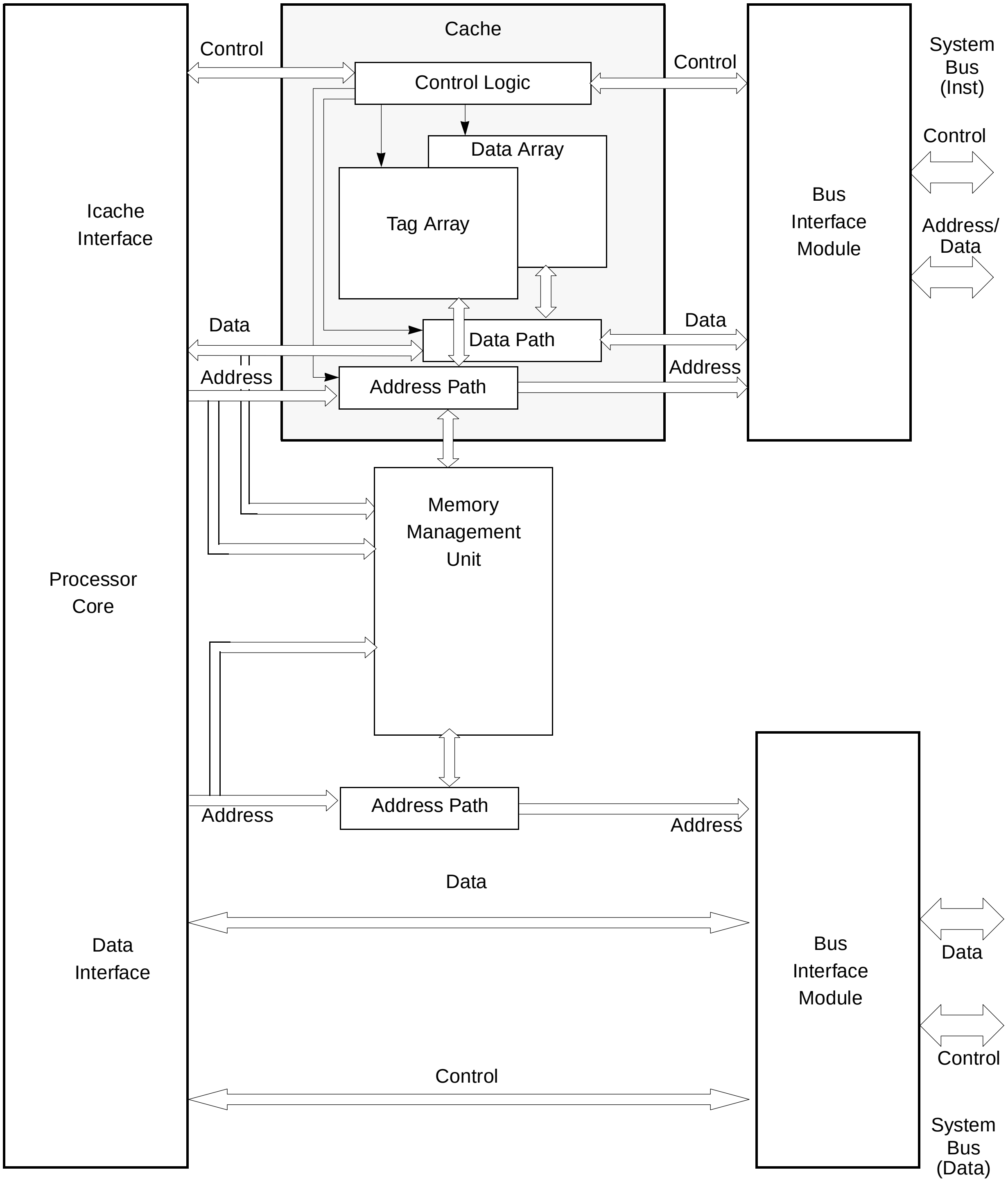
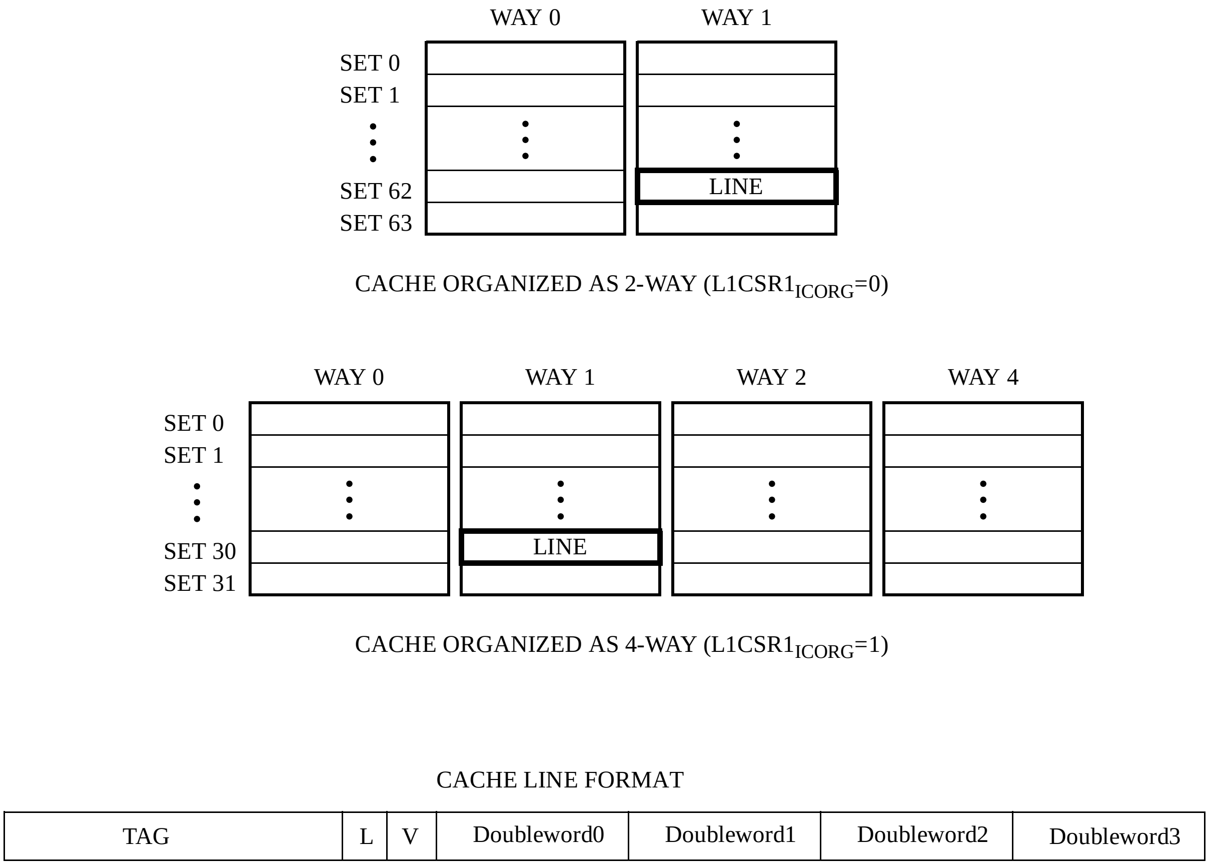


Figure 9-1. e200z446n3 ICache Block Diagram

Figure 9-2 illustrates the cache organization along with the cache line format.



TAG - 22 bit Physical Address Tag

L - Lock bit

V - Valid bit

Figure 9-2. Cache Organization and Line Format

Virtual address bits A[21–26] (A[22–26] when organized as 4-way) 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 ICache is searched for a tag match on instruction accesses from the CPU. If a match is found, the cached data is forwarded to the instruction fetch unit.

When a 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 line-fill buffer and the critical double word is forwarded to the CPU.

During a cache line fill, double words received from the bus are placed into the cache line-fill 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 line-fill buffer if the requested information has been already received.

When a cache linefill occurs, the line-fill buffer contents holding the previous linefill are placed into the cache array in one cycle.

The ICache always fills an entire line, thereby providing validity on a line-by-line basis. A cache line is always in either a valid or invalid state. Valid lines have their V bit set, indicating the line contains valid data consistent with memory. Invalid lines have their V bit clear, causing the cache line to be ignored during lookups. In addition, a cache line may be locked (L bit set), indicating the line is not available for replacement.

The ICache should be explicitly invalidated after a hardware reset; reset does not invalidate the cache lines. Following initial power-up, the ICache contents will be undefined. The L and V bits may be set on some lines, necessitating the invalidation of the cache by software before being enabled.

To determine if the address is already allocated in the cache, use the following procedure

1. Use the cache set index, virtual address bits A[21:26] (A[22:26] for the 4-way configuration) to select one cache set. A set is defined as the grouping of lines (one from each way) corresponding to the same index in the cache array.
2. Use the higher order physical address bits A[0:20] (A[0:21] for the 4-way configuration) as a tag reference or to update the cache line tag field.
3. Compare the tags from the selected cache set 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. Use virtual address bits A[27:28] to select one of the four double words in each line. A cache hit indicates that the selected double word in that cache line contains valid data.

Figure 9-3 illustrates the general flow of ICache operation.

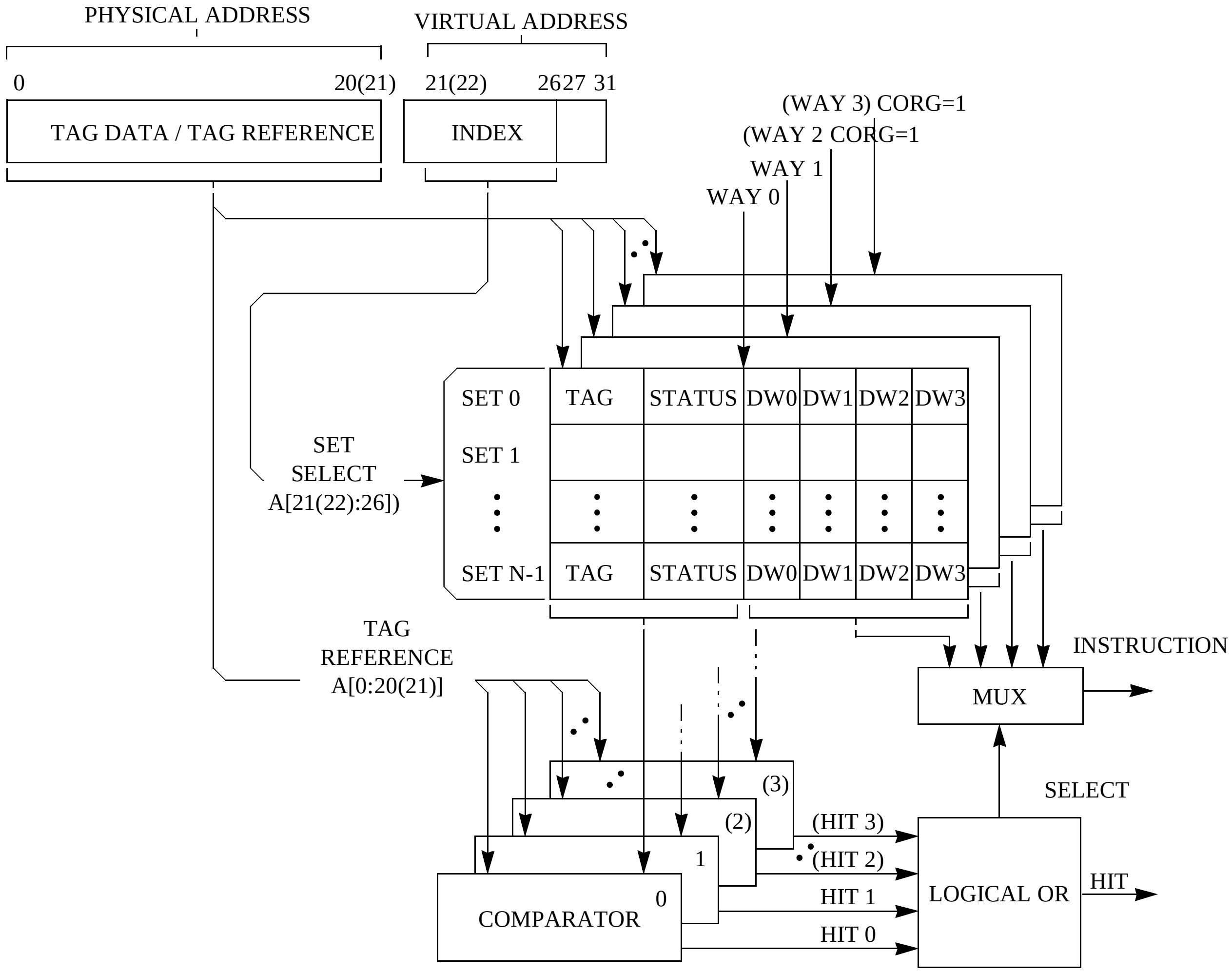


Figure 9-3. ICache Lookup Flow

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 by using the WID control bits in L1CSR0.

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 a data cache (not present in the e200z4) as well as providing general control over disabling ways in I and D 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.

WID	0
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

SPR - 1010; Read/Write; Reset - 0x0

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 in determining the replacement policy of the instruction cache.
4–31	—	Reserved[1]

[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) is a 32-bit register used for general control of the instruction cache. The L1CSR1 register is accessed using a **mfspr** or **mtspr** instruction. The SPR number for L1CSR0 is 1011 in decimal. The L1CSR1 register is shown in Figure 9-5.

0	ICECE	ICEI	0	ICEDT	0	ICUL	ICLO	ICLFC	ICLOA	ICEA	ICORG	0	ICABT	ICINV	ICE
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

SPR - 1011; Read/Write; Reset - 0x0

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[1]
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 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 on cache linefills. When ICEDT=01, a double-bit error will be injected into each double word written into the cache by inverting the two uppermost parity check bits (p_chk[0:1]). ICEI will cause injection of errors regardless of the setting of ICECE, although reporting of errors will be masked when ICECE=0.
17	—	Reserved[1]
18:19	ICEDT	Instruction Cache Error Detection Type 00 Parity Error Detection is selected 01 EDC Error Detection is selected 1x Reserved
20	—	Reserved[1]
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 will remain 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 will remain 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 cycles to complete. Clearing occurs regardless of the enable (ICE) value.
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

Table 9-2. L1CSR1 Field Descriptions (Continued)

Bits	Name	Description
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	ICORG	Cache Organization 0 The cache is organized as 64 sets and 2 ways 1 The cache is organized as 32 sets and 4 ways
28	—	Reserved[1]
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 (ICFI)	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 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 L1CSR0 cache control operations are still available.

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

9.4.3 L1 Cache Configuration Register 0 (L1CFG0)

The L1 cache configuration register 0 (L1CFG0) is a 32-bit read-only register. L1CFG0 provides information about the configuration of the e200z4 L1 cache design. Since no data cache is present on the e200z4, and the bus architecture is harvard, this register reads as all zeros in fields other than the CARCH field, which indicates ICache-only. The contents of the L1CFG0 register can be read using a **mfspr** instruction. The SPR number for L1CFG0 is 515 in decimal. The L1CFG0 register is shown in Figure 9-6.

CARCH	CWPA	CFAHA	DCFISWA	0	DCBSIZE	DCREPL	DCLA	DCPA	DCNWAY	DCSIZE
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
10	0	0	0	0 0	00	00	0	0	00000000	00000000000

SPR - 515; Read-only; 0x80000000

Figure 9-6. L1 Cache Configuration Register 0 (L1CFG0)

9.4.4 L1 Cache Configuration Register 1 (L1CFG1)

The L1 Cache Configuration Register 1 (L1CFG1) is a 32-bit read-only register. L1CFG1 provides information about the configuration of the L1 instruction cache design. The contents of the L1CFG1 register can be read using a **mfspr** instruction. The SPR number for L1CFG1 is 516 in decimal. The L1CFG1 register is shown in Figure 9-7.

0	ICFISWA	0	ICBSIZE	ICREPL	ICLA	ICECA	ICNWAY	ICSIZE
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
0000	1	0 0	00	10	1	1	00000001 (2way)/ 00000011 (4 way)	00000000100 (4 Kbyte)

SPR - 516; Read-only

Figure 9-7. L1 Cache Configuration Register 1 (L1CFG1)

The L1CFG1 bits are described in Table 9-3.

Table 9-3. 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 APU Available 1 The instruction cache implements the line locking functionality

Table 9-3. L1CFG1 Field Descriptions (Continued)

Bits	Name	Description
12	ICECA	Instruction Cache Error Checking Available 1 The instruction cache implements error checking
13–20	ICNWAY	Instruction Cache Number of Ways 0x01 The instruction cache is 2-way set-associative (when configured by L1CSR1[ICORG] = 0) 0x03 The instruction cache is 4-way set-associative (when configured by L1CSR1[ICORG] = 1)
21–31	ICSIZE	Instruction Cache Size 0x004 The size of the instruction cache is 4 Kbytes

9.5 Cache Organization Control

The cache may be configured by software to structure the cache into the following two organizations:

- A 2-way, 64 set organization
- A 4-way, 32 set organization

The control is provided via the L1CSR1[ICORG] control bit. The default organization following reset is 2-way, 64 sets.

NOTE

Software must disable and invalidate the cache prior to altering the cache organization and re-enabling the cache to prevent unexpected behavior.

When organized as 2-way, less power may be consumed, since a maximum of 2 ways will be enabled on each cache access, but performance may be affected due to the reduced associativity.

The cache configuration is reflected in the content of L1CFG1, and will be representative of the current cache organization based on L1CSR1[ICORG].

9.6 Cache Operation

This section explains the cache operation.

9.6.1 Cache Enable/Disable

The ICache is enabled or disabled by using the L1CSR1[ICE] cache enable bit. The cache enable bit is cleared by power-on reset or normal reset, disabling the ICache.

When the ICache is disabled, the cache tag status bits are ignored, and the cache is not accessed for instruction fetches. All instruction fetch 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) is independent of the state of L1CSR1[ICE]. Disabling the ICache does not affect the translation logic in the memory management unit. Translation attributes are still used when generating attribute information on the system bus.

Altering the 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 invalidated 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** L1FINV1 and **mtspr** L1CSR{0,1}) do not affect a cache when it is disabled.

9.6.2 Cache Fills

Cache line fills are requested when a cacheable instruction miss occurs. The cache line fill is performed critical double word first on the bus, using a burst access. The critical double word is forwarded to the instruction unit before being written to the cache, thus minimizing stalls due to fill delays. Cache line fills load a four double word line-fill buffer, and updates to the cache array are performed in a single cycle when the next cache line fill is initiated. Subsequent instruction accesses may hit in the line-fill buffer and data supplied from the buffer to the CPU.

Data may be streamed to the instruction fetch unit as it arrives from the bus if a corresponding request is pending. In addition, the cache supports hit under fill, allowing subsequent instruction 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 line-fill buffer is subsequently requested by the CPU.

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 is terminated and the cache line marked invalid.

9.6.3 Cache Line Replacement

On an ICache miss, the cache controller uses a pseudo-round-robin replacement algorithm to determine which cache line is selected to be replaced. There is a single replacement counter for the 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 line fill occurs.

The replacement counter is initialized to point to way 0 on a reset or on a respective cache invalidate all operation. The replacement counter may also be set to a specific value by a L1FINV1 command.

9.6.4 Cache-Inhibited Accesses

When the cache-inhibited attribute is indicated by translation and on the cache lookup any line has a tag parity/EDC error, or if a cache miss occurs, the access is performed as single beat transactions on the system bus and the cache is ignored. Cache inhibited status is ignored on all cache hits that do not incur a parity/EDC error. If a cache hit occurs to a line with a data error on the requested double word, the hit is ignored and treated as a miss and a single beat access is performed. The cache remains unchanged and no cache parity/EDC error is reported.

9.6.5 Cache Invalidation

The e200z4 supports full invalidation of the ICache under software control. The ICache may be invalidated through the L1CSR1[ICINV] cache invalidate control bit. This function is available even when the ICache is disabled.

Reset does not invalidate the ICache automatically. Software must use the ICINV control for invalidation after a reset. Proper use of this bit is to determine that it is clear and then to set it with a pair of **mfspr mtspr** operations. A 0-to-1 transition on {D,I}CINV causes a flash invalidation to be initiated which lasts for multiple CPU cycles. Once set, the 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 L1CSR1 that attempts to change the state of 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 the L1CSR1[ICEDT] field should be configured properly at the time the invalidation operation is initiated. A subsequent change to the error detection type control requires an invalidation to avoid improper interpretation of previously stored tag parity/check bits.

During the process of performing the invalidation, the ICache does not respond to accesses and remains busy. Interrupts may still be recognized and processed, potentially aborting the invalidation operation. When this occurs, the L1CSR1[ABT] bit is set to indicate unsuccessful completion of the operation. Software should read the L1CSR1 register to determine that the operation has completed (L1CSR1[ICINV] bit cleared) and then check the status of the L1CSR1[ABT] bit to determine completion status.

NOTE

> Note that while most implementations of the e200z4 stall further instruction execution during this invalidation interval, it is not guaranteed across all implementations. Thus software should be written using these guidelines.

Individual cache lines may be invalidated using **icbi** instruction. This instruction requires the cache to be enabled in order to operate normally.

9.6.6 Cache Invalidate by Set and Way

The e200z4 supports cache invalidation under software control. The ICache may be invalidated by index and way through a **mtspr l1finv1** instruction.

9.6.6.1 L1FINV1

The L1 flush and invalidate control register (L1FINV1) is a 32-bit SPR used to select a cache set and way to be invalidated. No tag match is required. This function is available even when the ICache is disabled.

For invalidation operations, a tag parity error or EDC error will not prevent the line from being invalidated, and no error is reported.

The SPR number for L1FINV1 is 959 in decimal. The L1FINV1 register is shown in Figure 9-8.

0						CWAY		0													CSET						0			CCMD	
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

SPR - 959; Read/Write; Reset - 0x0

Figure 9-8. L1 Flush/Invalidate Register 1 (L1FINV1)

The L1FINV1 bits are described in Table 9-4.

Table 9-4. L1FINV1 Field Descriptions

Bits	Name	Description
0–5	—	Reserved[1] for way extension
6–7	CWAY	Cache Way Specifies the cache way to be selected. When configured for 2 ways, values greater than 1 result in undefined command operations.
8–20	—	Reserved[1] for set extension
21–26	CSET	Cache Set Specifies the cache set to be selected. When configured for 4 ways, values greater than 31 result in undefined command operations.
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.

9.7 Cache Parity and EDC Protection

Cache parity is supported for both the tag and data arrays of the ICache. Six parity check bits are provided for each tag entry, and eight parity check bits are provided for each double word in the data arrays of the ICache. These bits can be used for either standard parity checking (single-bit error detection) or for multi-bit error detection (EDC—DED, double error detection). When utilizing EDC protection, many multi-bit errors are also detected. Parity or EDC checking is controlled by the L1CSR1[ICECE] and L1CSR1[ICEDT] control fields.

Data parity or EDC errors are ignored for cache misses because the parity is updated for the new line-fill data being stored.

Parity or EDC errors are not signaled when cache error checking is disabled (L1CSR1[ICECE] = 0).

Signaling of a parity error or EDC error may optionally cause a machine check exception to occur and one or more syndrome bits to be set in the machine check syndrome register. If correction/auto-invalidation on error is enabled (L1CSR1[ICEA] = 01), this occurs only when a line that was locked is invalidated due to an uncorrectable tag parity or EDC error. If machine check generation on error is enabled (L1CSR1[ICEA] = 00), machine checks are generated when a parity/EDC error is signaled.

Refer to Section 5.7.2, “Machine Check Interrupt (IVOR1),” and to Section 2.4.7, “Machine Check Syndrome Register (MCSR),” for a description of machine check conditions.

9.7.1 Cache Error Action Control

The L1CSR1[ICEA] control field allows for selection of several policies to apply when errors are detected during a cache lookup. They are further described in the following subsections.

9.7.1.1 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. Any parity or EDC error that could result in incorrect operation causes 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 (L1CSR1[CEA] = 00) and a parity or EDC error is detected on any portion of the accessed tags for a cacheable access, a machine check is reported, regardless of whether a cache hit or miss occurs. A machine check is also reported if a cache hit occurs and a parity or EDC error is detected on any portion of the accessed double word of data for an instruction access. For cache inhibited accesses, no errors are reported. If a parity/EDC error occurs on a cache lookup, the cache is ignored, a single-beat bus access is performed, and no parity/EDC error is generated.

9.7.1.2 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 causing cache refills to reload correct data from memory, without generation of exceptions. Exceptions are only generated when invalidations could cause a change in correct behavior, such as changing the locked status of a line. Parity or EDC errors do not generate machine check exceptions for cache-inhibited accesses. If a parity/EDC error occurs on a cache-inhibited access, a single-beat bus access is performed, the cache is ignored, and no parity/EDC error is generated.

When using EDC protection for the cache tags (L1CSR1[ICEDT] = 01), single-bit tag errors are corrected by the cache hardware during a correction/auto-invalidation cycle. For cacheable accesses, lines with multi-bit errors are invalidated on cache lookups. When operating with only parity protection for the cache tags (L1CSR1[ICEDT] = 00), cache entries with detectable tag errors are invalidated rather than corrected by the cache hardware during a correction/auto-invalidation cycle.

Note that due to the relative storage capacities, the data arrays have a higher probability of incurring an error than the tag arrays. Therefore, most errors are transparently corrected, even if they are double-bit or multi-bit errors.

If correction/auto-invalidation on error is enabled (L1CSR1[ICEA] = 01) and an error is detected on any portion of the accessed tags for a cacheable 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 error is corrected if possible (correction is not possible during operation with only parity protection for the tags), and rewritten to correct the stored error. Tag entries with uncorrectable errors are invalidated. 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 or EDC error and a parity or EDC error is detected on any portion of the accessed double word of data, a miss is forced. 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 or EDC error and no parity or EDC error is detected on any portion of the accessed double word of data, data parity or EDC errors on all other lines are ignored. No invalidations occur.

For all cases of invalidations, if any locked line was invalidated, a machine check occurs, even though correction/auto-invalidation is selected. Invalidation is not blocked for locked lines on cacheable accesses. The lock bit remains unmodified by the invalidation operation to allow potential software recovery.

If a refill of a locked line due to a data parity error encounters an external bus error during the line fill, a machine check is generated; the line is invalidated; and the lock bit remains set.

9.7.2 Parity/EDC Error Handling for Cache Control Operations and Instructions

Parity/EDC errors are not signaled when the L1CSR1[ICECE] cache error checking enable bit is cleared. The following sections describe error handling for cache control operations and cache control instructions when set.

9.7.2.1 L1FINV1 operations

For invalidation operations by the L1FINV1 control register, tag parity or EDC errors result in the specified line being invalidated. No error is reported, regardless of the setting of L1CSR1[ICEA]. Data parity or EDC errors are ignored. Parity or EDC errors on all other ways not specified by the CWAY value for the L1FINV1 are ignored, regardless of the settings of L1CSR1[ICEA].

9.7.2.2 Cache Touch Instructions (icbt)

Parity/EDC errors are not signaled on a lookup for an **icbt** instruction. For those instructions, a parity or EDC error results in a nop and no error is reported, regardless of error checking being enabled. No invalidations will occur.

9.7.2.3 icbi Instructions

For **icbi** instructions, on a hit to any line without a tag parity/EDC error or on a hit to an unlocked line with a 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’ and 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 that miss are unaffected. No machine check is generated.

If a hit occurs to a line with a tag parity/EDC error that is locked (after a correction for L1CSR1[ICEA] = ‘01’), the line is left unaffected. No machine check is generated, regardless of the setting of L1CSR1[ICEA].

Otherwise, if a miss occurs, all parity/EDC errors are ignored; the lines are left unaffected; and no machine check is generated, regardless of the setting of L1CSR1[ICEA].

All data parity or EDC errors are ignored regardless of L1CSR1[ICEA].

9.7.2.4 Cache Locking Instructions (icbtls, icblc)

For **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 parity or EDC errors are ignored. When L1CSR1[ICEA] = ‘00’, tag parity/EDC errors on other lines are ignored. When L1CSR1[ICEA] = ‘01’, uncorrectable tag parity or EDC errors on other lines also causes unlocked lines to be invalidated, regardless of hit or miss. No machine check is generated regardless of the setting of L1CSR1[ICEA].

For cacheable **icbtls** instructions that hit only to a line with a tag parity or EDC error or that miss in all ways, a machine check is generated if L1CSR1[ICEA] = ‘00’ and any line has a tag parity/EDC error. If L1CSR1[ICEA] = ‘01’, lines with an uncorrectable tag parity/EDC error are invalidated. If a line that was locked was invalidated, a machine check is generated.

For cacheable **icblc** instructions that hit only to a line with a tag parity or EDC error or that miss in all ways, a machine check is generated if L1CSR1[ICEA] = ‘00’ and any line with a tag parity/EDC error is locked. If L1CSR1[ICEA] = ‘01’, lines with an uncorrectable tag parity/EDC error are invalidated. If a line that was locked or had a lock parity error was invalidated, a machine check is generated.

9.7.3 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 on any line 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 errors if no line is locked, and invalidations are performed for uncorrectable tag errors. If a cache-inhibited 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. No machine check is generated in this case. 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. No machine check is generated in this case.

9.7.4 EDC Checkbit/Syndrome Coding Scheme Generation

When operating with EDC enabled (L1CSR1[ICEDT] =01), double bit error detection codes are used to protect both the tag and data portions of a 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 the following figures.

Table 9-5 shows the checkbit coding for each tag entry. A '*' in the table indicates the bit is XORed to form the final checkbit value.

Table 9-5. Tag Checkbit Generation

Checkbit 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 9-6 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 9-6. Data Checkbit Generation

Checkbit 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 9-6. Data Checkbit Generation (Continued)

Checkbit 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				*			*	*				*	*	*			*	*	*						*	*	*	*	*	*	*	*

9.7.5 Cache Error Injection

Cache error injection provides a way to test error recovery by intentionally injecting parity errors into the instruction cache 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.

Cache parity error injection is not performed for cache debug write accesses because parity bit values written can be directly controlled (See Section 9.15.3, “Cache Debug Access Control Register (CDACNTL)”).

In order to clear the parity errors, a cache invalidation or an invalidation of the lines that could have had an injected parity error may be performed. Line invalidation may be performed by an **icbi** instruction or an L1FINV1 invalidation operation.

9.7.6 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. Refer to Section 13.4.3, “Cache Error Cross-Signaling Operation” and Section 13.3.11, “Cache Error Cross-signaling Signals” for more details of operation.

9.8 Cache Management Instructions

This section describes the implementation of the cache management instructions in the e200z4 core. See the *EREF* for complete descriptions.

- **icbi—**Instruction Cache Block Invalidate

  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 line fill is in progress and the line-fill data corresponds to the EA associated with an **icbi**, the instruction cache is not updated with line-fill data.
- **icbt—**Instruction Cache Block Touch

  If $\text{HID0}_{\text{NOPTI}}$ is set, this instruction is treated as a no-op.
- **dcba—**Data Cache Block Allocate

This instruction is treated as a no-op.

- **dcbf—**Data Cache Block Flush

  This instruction is treated as a no-op.
- **dcbi—**Data Cache Block Invalidate

  This instruction is privileged and treated as a no-op in supervisor mode.
- **dcbst—**Data Cache Block Store

  This instruction is treated as a no-op.
- **dcbt—**Data Cache Block Touch

  This instruction is treated as a no-op.
- **dcbtst—**Data Cache Block Touch

  This instruction is treated as a no-op.
- **dcbz—**Data Cache Block Set to Zero

  This instruction causes an Alignment exception.

9.9 Touch Instructions

Due to the limitations of using **icbt** instructions, a program that uses these instructions improperly may actually see a degradation in performance from their use. To avoid this, the e200z4 provides the HID0[NOPTI] control bit to cause these instructions to be treated as no-ops

9.10 Cache Line Locking/Unlocking

This section describes the cache line locking and unlocking functionality.

9.10.1 Overview

The e200z4 supports the Freescale EIS cache line locking category, which defines user-mode instructions that 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 no-op instructions by the e200z4.

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] bit 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-op) and sets ESR[ILK]. If MSR[UCLE] is set, cache-locking instructions can be executed in user-mode and do not take a DSI for cache-locking. However, they may still cause a DSI for access violations or cause machine checks for parity or external termination errors.

There are cases when attempting to set a lock fails even when no DSI or DTLB exceptions occur. These are as follows:

- Target address is marked cache-inhibited and a cache miss occurs.

- Cache is disabled or all ways of the cache are disabled for replacement.
- 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 L1CSR1[CUL], the cache unable to lock bit, is set.

For **icbtls,** an attempt is made to lock the corresponding cache line assuming no exception conditions occur (DSI or DTLB error). 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 results in an overlocking situation. In this case, L1CSR1[CLO], the cache overlock bit, 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 L1CSR1[CLOA] bit.

The cache unable to lock conditions have priority over the cache overlock 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 9-7.

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.

Cache line locking interacts with the ability to control replacement of lines in certain cache ways by the L1CSR0 WID control bits. If an **icbtls** cache line locking instruction 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 field. In this case, no replacement is 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 by WID. 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:

1. Add the line address to its list of locked lines.
2. Execute the appropriate set lock instruction to lock the cache line.
3. Modify save/restore register 0 to point to the instruction immediately after the locking instruction that caused the DSI.
4. Execute an **rfi**.

If the locking instruction is a clear lock instruction, and if the handler decides to unlock the line, it should do the following:

1. Remove the line address from its list of locked lines.
2. Execute the appropriate clear lock instruction to unlock the cache line.
3. Modify save/restore register 0 to point to the instruction immediately after the locking instruction that caused the DSI.
4. Execute an **rfi**.

9.10.2 icbtls—Instruction Cache Block Touch and Lock Set

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 the MSR[UCLE] (user-mode cache lock enable) bit is set, **icbtls** 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 cause the instruction to be treated as a no-op.

The e200z4 only supports CT = 0. If CT is some value other than 0, the **icbtls** is treated as a no-op and the L1CSR1[ICUL] bit is set, indicating an unable-to-lock condition occurred. No other exceptions are reported. If the instruction cache is disabled, the **icbtls** is treated as a no-op and the L1CSR1[ICUL] bit 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 causes a DSI interrupt for access violations. It also causes 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 treated as a no-op, no DSI is taken due to the cache-inhibited status, and L1CSR1[ICUL] 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 cannot be locked, the L1CSR1[ICUL] bit is set, indicating an unable-to-lock condition occurred. This also occurs whenever **icbtls** must be treated as a no-op.

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 the L1CSR1[ICLO] bit is set, indicating an over-lock 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 if 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)
```

9.10.3 icblc—Instruction Cache Block Lock Clear

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 MSR[UCLE] (user-mode cache lock enable) 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 cause the instruction to be treated as a no-op.

The e200z4 only supports CT = 0. If CT is some value other than 0, the **icblc** is treated as a no-op. No other exceptions are reported. If the instruction cache is disabled, the **icblc** is treated as a no-op. No other exceptions are reported.

The **icblc** instruction requires either execute or read (X or R) permissions with respect to translation. It causes a DSI interrupt for access violations and a Data TLB error interrupt for the target addresses that cannot be translated.

9.10.4 Effects of Other Cache Instructions on Locked Lines

icbt has no effect on the state of a cache line's lock bit.

icbi invalidates and unlocks a cache line in the L1 ICache.

9.10.5 Flash Clearing of Lock Bits

The e200z4 supports flash clearing of cache lock bits under software control by using the ICFCL (cache flash clear locks) control bit in the L1CSR1 register.

Lock bits are not cleared automatically upon power-up (m_por) or normal reset (p_reset_b). Software must use the ICLFC 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 ICLFC causes a flash clearing of the lock bits to be initiated, which lasts for multiple CPU cycles. Once set, the ICLFC bit is 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 L1CSR1 that attempts to change the state of L1CSR1[ICLFC] 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, L1CSR1[ABT] is set to indicate unsuccessful completion of the operation. Software should read the L1CSR1 register to determine that the operation has completed (L1CSR1[ICLFC] cleared), and then check the status of L1CSR1[ABT] to determine completion status.

NOTE

Note that while most implementations of the e200z4 will stall further instruction execution during this flash clearing interval, it is not guaranteed across all implementations. Thus, software should be written using these guidelines.

9.11 Cache Instructions and Exceptions

All cache management instructions (except **icbt**) can generate DSI exceptions due to permission violations or TLB miss exceptions if the effective address cannot be translated.

The cache locking instructions **icblc** and **icbtls** generate DSI exceptions if the MSR[UCLE] bit is clear and the locking instruction is executed in user mode (MSR[PR] = 1). Instruction cache locking instructions that result in a DSI exception for this reason set the ESR[ILK] bit (documented as DLK1 in the *EREF*).

9.11.1 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 9-7.

Table 9-7. Special Case Handling

Operation	CT!=0	Cache Disabled	TLB Miss	User & UCLE = 0	Protection Violation	Cache Parity Error	CI and Miss in Cache	All Available Ways Locked	External Terminatio n Error
icbt	NOP	NOP	NOP	—	NOP	NOP	NOP	NOP	NOP
icbtls icblc	ICUL NOP	ICUL NOP	DTLB DTLB	ILK ILK	DSI DSI	MC MC	ICUL —	ICLO —	MC —

Table 9-7. Special Case Handling

Operation	CT!=0	Cache Disabled	TLB Miss	User & UCLE = 0	Protection Violation	Cache Parity Error	CI and Miss in Cache	All Available Ways Locked	External Terminatio n Error
icbi	—	NOP	DTLB	—	DSI	—	—	—	—

Note:

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; DTLB = data TLB interrupt
ICUL = no-op, and set L1CSR1[CUL]
ICLO = no-op, and set L1CSR1[CLO]
ILK = data storage interrupt (DSI) and set ESR[ILK]
MC = Machine Check and update MCAR

9.11.2 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 9-8 shows the definitions of the p_d_ttype[0:5] encodings.

Table 9-8. Transfer Type Encoding

p_d_ttype[0:4]	Transfer Type	Instruction
00000	Normal	normal loads/stores
00001	Atomic	**lbarx, lharx, lwarx, stbcx., sthcx., and stwcx.**
00010	Reserved for Flush Data Block	Reserved for **dcbst**
00011	Reserved for Flush and Invalidate Data Block	Reserved for **dcbf**
00100	Reserved for Allocate and Zero Data Block	Reserved for **dcbz**
00101	Reserved for Invalidate Data Block	Reserved for **dcbi**
00110	Invalidate Instruction Block	**icbi**
00111	Multiple Word Load/Store	**lmw**, **stmw**
01000	TLB Invalidate	**tlbivax**
01001	TLB Search	**tlbsx**
01010	TLB Read entry	**tlbre**
01011	TLB Write entry	**tlbwe**
01100	Touch for Instruction	**icbt**
01101	Lock Clear for Instruction	**icblc**

Table 9-8. Transfer Type Encoding

p_d_ttype[0:4]	Transfer Type	Instruction
01110	Touch for Instruction and Lock Set	**icbtls**
01111	Reserved for Lock Clear for Data	Reserved for **dcblc**
10000	Reserved for Touch for Data	Reserved for **dcbt**
10001	Reserved for Touch for Data Store	Reserved for **dcbtst**
10010	Reserved for Touch for Data and Lock Set	Reserved for **dcbtls**
10011	Reserved for Touch for Data Store and Lock Set	Reserved for **dcbtstls**

9.12 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 multiprocessor cache-coherent systems.

9.13 Page Table Control Bits

The Power ISA embedded category architecture allows certain memory characteristics to be set on a page and 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. Generally, certain mixing of WIMG settings is allowed by the Power ISA; however, others may present cache coherence paradoxes and are considered programming errors.

9.13.1 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 instruction fetch operation. Cache inhibited status is ignored on all cache hits.

9.14 Effect of Hardware Debug on Cache Operation

Hardware debug facilities utilize normal CPU instructions to access register and memory contents during a debug session. During hardware debug, the MMU page attributes are controllable by the debug firmware by means of the OnCE control register (OCR) settings. Refer to Section 11.4.6.3, “e200 OnCE Control Register (OCR).”

9.15 Cache Memory Access For Debug/Error Handling

The cache memory provides resources needed to do foreground accesses by means of **mtdcr** instructions executed by the processor or background accesses by means of the JTAG/OnCE port to read and write the cache SRAM arrays. Accesses are supported by 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 or by debug and 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 by 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 line-fill 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 are written to external memory and the late write and line-fill buffers are written to the cache arrays.

9.15.1 Cache Memory Access By Means Of 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 9-9 and Table 9-10). 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 or data values. The second step is to write the CDACNTL register with desired tag or data location, full parity information (for data writes only), and assert the R/W and GO bits in CDACNTL.

For a cache read access, software must first access and write the CDACNTL register with the 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 full parity information (data reads only).

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 updates all 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 that is not being modified to properly compute or store all 8 parity/EDC check bits when the modified 32-bit data is written. Tag writes are accomplished in a single pass.

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
                .
                .
```

9.15.2 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 9-9 and Table 9-10). 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 information (for data writes only), and assert the R/W and GO bits in CDACNTL.

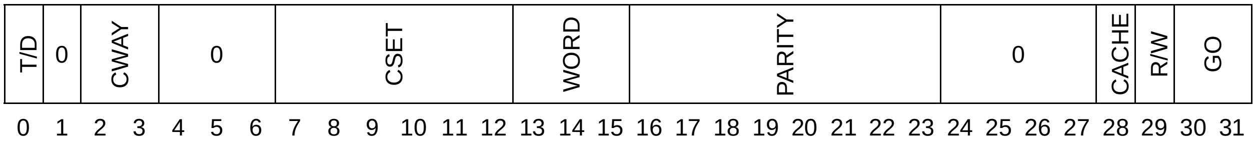
For a cache read access, the user must first access and write the CDACNTL register with desired tag or data location, and assert the R/W and GO bits in CDACNTL. The second step is to access and read the CDADATA register for the tag or data and read the CDACNTL register for parity (data reads only).

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 the validity of any other information in the CDACNTL or CDADATA registers.

9.15.3 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 data SRAM parity bit values that must be supplied by the user for write accesses and which will be supplied by the cache for read accesses of the data SRAM arrays.

The CDACNTL register is shown in Figure 9-9.



DCR - 351; Read/Write; Reset - 0x0

Figure 9-9. CDACNTL Register

Table 9-9 provides bit definitions for the cache debug access control register.

Table 9-9. Cache Debug Access Control Register Definition

Bit(s)	Name	Description
0	T/D	Tag/Data: 0 Data array selected 1 Tag array selected
1	—	Reserved[1]
2–3	CWAY	Cache Way Specifies the cache way to be selected. When configured for 2 ways, values greater than 1 result in undefined command operations.
4–6	—	Reserved[1]
7–12	CSET	Cache Set: Specifies the cache set to be selected. When configured for 4 ways, values greater than 31 result in undefined command operations.
13–15	WORD	Word (Data array access only) Specifies one of eight words of selected set
16–23	PARITY/EDC CHECK BITS	Parity check bits[2] Parity Mode (L1CSR1[ICEDT] = 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 (L1CSR1[ICEDT] = 01): Data Array: parity check bits for data. Bits 16–23 correspond to p_dchk[0–7] (See Table 9-6). Tag Array: parity check bits for tag. Bits 16–21 correspond to p_tchk[0–5] (See Table 9-5). Bits 22–23 reserved.
24–27	—	Reserved[1]

Table 9-9. Cache Debug Access Control Register Definition (Continued)

Bit(s)	Name	Description
28	CACHE	Cache Select Specifies the cache to be selected 0 Selects the data cache or unified cache for the operation. (No-operation on e200z4) 1 Selects the instruction cache for the operation. **Note:** This bit should be written with a ‘1’ by software to access the ICache. This bit is provided for compatibility with other e200 processors.
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 if the access is to the data array, 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.

9.15.3.1 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.

The CDADATA register is shown in Figure 9-10.

TAG or DATA

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

DCR - 350; Read/Write; Reset - Undefined/Unaffected

Figure 9-10. CDADATA Register

Table 9-10 provides bit definitions for the cache debug access data register.

Table 9-10. Cache Debug Access Data Register Bit Definitions

Bit(s)	Name	Description
0–31	TAG	TAG Array Access Data 0–21 Tag compare bits 22 Reserved 23 Valid bit 24 Lock bit 27–31 Reserved, write as zero
	DATA	DATA Array Access Data (Bytes 0–3 of the selected word) 0–7 Byte 0 8–15 Byte 1 16–23Byte 2 24–31Byte 3

9.16 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 an **mfspr** or **mtspr** instruction. The SPR number for HDBCR0 is 976 in decimal.

The HDBCR0 register is shown in Figure 9-11.

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																															ISTRM

SPR - 976; Read/Write; Reset - 0x0; Supervisor-only

Figure 9-11. Hardware Debug Control Register 0 (HDBCR0)

The HDBCR0 bits are described in Table 9-11.

Table 9-11. HDBCR0 Field Descriptions

Bits	Name	Description
0–30	—	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.

Chapter 10
Memory Management Unit

10.1 Overview

The memory management unit is a 32-bit Power ISA embedded category-compliant implementation, with the following feature set:

- Virtual memory support
- 32-bit virtual and physical addresses
- 8-bit process identifier
- 16-entry fully associative TLB
- Hardware assist for TLB miss exceptions
- Per-entry multiple page size support from 1 Kbyte to 4 Gbyte
- Entry flush protection
- Software managed by **tlbre**, **tlbwe**, **tlbsx**, **tlbsync**, and **tlbivax** instructions
- Freescale EIS MMU architecture compliant
- 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 discusses the following topics:

- 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”

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 which 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 e200z4 MMU supports twenty-three page sizes (1 Kbyte, 2 Kbyte, 4 Kbyte, 8 Kbyte, 16 Kbyte, 32 Kbyte, 64 Kbyte, 128 Kbyte, 256 Kbyte, 512 Kbyte, 1 Mbyte, 2 Mbyte, 4 Mbyte, 8 Mbyte, 16 Mbyte, 32 Mbyte, 64 Mbyte, 128 Mbyte, 256 Mbyte, 512 Mbyte, 1 Gbyte, 2 Gbyte, 4 Gbyte). In order 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 are 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 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). An address space value of 0b1 can be used to denote non interrupt-related address spaces(or possibly all user 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 Book E 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 e200z4 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 either match 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 to the corresponding fields in the TLB entries.

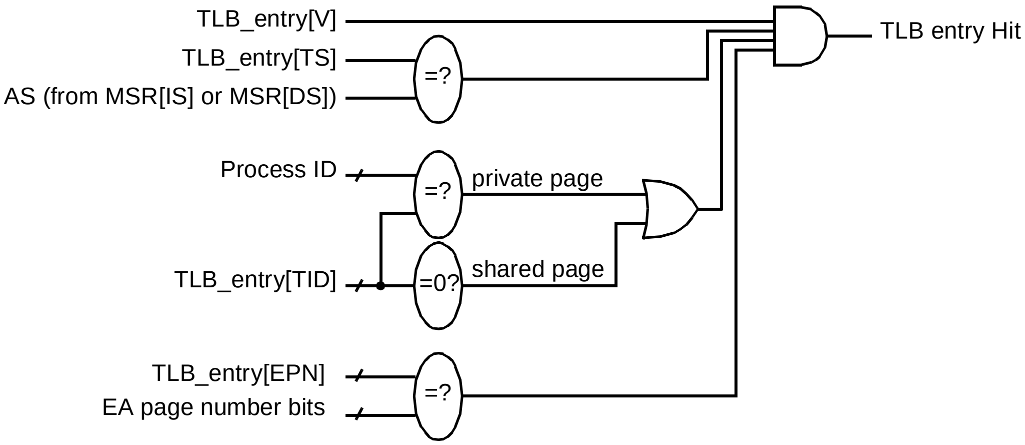


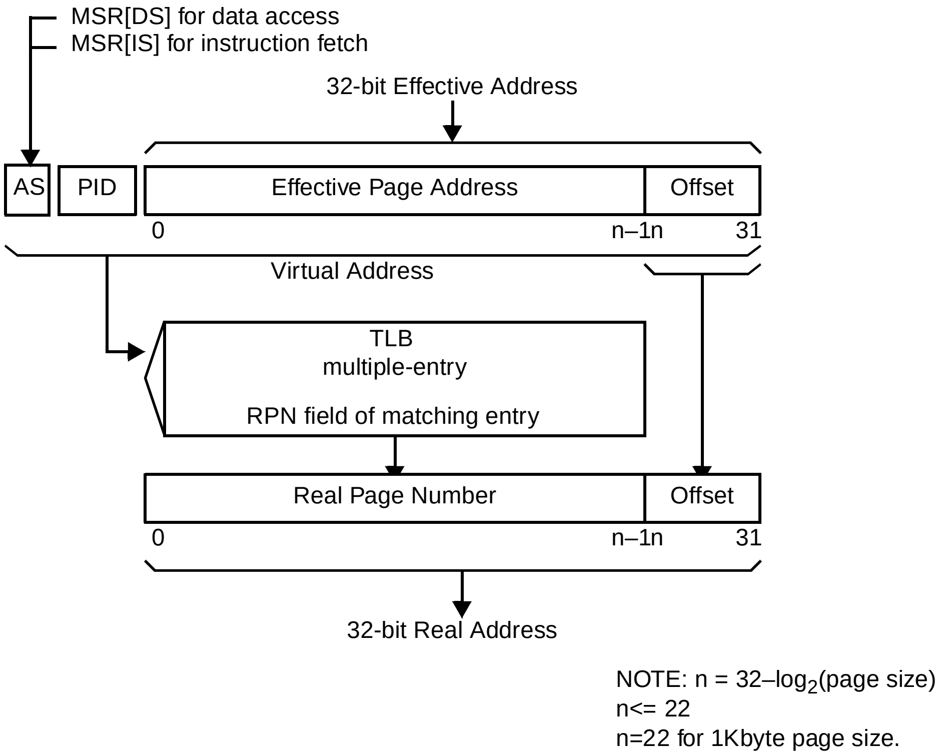
Figure 10-1. Virtual Address and TLB-Entry Compare Process

The page size defined for a TLB entry determines how many bits of the effective address are compared to 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 and EPN Field Comparison

SIZE Field	Page Size (2^{SIZE}Kbytes)	EA to EPN Comparison
0b00000	1 Kbyte	EA[0:21] =? EPN[0:21]
0b00001	2 Kbyte	EA[0:20] =? EPN[0:20]
0b00010	4 Kbyte	EA[0:19] =? EPN[0:19]
0b00011	8 Kbyte	EA[0:18] =? EPN[0:18]
0b00100	16 Kbyte	EA[0:17] =? EPN[0:17]
0b00101	32 Kbyte	EA[0:16] =? EPN[0:16]
0b00110	64 Kbyte	EA[0:15] =? EPN[0:15]
0b00111	128 Kbyte	EA[0:14] =? EPN[0:14]
0b01000	256 Kbyte	EA[0:13] =? EPN[0:13]
0b01001	512 Kbyte	EA[0:12] =? EPN[0:12]
0b01010	1 Mbyte	EA[0:11] =? EPN[0:11]
0b01011	2 Mbyte	EA[0:10] =? EPN[0:10]
0b01100	4 Mbyte	EA[0:9] =? EPN[0:9]
0b01101	8 Mbyte	EA[0:8] =? EPN[0:8]
0b01110	16 Mbyte	EA[0:7] =? EPN[0:7]
0b01111	32 Mbyte	EA[0:6] =? EPN[0:6]
0b10000	64 Mbyte	EA[0:5] =? EPN[0:5]
0b10001	128 Mbyte	EA[0:4] =? EPN[0:4]
0b10010	256 Mbyte	EA[0:3] =? EPN[0:3]
0b10011	512 Mbyte	EA[0:2] =? EPN[0:2]
0b10100	1Gbyte	EA[0:1] =? EPN[0:1]
0b10101	2 Gbyte	EA[0] =? EPN[0]
0b10110	4Gbyte	(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=22 for 1Kbyte 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 and 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.

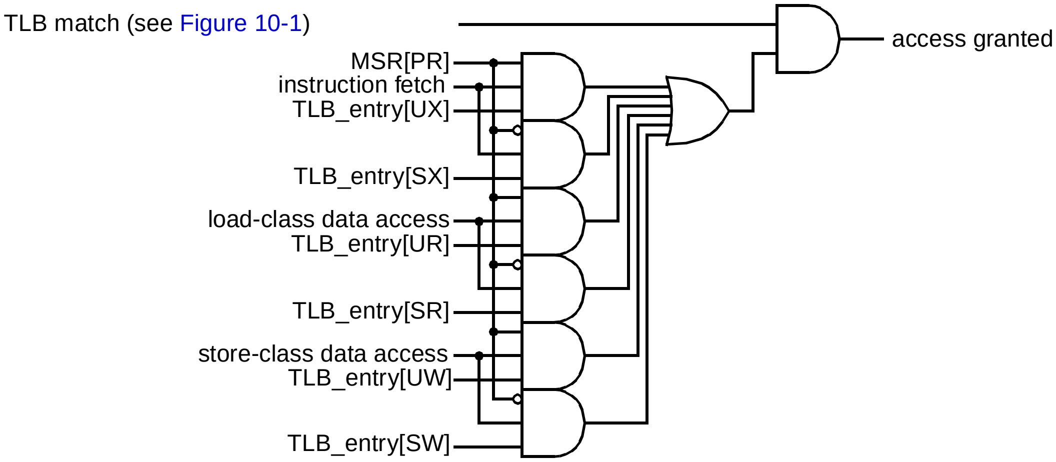


Figure 10-3. Granting of Access Permission

10.3 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, and so on. 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 e200z4 MMU supports a TLB that is fully associative and supports variable page sizes; thus it corresponds to TLB1.

TLB1 consists of a 16-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. 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')

Table 10-2. TLB Entry Bit Definitions

Field	Comments
EPN[0–21]	Effective page number (compared against effective address)
RPN[0–21]	Real page number (translated address)
SIZE[0–4]	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) is a 32-bit read-only register. The SPR number for MMUCFG is 1015 in decimal. MMUCFG provides information about the configuration of the e200z4 MMU design.

The MMUCFG register is shown in Figure 10-4.

0	RASIZE	0	NPIDS	PIDSIZE	0	NTLBS	MAVN
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

SPR - 1015; Read-Only

Figure 10-4. MMU Configuration Register (MMUCFG)

The MMUCFG bits are described in Table 10-3.

Table 10-3. MMUCFG Field Descriptions

Bits	Name	Function
0–7 [32–39]	—	Reserved[1]
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[1]
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[1]
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

[1] These bits are not implemented and will be read as zero.

10.4.2 TLB0 Configuration Register (TLB0CFG)

The TLB0 configuration register (TLB0CFG) is a 32-bit read-only register. The SPR number for TLB0CFG is 688 in decimal. TLB0CFG provides information about the configuration of TLB0. Because the e200z4 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.

ASSOC								MINSIZE				MAXSIZE				IPROT	AVAIL	P2PSA	0	NENTRY											
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

SPR - 688; Read-Only

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
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. The SPR number for TLB1CFG is 689 in decimal. TLB1CFG provides information about the configuration of TLB1 in the e200z4 MMU.

The TLB1CFG register is shown in Figure 10-6.

ASSOC								MINSIZE				MAXSIZE				IPROT	AVAIL	P2PSA	0	NENTRY											
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

SPR - 689; Read-Only

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 0x10 Indicates that TLB1 associativity is 16
8–11 [40–43]	MINSIZE	Minimum Page Size 0x0 Smallest page size is 1 Kbyte
12–15 [44–47]	MAXSIZE	Maximum Page Size 0xB Largest page size is 4 Gbytes
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
18 [50]	P2PSA	Power-of-2 Page Size Availability 1 All odd powers of 2 page sizes between MINSIZE and MAXSIZE are supported (2 Kbytes, 8 Kbytes, 32 Kbytes, etc.)
19 [51]	—	Reserved[1]
20–31 [52–63]	NENTRY	Number of Entries 0x10 TLB1 contains 16 entries

[1] These bits are not implemented and will be read as zero.

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), 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 the appropriate MSR[IS] or MSR[DS] values that 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 which entry is related to the DSI or ISI exception. Note that by the time the search occurs, it is possible that the relevant entry no longer exists in the TLB 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

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

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 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 that 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 that 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]

Table 10-6. tlbivax EA Bit Definitions

Bits	Field
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.

tlbsync tlbsync

TLB Synchronize

tlbsync

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

10.6 TLB Operations

This section discusses the following operations

- 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 must be 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 will be 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 only has to update MAS3 by **mtspr** before executing **tlbwe**. If the defaults are not applicable to the TLB entry being loaded, the TLB miss exception handler has to 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, since 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 the following table:

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 13.3.2.5, “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 13.3.2.5, “Reset Base (p_rstbase[0:29])”
SIZE[0–4]	00010	4 Kbyte 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

Table 10-7. TLB Entry 0 Values After Reset (Continued)

Field	Reset Value	Comments
E	value of *p_rst_endmode*	Value present on *p_rst_endmode*. See Section 13.3.2.6, “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*. See Section 13.3.2.7, “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 until the exception status of the instruction(s) in progress is known. If G = 0 (unguarded), then these accesses may be issued to the bus regardless of the completion status of other instructions. Because the e200z4 does not make requests to the bus for load or store instructions until it is known that prior instructions will complete without exceptions, proper operation will always occur to guarded storage.

10.7 MMU Control Registers

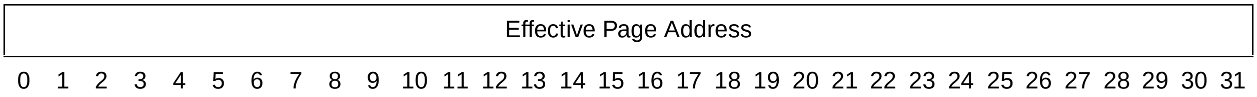
This section discusses the following registers:

- Section 10.7.1, “DEAR Register”
- Section 10.7.2, “MMU Control and Status Register 0 (MMUCSR0)”
- Section 10.7.3, “MMU Assist Registers (MAS)”
- Section 10.7.4, “MAS Registers Summary”
- Section 10.7.5, “MAS Register Updates”

10.7.1 DEAR Register

The data exception address register (DEAR) is loaded with the effective address of the data access that results in an alignment, data TLB miss, or DSI exception. The DEAR can be read or written using the **mfspr** and **mtspr** instructions.

The DEAR register is shown in Figure 10-7.

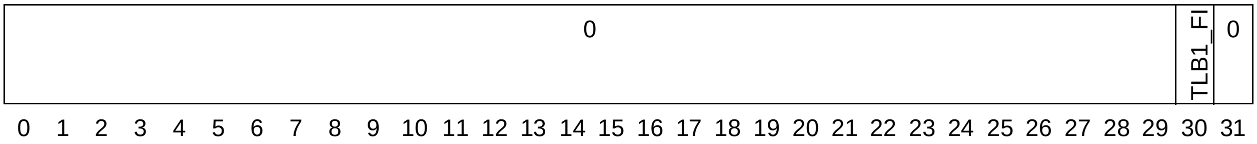


SPR - 61; Read/ Write; Reset - Unaffected

Figure 10-7. DEAR

10.7.2 MMU Control and Status Register 0 (MMUCSR0)

The MMU control and status register 0 (MMUCSR0) is a 32-bit register. The SPR number for MMUCSR0 is 1012 in decimal. MMUCSR0 controls the state of the MMU. The MMUCSR0 register is shown in Figure 10-8.



SPR - 1012; Read/ Write; Reset - 0x0

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[1]
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 results 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[1]

[1] These bits are not implemented; they are read as zero, and writes are ignored.

10.7.3 MMU Assist Registers (MAS)

The e200z4 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 e200z4 does not implement the MAS5 register, present in other Freescale EIS designs, because the **tlbsx** instruction only searches based on a single SPID value.

The MAS0 register is shown in Figure 10-9.

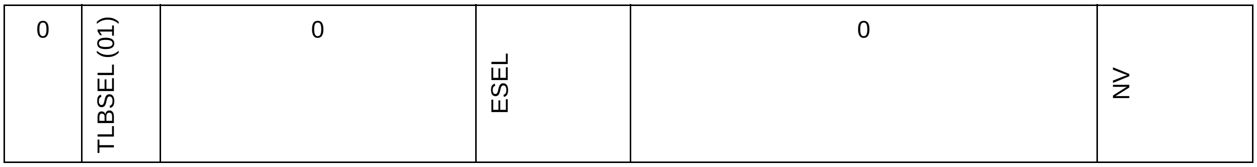


Figure 10-9. MMU Assist Register 0 (MAS0)

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

SPR - 624; Read/ Write; Reset - Unaffected

Figure 10-9. MMU Assist Register 0 (MAS0)

The MAS0 fields are defined in Table 10-9.

Table 10-9. MAS0 —MMU Read/Write and Replacement Control

Bit	Name	Comments, or Function when Set
0–1 [32–33]	—	Reserved[1]
2–3 [34–35]	TLBSEL	Selects TLB for access: 00 = TLB0, 01 = TLB (Ignored by the e200, should be written to 01 for future compatibility)
4–11 [36–43]	—	Reserved[1]
12–15 [44–47]	ESEL	Entry select for TLB.
16–27 [48–59]	—	Reserved[1]
28–31 [60–63]	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).

[1] These bits are not implemented; they are read as zero, and writes are ignored.

The MAS1 register is shown in Figure 10-10.

<table>
<tr><td>VALID</td><td>IPROT</td><td colspan="6">0</td><td colspan="8">TID</td><td colspan="3">0</td><td>TS</td><td colspan="5">TSIZ</td><td colspan="7">0</td></tr>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td><td>30</td><td>31</td></tr>
</table>

SPR - 625; Read/ Write; Reset - Unaffected

Figure 10-10. MMU Assist Register 1 (MAS1)

The MAS1 fields are defined in Table 10-10.

Table 10-10. MAS1—Descriptor Context and Configuration Control

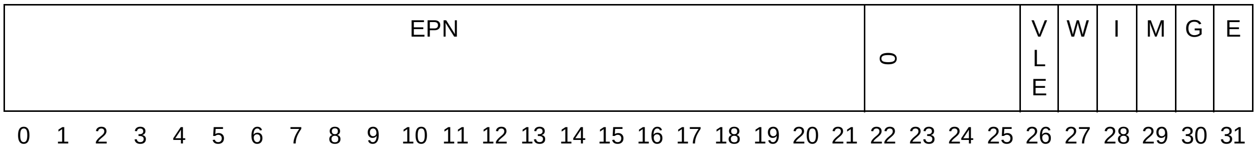
Bit	Name	Comments, or Function when Set
0 [32]	VALID	TLB Entry Valid 0 This TLB entry is invalid 1 This TLB entry is valid
1 [33]	IPROT	Invalidation Protect 0 Entry is not protected from invalidation 1 Entry is protected from invalidation as described in Section 10.6.6, “IPROT Invalidation Protection.” Protects TLB entry from invalidation by **tlbivax** (TLB1 only), or flash invalidates through MMUSCR0[TLB1_FI].

Table 10-10. MAS1—Descriptor Context and Configuration Control (Continued)

Bit	Name	Comments, or Function when Set
2–7 [34–39]	—	Reserved[1]
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[1]
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 Kbyte 0b00001 2 Kbytes 0b00010 4 Kbytes 0b00011 8 Kbytes 0b00100 16 Kbytes 0b00101 32 Kbytes 0b00110 64 Kbytes 0b00111 128 Kbytes 0b01000 256 Kbytes 0b01001 512 Kbytes 0b01010 1 Mbyte 0b01011 2 Mbytes 0b01100 4 Mbytes 0b01101 8 Mbytes 0b01110 16 Mbytes 0b01111 32 Mbytes 0b10000 64 Mbytes 0b10001 128 Mbytes 0b10010 256 Mbytes 0b10011 512 Mbytes 0b10100 1 Gbyte 0b10101 2 Gbytes 0b10110 4 Gbytes All other values are undefined
25–31 [57–63]	—	Reserved[1]

[1] These bits are not implemented; they are read as zero, and writes are ignored.

The MAS2 register is shown in Figure 10-11.



SPR - 626; Read/ Write; Reset - Unaffected

Figure 10-11. MMU Assist Register 2 (MAS2)

The MAS2 fields are defined in Table 10-11.

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 Access to this page are not guarded, and can be performed before it is known if they are required by the sequential execution model 1 All loads and stores to this page are performed without speculation (that is, they are known to be required) The e200z4 uses the guarded attribute as described in Section 9.13, "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 12.2.4, "Byte Lane Specification," for more information

[1] These bits are not implemented; they are read as zero, and writes are ignored.

The MAS3 register is shown in Figure 10-12.

RPN	U0	U1	U2	U3	UX	SX	UW	SW	UR	SR
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

SPR - 627; Read/ Write; Reset - Unaffected

Figure 10-12. MMU Assist Register 3 (MAS3)

The MAS3 fields are defined in Table 10-12.

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.

0	TLBSELD (01)	0	TIDSELD	0	TSIZED	0	VLED	WD	ID	MD	GD	ED
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

SPR - 628; Read/ Write; Reset - Unaffected

Figure 10-13. MMU Assist Register 4 (MAS4)

The MAS4 fields are defined in Table 10-13.

Table 10-13. MAS4—Hardware Replacement Assist Configuration Register

Bit	Name	Comments, or Function when Set
0–1 [32–33]	—	Reserved[1]
2–3 [34–35]	TLBSELD	Default TLB selected 00 TLB0 01 TLB1
4–13 [36–45]	—	Reserved[1]

Table 10-13. MAS4—Hardware Replacement Assist Configuration Register (Continued)

Bit	Name	Comments, or Function when Set
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 (8’h00)) (Use all zeros, the globally shared value)
16–19 [48–51]	—	Reserved[1]
20–24 [52–56]	TSIZED	Default TSIZE value
25 [57]	—	Reserved[1]
26 [58]	VLED	Default VLE value
27–31 [59–63]	DWIMGE	Default WIMGE values

[1] These bits are not implemented; they are read as zero, and writes are ignored.

The MAS6 register is shown in Figure 10-14.

0								SPID								0															SAS
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

SPR - 630; Read/ Write; Reset - Unaffected

Figure 10-14. MMU Assist Register 6 (MAS6)

The MAS6 fields are defined in Table 10-14.

Table 10-14. MAS6—TLB Search Context Register 0

Bit	Name	Comments, or Function when Set
0–7 [32–39]	—	Reserved[1]
8–15 [40–47]	SPID	PID value for searches
16–30 [48–62]	—	Reserved[1]
31 [63]	SAS	AS value for searches

[1] These bits are not implemented; they are read as zero, and writes are ignored.

10.7.4 MAS Registers Summary

The MAS registers are summarized in Figure 10-15.

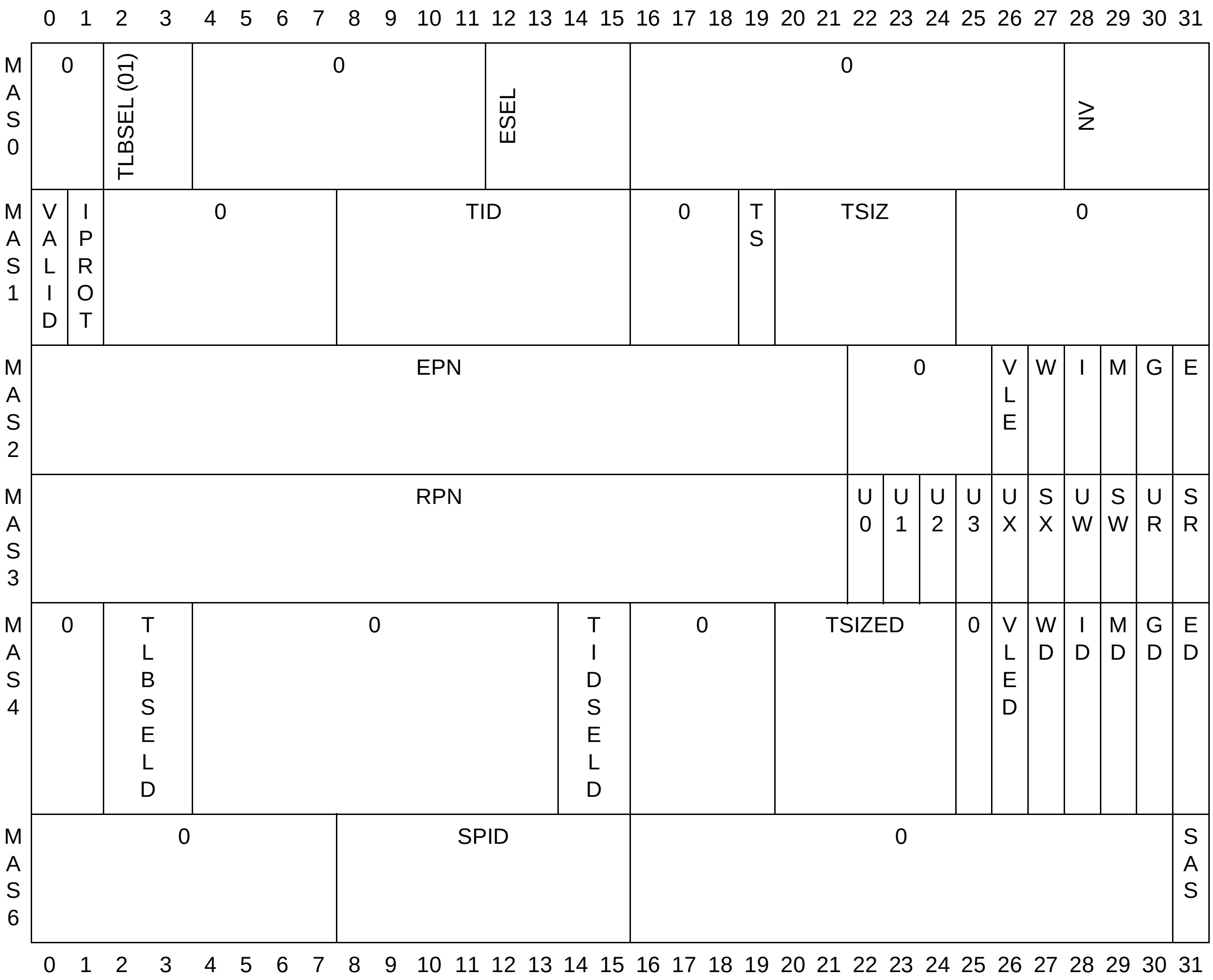


Figure 10-15. MMU Assist Registers Summary

10.7.5 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

Table 10-15. MMU Assist Register Field Updates (Continued)

Bit/Field	MAS affected	Instr/Data TLB Error	tlbsx hit	tlbsx miss	tlbre	tlbwe	ISI/DSI
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 provides the ability to invalidate 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 because 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[0:4]* encodings.

Table 10-16. Transfer Type Encoding

p_d_ttype[0:4]	Transfer Type	Instruction
00000	Normal	Normal loads/stores
00001	Atomic	**lbarx**, **lharx**, **lwarx**, **stbcx.**, **sthcx.**, and **stwcx.**
00010	Reserved for Flush Data Block	Reserved for **dcbst**
00011	Reserved for Flush and Invalidate Data Block	Reserved for **dcbf**
00100	Reserved for Allocate and Zero Data Block	Reserved for **dcbz**
00101	Reserved for Invalidate Data Block	Reserved for **dcbi**
00110	Invalidate Instruction Block	**icbi**
00111	multiple word load/store	**lmw**, **stmw**
01000	TLB Invalidate	**tlbivax**
01001	TLB Search	**tlbsx**
01010	TLB Read entry	**tlbre**
01011	TLB Write entry	**tlbwe**
01100	Touch for Instruction	**icbt**
01101	Lock Clear for Instruction	**icblc**
01110	Touch for Instruction and Lock Set	**icbtls**
01111	Reserved for Lock Clear for Data	Reserved for **dcblc**
10000	Reserved for Touch for Data	Reserved for **dcbt**
10001	Reserved for Touch for Data Store	Reserved for **dcbtst**
10010	Reserved for Touch for Data and Lock Set	Reserved for **dcbtls**
10011	Reserved for Touch for Data Store and Lock Set	Reserved for **dcbtstls**

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 substitute default values for the access protection (UX, UR, UW, SX, SR, SW) bits as well as the values obtained from the OnCE control register for page attribute (VLE, W, I, M, G, E) bits that are normally provided by a matching TLB entry. In addition, no address translation is performed. 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 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 DSI) will remain in effect. Refer to Section 11.4.6.3, “e200 OnCE Control Register (OCR) for more detail on controlling MMU operation during debug sessions.

10.11 External Translation Alterations for Real-time Systems

In order to support real-time 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 to which a given logical page maps can be dynamically altered by means of the *p_extpid[6:7]* inputs. This capability is provided for TLB1 entries 0–15 for a restricted subset of PID values.

Enabling of the dynamic mapping capability is controlled by the *p_extpid_en* control input. This input is sampled with the rising edge of the clock, and when asserted, allows for use of the dynamic remapping capability.

When one or more of TLB1 entries 0–15 is programmed with a TID value of 8‘b1111xxxx, 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 TLB entries 0–15 that are programmed with a TID value of 8‘b1111xx11, 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.

TLB entries 0–15 that are programmed with a TID value of 8‘b1111nm00 match a PID0 value of 8‘b1111nmxx when *p_extpid[6:7]* inputs are 00. Those programmed with a TID value of 8‘b1111nm01 match a PID0 value of 8‘b1111nmxx when *p_extpid[6:7]* inputs are 01, and those programmed with a TID value of 8‘b1111nm10 will match a PID0 value of 8‘b1111nmxx when *p_extpid[6:7]* inputs are 10. TLB entries 0–15 programmed with a TID value of 8‘b1111nm11 match a PID0 value of 8‘b1111nmxx 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 will 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 10-16 shows the equivalent logical operation of the capability.

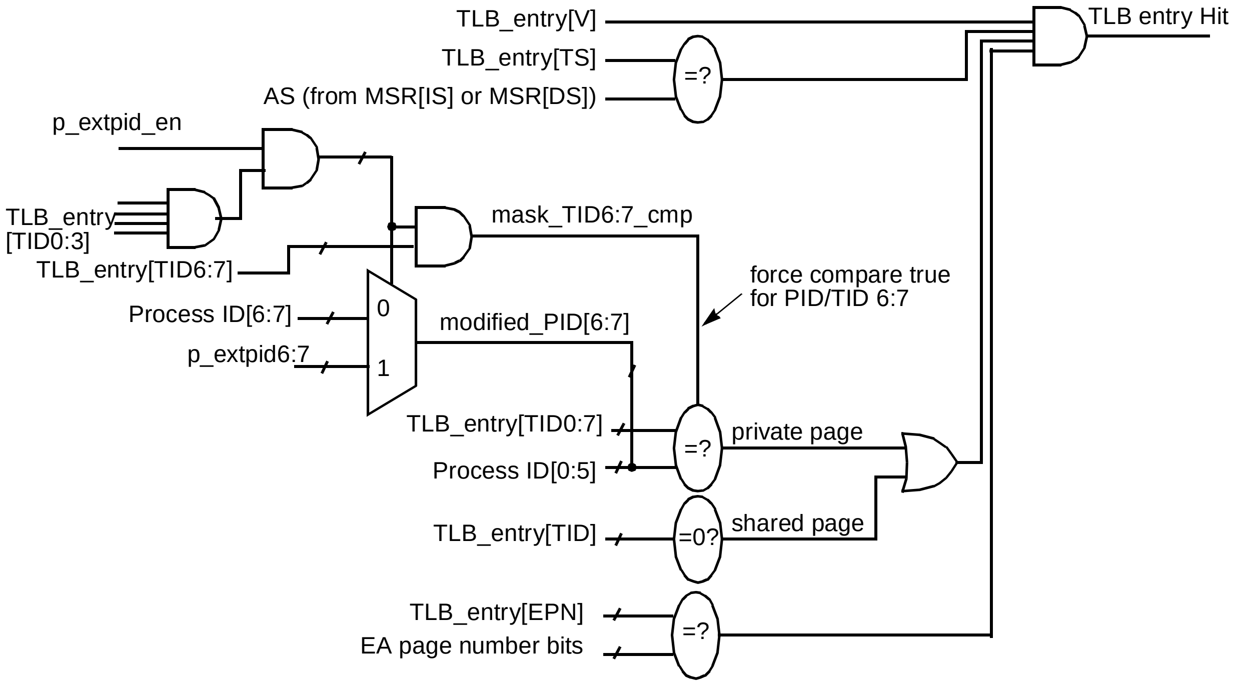


Figure 10-16. External Translation Alteration TLB Entry Compare Process

Chapter 11
Debug Support

This chapter describes the debug features of the e200z446n3 core.

11.1 Overview

Internal debug support in the e200z446n3 core allows for software and hardware debug by providing debug functions, such as instruction and data breakpoints and program trace modes. A set of software accessible debug registers and interrupt mechanisms are provided for software based debugging. These facilities are also available to a hardware based debugger that 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. The e200z446n3 supports a subset of these defined facilities. In addition to the facilities defined in the Power ISA, the core 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 core also provides support for run-time integrity checking by a parallel signature unit, which is capable of monitoring the CPU data read and data write AHB buses, and accumulating a pair of 32-bit MISR signatures of the data values transferred over these buses.

11.1.1 Software Debug Facilities

The e200z446n3 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 a set of debug control registers (DBCR0–6, DBERC0), a set of address compare registers (IAC1–8, DAC1, and DAC2), a set of data value compare registers (DVC1, DVC2), a configurable debug counter, a debug status register (DBSR) for enabling and recording various kinds of debug events, and a special debug interrupt type built into the interrupt mechanism (see Section 5.7.16, “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 can 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] is cleared (and DBCR0[EDM] is cleared as well), no debug events occur, and no status flags are set in DBSR unless already set. In addition, 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.

11.1.1.1 Power ISA Embedded Category Compatibility

The e200z446n3 core implements a subset of the 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.

11.1.2 Additional Debug Facilities

In addition to the debug category defined in the Power ISA, the e200z446n3 provides the capability to link instruction and data breakpoints, a configurable debug event counter to allow debug exception generation capability, and a sequential breakpoint control mechanism.

The core also defines two new debug events (CIRPT, CRET) for debugging around critical interrupts.

In addition, the e200z446n3 implements the debug instruction set. When enabled, this 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** instruction.

The e200 also provides the capability for sharing resources between hardware and software debuggers. See Section 11.1.4, “Sharing Debug Resources by Software/Hardware.”

11.1.3 Hardware Debug Facilities

The e200z446n3 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 IEEE 1149.1 TAP (test access port) controller.

By using public instructions, the external debugger can freeze or halt the 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 by the settings in DBERC0. When the hardware debug facility is enabled, software is blocked from modifying the “hardware-owned” debug facilities. In addition, since resources are “owned” by the hardware debugger, inconsistent values may be present if software attempts to read “hardware-owned” debug-related resources.

When hardware debug is enabled by setting [E]DBCR0[EDM] = 1, the control registers and resources described in Section 11.3, “Debug Registers,” are reserved for use by the external debugger. The same events described in Section 11.2, “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 the EDBSR0 register (not the DBSR) regardless of

MSR[DE], and no debug interrupts are generated unless the resource is granted back to software by DBERC0 settings. Instead, the CPU enters debug mode when an enabled event causes a EDBSR0 bit to become set. DBCR0[EDM], EDBSR0, and DBERC0 may only be written through the OnCE port.

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.

11.1.4 Sharing Debug Resources by Software/Hardware

Debug resources may be shared by a hardware debugger and software debug based on the settings of debug control register DBERC0. 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 [E]DBCR0[EDM] = 1 cause an entry into debug mode. DBERC0 is read-only by software. When resource sharing is enabled (DBCR0[EDM] = 1 and DBERC0[IDM] = 1), only software-owned resources may be modified by software. 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 with read-modify-write operations to implement any control sharing with software. Hardware-owned resources set status bits in the EDBSR0 register 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 is performed.

11.1.4.1 Simultaneous Hardware and Software Debug Event Handing

Since it is possible for a hardware-owned resource to produce a debug event at the same time as a software-owned resource produces a different debug event, a priority ordering mechanism is implemented to guarantee that the hardware event is handled as soon as possible while recognition of the software event is preserved. Initially, the CPU gives the highest priority to the software event in order to reach a recoverable boundary. It then gives highest priority to the hardware event in order to enter debug mode as near as possible to the point of event occurrence. This is implemented by allowing software exception handling to begin internal to the CPU and to reach the point where the current program counter and MSR values have been saved into DSRR0/1, and the new PC pointing 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 11-1 shows the e200z446n3 debug resources.

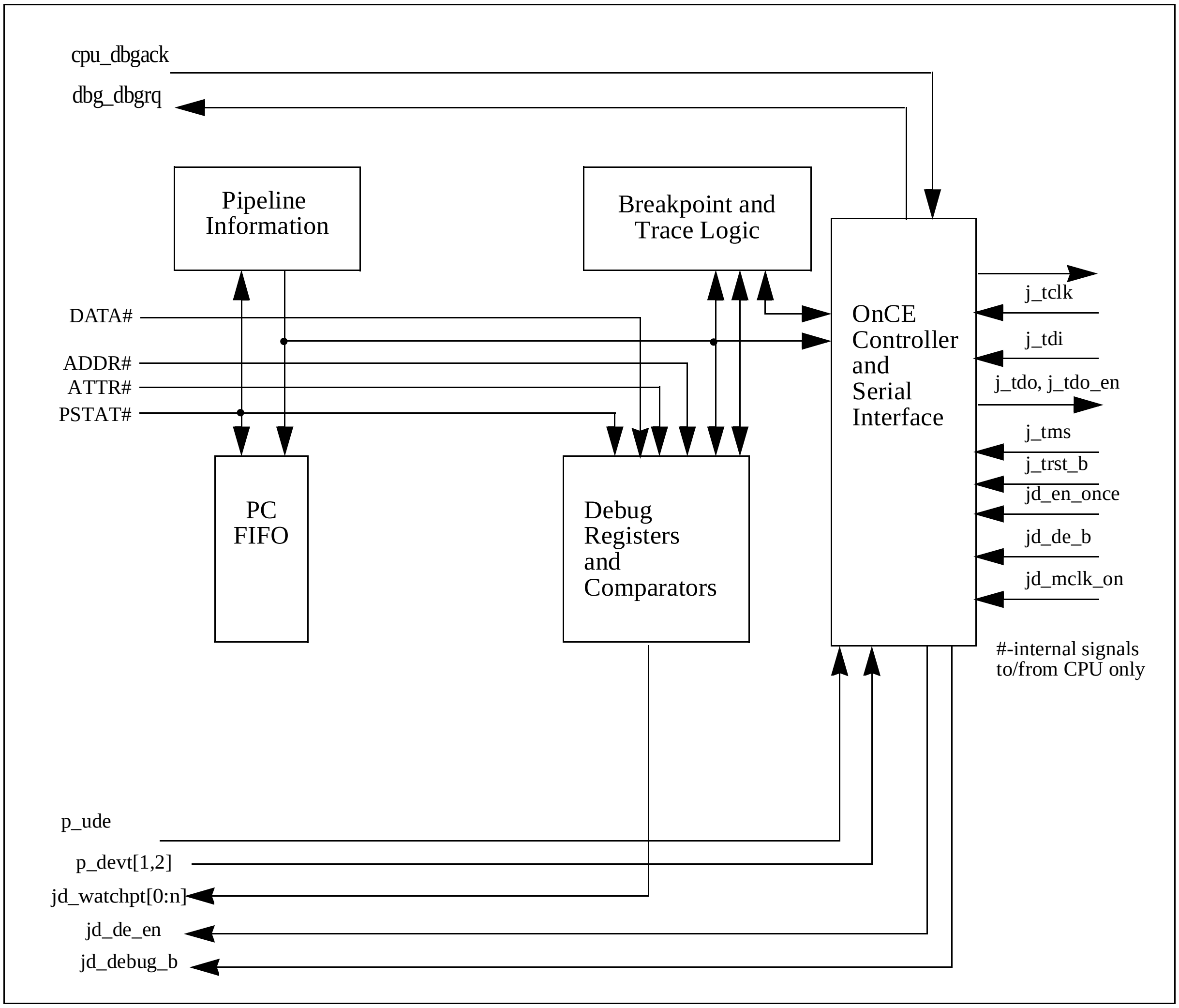


Figure 11-1. Debug Resources

11.2 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. Once a debug status register (DBSR) bit is set by a debug resource that is owned by software (other than MRR and CNT1TRG), a debug interrupt is generated 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 will be delayed until MSR[DE] is later set to ‘1’.

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 that enabled a delayed debug interrupt by setting the MSR[DE] bit. 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 the Power ISA:

- Instruction Address Compare debug events
- Data Address Compare debug events
- Trap debug events
- Branch Taken debug events
- Instruction Complete debug events
- Interrupt Taken debug events
- Return debug events
- Unconditional debug events

These events are described in detail in the *EREF*.

In addition, e200z446n3 defines additional debug events:

- The debug counter debug events DCNT1 and DCNT2, which are described in Section 11.2.11, “Debug Counter Debug Event.”
- The external debug events DEVT1 and DEVT2, which are described in Section 11.2.12, “External Debug Event.”
- The critical interrupt taken debug event CIRPT, which is described in Section 11.2.8, “Critical Interrupt Taken Debug Event.”
- The critical return debug event CRET, which is described in Section 11.2.10, “Critical Return Debug Event.”

The e200z446n3 debug configuration supports most of these event types. However, instruction address compare and data address compare *real address* mode is not supported.

A brief description of each of the event types follows. In these descriptions, DSRR0 and DSRR1 are used, assuming that the debug instruction set is enabled. If it is disabled, use CSRR0 and CSRR1 respectively.

11.2.1 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 is marked with a set of bits indicating whether an instruction address compare occurred on that half word. Debug exceptions occur if enabled and 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.

11.2.2 Data Address Compare Event

When the debug instruction set is enabled, data address compare debug events occur when execution of a load or store class instruction results in a data access meeting 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.

NOTES

In contrast to the the Power ISA definition, data address compare events on the e200z446n3 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 instruction set 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, and a data address compare debug exception also occurs or causes a debug counter event to occur, the result is an imprecise debug interrupt. The address value saved in DSRR0 (or CSRR0 if the debug instruction setis disabled) is the address of the load or store class instruction, and the DBSR[IDE] bit 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 **lmw** or **stmw** 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.

11.2.2.1 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 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 that 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 11-1 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 11-1, the DAC_OFST and DSRR0 values assume no dual issue occurs.

If dual-issue occurs with the first, second, or third column, then the DAC_OFST and DSRR0 values will point beyond the values shown.

Table 11-1. 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 3’b001. 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 3’b010. 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.

Table 11-1. 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	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.
DVC DACx	DACy	—	Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 3’b001. 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 3’b001. 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 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, Normal Ldst	DSI Error, with or without DAC	Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 3’b001. 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 3’b001. 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 11-1. 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 11-2–Table 11-5 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 11-2 shows the DAC events and resultant updates for dual-issue case 1.

Table 11-2. DAC Events and Resultant Updates, Dual-Issue Case 1

Event(s)	Result
Instruction Sequence: The following pairs dual-issue: (1) load/store and (2) **alu**, (3) load/store and (4) **alu**, (5) load/store and (6) **alu**	
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 3’b000. 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 3’b000. 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 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)

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 3’b010. 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 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: In this case if x==y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the “no DACy” case.

Table 11-2. DAC Events and Resultant Updates, Dual-Issue Case 1 (Continued)

Event(s)	Result
Instruction Sequence: The following pairs dual-issue: (1) load/store and (2) **alu**, (3) load/store and (4) **alu**, (5) load/store and (6) **alu**	

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 3’b010. 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 3’b100. 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 11-3 shows the DAC events and resultant updates for dual-issue case 2.

Table 11-3. DAC Events and Resultant Updates, Dual-Issue Case 2

Event(s)	Result
Instruction Sequence: The following pairs dual-issue: (1) load/store and (2) **alu,** (3) load/store and (4) **alu,** (5) **alu** and (6) load/store	
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 3’b000. 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 3’b000. 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 3’b101. DSRR0 points to instruction after instruction (6). No MASx register update. No ESR update.Debug counter update occurs for instructions (1)–(6) as appropriate.

Table 11-3. DAC Events and Resultant Updates, Dual-Issue Case 2 (Continued)

Event(s)	Result
Instruction Sequence: The following pairs dual-issue: (1) load/store and (2) **alu,** (3) load/store and (4) **alu,** (5) **alu** and (6) load/store	

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 3’b010. 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 3’b101. 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.

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 3’b010. 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 3’b101. 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 11-4 shows the DAC events and resultant updates, dual-issue case 3.

Table 11-4. DAC events and Resultant Updates, Dual-issue case 3

Event(s)	Result
Instruction Sequence: The following pairs dual-issue: (1) load/store and (2) **alu**, (3) **alu** and (4) **alu**, (5) load/store and (6) **alu**	
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 3’b000. 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 3’b000. 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 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).

Instruction (1):
DVC DACx
Instruction (5):
DTLB Error, with or without DAC
	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 3’b011. 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 3’b100. 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 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 if x==y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the “no DACy” case.

Table 11-5 shows the DAC events and resultant updates for dual-issue case 4.

Table 11-5. DAC Events and Resultant Updates, Dual-Issue Case 4

Event(s)	Result
Instruction Sequence: The following pairs dual-issue: (1) load/store and (2) **alu**, (3) load/store and (4) **alu**, (5) **alu** and (6) **alu**	
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 3'b000. 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 3'b000. 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 3'b011. 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 3'b010. 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 3'b011. 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.

11.2.3 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 that 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 the IAC1 (or IAC3) event enable is set.

A linked data address compare debug event occurs when the debug instruction set is enabled, 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 meets the criteria for generating an instruction address compare event. 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 are not 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.

11.2.4 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, the DBSR[TRAP] bit is set to 1 to record the debug exception.

11.2.5 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, the DBSR[BRT] bit is set to 1 to record the debug exception, and the address of the branch instruction is recorded in DSRR0.

11.2.6 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 because the instruction causes a different exception that 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 1 to record the debug exception and the address of the next instruction to be executed is 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, thus DBSR[IDE] is not generally set by an ICMP debug event.

One circumstance may cause the DBSR[ICMP] and DBSR[IDE] bits to be set. This occurs when a EFPU round exception occurs. Since 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, and DBSR[IDE] is set to indicate the imprecise recognition of a debug interrupt. In this case, the debug interrupt will be taken with SRR0 pointing to the instruction following the instruction that generated the EFPU round exception, and DSRR0 will point 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] to 1 while DBCR0[ICMP] = 1, nor by the execution of an instruction that sets DBCR0[ICMP] to 1 while MSR[DE] = 1 or DBCR0[EDM] = 1.

11.2.7 Interrupt Taken Debug Event

An interrupt taken debug event (IRPT) occurs if interrupt taken debug events are enabled (DBCR0[IRPT] = 1) and a non-critical interrupt occurs. Only non-critical 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, the DBSR[IRPT] bit is set to 1 to record the debug exception. The value saved in DSRR0 will be the address of the non-critical interrupt handler.

11.2.8 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 instruction set 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, the DBSR[CIRPT] bit is set to 1 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 instruction set is also enabled to avoid corruption of CSRR0/1.

11.2.9 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, the DBSR[RET] bit is set to 1 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** (before the MSR is updated by the **rfi** or **se_rfi**), then DBSR[IDE] is also set to 1 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 no higher priority exception is enabled to cause an interrupt. The debug save/restore register 0 is set to the address of the **rfi** or **se_rfi** instruction.

11.2.10 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, the DBSR[CRET] bit is set to 1 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** (before the MSR is updated by the **rfci** or **se_rfci**), then DBSR[IDE] is also set to 1 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. The debug save/restore register 0 will be set to the address of the **rfci** or **se_rfci** instruction. Note that this debug event should not normally be enabled unless the debug instruction set is also enabled to avoid corruption of CSRR0/1.

11.2.11 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 ‘1’ to record the debug exception.

11.2.12 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 ‘1’ to record the debug exception.

11.2.13 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 ‘1’ to record the debug exception.

11.3 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. The 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 an **mfspr** instruction, other than 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, 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 registers as appropriate when hardware modifications to the debug registers is performed.

11.3.1 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 11-2) 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 11-2) 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 13-12 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 because the data address and data value comparisons are only performed on the initial access in the case of a misaligned access; thus, accesses that 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 the MSR[SPE] bit.

B0	B1	B2	B3
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
B4	**B5**	**B6**	**B7**
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

SPR - 318 (DVC1), 319 (DVC2); Read/Write; Reset - Unaffected

Figure 11-2. DVC1, DVC2 Registers

11.3.2 Debug Counter Register (DBCNT)

The debug counter register (DBCNT) contains two 16-bit counters (CNT1 and CNT2) that 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. The DBCNT register is shown in Figure 11-3.

CNT1	CNT2
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

SPR - 562; Read/Write; Reset - Unaffected

Figure 11-3. DBCNT Register

Refer to Section 11.3.3.4, “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.

11.3.3 Debug Control and Status Registers

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 debug mode is enabled.

e200z446n3 requires that a context synchronizing instruction follow an **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 (by DBCR0) from setting DBSR flags.

11.3.3.1 Debug Control Register 0 (DBCR0)

Debug control register 0 is used to enable debug modes and controls which debug events are allowed to set DBSR or EDBSR0 flags. The e200z446n3 adds some implementation specific bits to this register, as seen in Figure 11-4.

EDM	IDM	RST		ICMP	BRT	IRPT	TRAP	IAC1	IAC2	IAC3	IAC4	DAC1		DAC2		RET	IAC5	IAC6	IAC7	IAC8	DEVT1	DEVT2	DCNT1	DCNT2	CIRPT	CRET	0				FT
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

SPR - 308; Read/Write; Reset[1] - 0x0

Figure 11-4. 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 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**. The DBCR0$_{RST}$ field will always be reset by **p_reset_b** regardless of the value of DBCR0$_{EDM}$.

Table 11-6 provides bit definitions for debug control register 0.

Table 11-6. 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. 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} unless permitted by settings in DBERC0. Hardware-owned events will set status bits in EDBSR0. **Note:** 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 to ‘1’ before other bits in this register (and other debug registers) may be altered. On the initial setting of this bit to ‘1’, all other bits are unchanged. This bit is only writable through the OnCE port.

Table 11-6. DBCR0 Bit Definitions (Continued)

Bit(s)	Name	Description
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 will 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, will cause 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
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

Table 11-6. DBCR0 Bit Definitions (Continued)

Bit(s)	Name	Description
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
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 TimeBase Timers are unaffected by set DBSR/EDBSR0 bits 1 Disable clocking of TimeBase timers if any DBSR bit is set (any EDBSR0 bit set if DBCR0$_{FT}$ owned by hardware) except MRR or CNT1TRG

11.3.3.2 Debug Control Register 1 (DBCR1)

Debug control register 1 is used to configure the instruction address compare operation. The DBCR1 register is shown in Figure 11-5.

IAC1US	IAC1ER	IAC2US	IAC2ER	IAC12M	0	IAC3US	IAC3ER	IAC4US	IAC4ER	IAC34M	0
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

SPR - 309; Read/Write; Reset[1] - 0x0

Figure 11-5. DBCR1 Register

[1] Reset by processor reset **p_reset_b** if $\text{DBCR0}_{\text{EDM}}$=0, as well as unconditionally by **m_por**. If $\text{DBCR0}_{\text{EDM}}$=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 11-7 provides bit definitions for the debug control register 1.

Table 11-7. DBCR1 Bit Definitions

Bit(s)	Name	Description
0–1	IAC1US	Instruction Address Compare 1 User/Supervisor Mode 00 IAC1 debug events 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 the e200 (Power ISA 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
4–5	IAC2US	Instruction Address Compare 2 User/Supervisor Mode 00 IAC2 debug events 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 (Power ISA 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

Table 11-7. DBCR1 Bit Definitions (Continued)

Bit(s)	Name	Description
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 (Power ISA 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 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 (Power ISA 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
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

11.3.3.3 Debug Control Register 2 (DBCR2)

Debug control register 2 is used to configure Data Address Compare and Data Value Compare operation. The DBCR2 register is shown in Figure 11-6.

DAC1US	DAC1ER	DAC2US	DAC2ER	DAC12M	DAC1LNK	DAC2LNK	DVC1M	DVC2M	DVC1BE	DVC2BE
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

SPR - 310; Read/Write; Reset[1] - 0x0

Figure 11-6. 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 will be reset by *p_reset_b*.

Table 11-8 provides bit definitions for debug control register 2.

Table 11-8. DBCR2 Bit Definitions

Bit(s)	Name	Description
0–1	DAC1US	Data Address Compare 1 User/Supervisor Mode 00 DAC1 debug events 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 e200 (Book E 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
4–5	DAC2US	Data Address Compare 2 User/Supervisor Mode. 00 DAC2 debug events 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 e200 (Book E 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

Table 11-8. DBCR2 Bit Definitions (Continued)

Bit(s)	Name	Description
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* since DAC2 debug events are not generated in the other compare modes.
12–13	DVC1M	Data Value Compare 1 Mode When DBCR4[DVC1C]=0– 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– 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.

Table 11-8. DBCR2 Bit Definitions (Continued)

Bit(s)	Name	Description
14–15	DVC2M	Data Value Compare 2 Mode When DBCR4[DVC2C]=0– 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– 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.
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. 1xxxxxxx Byte lane 0 is enabled for comparison with the value in bits 0–7 of DVC1. x1xxxxxx Byte lane 1 is enabled for comparison with the value in bits 8–15 of DVC1. xx1xxxxx Byte lane 2 is enabled for comparison with the value in bits 16–23 of DVC1. xxx1xxxx Byte lane 3 is enabled for comparison with the value in bits 24–31 of DVC1. xxxx1xxx Byte lane 4 is enabled for comparison with the value in bits 32–39 of DVC1. xxxxx1xx Byte lane 5 is enabled for comparison with the value in bits 40–47 of DVC1. xxxxxx1x Byte lane 6 is enabled for comparison with the value in bits 48–55 of DVC1. xxxxxxx1 Byte 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. 1xxxxxxx Byte lane 0 is enabled for comparison with the value in bits 0–7 of DVC2. x1xxxxxx Byte lane 1 is enabled for comparison with the value in bits 8–15 of DVC2. xx1xxxxx Byte lane 2 is enabled for comparison with the value in bits 16–23 of DVC2. xxx1xxxx Byte lane 3 is enabled for comparison with the value in bits 24–31 of DVC2. xxxx1xxx Byte lane 4 is enabled for comparison with the value in bits 32–39 of DVC2. xxxxx1xx Byte lane 5 is enabled for comparison with the value in bits 40–47 of DVC2. xxxxxx1x Byte lane 6 is enabled for comparison with the value in bits 48–55 of DVC2. xxxxxxx1 Byte lane 7 is enabled for comparison with the value in bits 56–63 of DVC2.

11.3.3.4 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 that 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 by 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 could 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 to ‘1’ and either DBCR0[IDM] or DBCR0[EDM] is set to 1. Counters are frozen during a hardware “debug session” (see Section 11.4.2, “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 is 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[CNT1TRG] 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, and other combinations are not supported:

- 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 the following combinations:

- 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.

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 **lmw** or **stmw** instruction, and no count will occur if either is interrupted by a critical input or external input interrupt prior to completion.

DBCR3 is a e200z446n3 implementation specific register and is shown in Figure 11-7.

DEVT1C1	DEVT2C1	ICMPC1	IAC1C1	IAC2C1	IAC3C1	IAC4C1	DAC1RC1	DAC1WC1	DAC2RC1	DAC2WC1	IRPTC1	RETC1	DEVT1C2	DEVT2C2	ICMPC2	IAC1C2	IAC2C2	IAC3C2	IAC4C2	DAC1RC2	DAC1WC2	DAC2RC2	DAC2WC2	DEVT1T1	DEVT2T1	IAC1T1	IAC3T1	DAC1RT1	DAC1WT1	CNT2T1	CONFIG
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

SPR - 561; Read/Write; Reset[1] - 0x0

Figure 11-7. DBCR3 Register

[1] Reset by processor reset **p_reset_b** if $\text{DBCR0}_{\text{EDM}}$=0, as well as unconditionally by **m_por**. If $\text{DBCR0}_{\text{EDM}}$=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 11-9 provides bit definitions for the debug control register 3.

Table 11-9. 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 that 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
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

Table 11-9. DBCR3 Bit Definitions (Continued)

Bit(s)	Name	Description
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 that 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
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

Table 11-9. DBCR3 Bit Definitions (Continued)

Bit(s)	Name	Description
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. For these cases, it is possible that the instruction that updates the counters or control over the counters will 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 a **mtspr** instruction that is loading the counter with a different value, a counter event will be generated following 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 a **mtspr** instruction that modifies a counter, regardless of whether a debug event is generated based on the old counter value.

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 a **mtspr** instruction that is loading 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 that 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.

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] bit.

11.3.3.5 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 11-8.

0	DVC1C	0	DVC2C	0												DAC1XM				DAC2XM				0							
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

SPR - 563; Read/Write; Reset[1] - 0x0

Figure 11-8. 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** and only software-owned resources indicated by DBERC0 will be reset by **p_reset_b**.

Table 11-10 provides bit definitions for debug control register 4.

Table 11-10. 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

Table 11-10. DBCR4 Bit Definitions (Continued)

Bit(s)	Name	Description
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 4 Kbytes are supported. 1101–1111 = Reserved DAC1XM allows for binary power of 2 address range compares for DAC1 without requiring the use of DAC2.
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 Kbytes 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

11.3.3.6 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 11-9.

IAC5US	IAC5ER	IAC6US	IAC6ER	IAC56M	0	IAC7US	IAC7ER	IAC8US	IAC8ER	IAC78M	0
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

SPR - 564; Read/Write; Reset[1] - 0x0

Figure 11-9. 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 11-7 provides bit definitions for debug control register 5.

Table 11-11. DBCR5 Bit Definitions

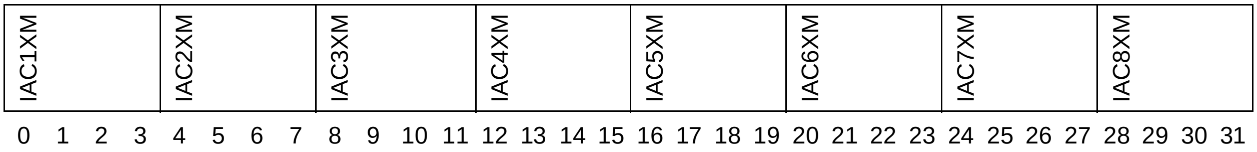
Bit(s)	Name	Description
0–1	IAC5US	Instruction Address Compare 5 User/Supervisor Mode 00 IAC5 debug events not affected by MSR_{PR} 01 Reserved 10 IAC5 debug events can only occur if MSR_{PR}=0 (Supervisor mode) 11 IAC5 debug events can only occur if MSR_{PR}=1. (User mode)
2–3	IAC5ER	Instruction Address Compare 5 Effective/Real Mode 00 IAC5 debug events are based on effective address 01 Unimplemented in e200 (Book E 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 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)
6–7	IAC6ER	Instruction Address Compare 6 Effective/Real Mode 00 IAC6 debug events are based on effective address 01 Unimplemented in e200 (Book E 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 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 (Book E 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 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)

Table 11-11. DBCR5 Bit Definitions (Continued)

Bit(s)	Name	Description
22–23	IAC8ER	Instruction Address Compare 8 Effective/Real Mode 00 IAC8 debug events are based on effective address 01 Unimplemented in e200 (Book E 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

11.3.3.7 Debug Control Register 6 (DBCR6)

Debug control register 6 is used to extend instruction address compare matching functionality. DBCR6 is shown in Figure 11-10.



IAC1XM				IAC2XM				IAC3XM				IAC4XM				IAC5XM				IAC6XM				IAC7XM				IAC8XM			
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

SPR - 603; Read/Write; Reset[1] - 0x0

Figure 11-10. 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 11-12 provides bit definitions for Debug Control Register 6.

Table 11-12. 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 Kbytes are supported. 1101–1111 Reserved IAC1XM allows for binary power of 2 address range compares for IAC1 without requiring the use of IAC2.

Table 11-12. DBCR6 Bit Definitions (Continued)

Bit(s)	Name	Description
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 Kbytes 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 Kbytes 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 Kbytes are supported. 1101–1111 = Reserved IAC4XM allows for binary power of 2 address range compares for IAC4 without requiring the use of IAC3.
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 Kbytes 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 Kbytes 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 Kbytes are supported. 1101 - 1111 = Reserved IAC7XM allows for binary power of 2 address range compares for IAC7 without requiring the use of IAC8.

Table 11-12. DBCR6 Bit Definitions (Continued)

Bit(s)	Name	Description
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 Kbytes are supported. 1101–1111 Reserved IAC8XM allows for binary power of 2 address range compares for IAC8 without requiring the use of IAC7.

11.3.3.8 Debug Status Register (DBSR)

The debug status register (DBSR) contains status on debug events and the most recent processor reset. The debug status register is set by hardware and read and cleared by software. 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 a 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) by DBERC0), a set bit in DBSR other than 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 are not set by hardware in DBSR.

The DBSR register is shown in Figure 11-11.

IDE	UDE	MRR		ICMP	BRT	IRPT	TRAP	IAC1	IAC2	IAC3	IAC4-8	DAC1R	DAC1W	DAC2R	DAC2W	RET	0				DEVT1	DEVT2	DCNT1	DCNT2	CIRPT	CRET	VLES	DAC_OFST			CNT1TRG
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

SPR - 304; Read/Write; Reset[1] - 0x1000_0000

Figure 11-11. DBSR 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.** DBSR$_{MRR}$ is always updated by **p_reset_b** however.

Table 11-13 provides bit definitions for the debug status register.

Table 11-13. DBSR Bit Definitions

Bit(s)	Name	Description
0	IDE	Imprecise Debug Event Set to 1 if MSR[DE] = 0 and DBCR0[IDM] = 1 and a debug event causes its respective Debug Status Register bit to be set to 1. It may also be set to ‘1’ 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 to 1 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 to 1 if an Instruction Complete debug event occurred.
5	BRT	Branch Taken Debug Event Set to 1 if an Branch Taken debug event occurred.
6	IRPT	Interrupt Taken Debug Event Set to 1 if an Interrupt Taken debug event occurred.
7	TRAP	Trap Taken Debug Event Set to 1 if a Trap Taken debug event occurred.
8	IAC1	Instruction Address Compare 1 Debug Event Set to 1 if an IAC1 debug event occurred.
9	IAC2	Instruction Address Compare 2 Debug Event Set to 1 if an IAC2 debug event occurred.
10	IAC3	Instruction Address Compare 3 Debug Event Set to 1 if an IAC3 debug event occurred.
11	IAC4-8	Instruction Address Compare 4–8 Debug Event Set to 1 if an IAC4, IAC5, IAC6, IAC7, or IAC8 debug event occurred.
12	DAC1R	Data Address Compare 1 Read Debug Event Set to 1 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 to 1 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 to 1 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 to 1 if a write-type DAC2 debug event occurred while DBCR0[DAC2] = 0b01 or DBCR0[DAC2] = 0b11
16	RET	Return Debug Event Set to 1 if a Return debug event occurred

Table 11-13. DBSR Bit Definitions (Continued)

Bit(s)	Name	Description
17–20	—	Reserved
21	DEVT1	External Debug Event 1 Debug Event Set to 1 if a DEVT1 debug event occurred
22	DEVT2	External Debug Event 2 Debug Event Set to 1 if a DEVT2 debug event occurred
23	DCNT1	Debug Counter 1 Debug Event Set to 1 if a DCNT1 debug event occurred
24	DCNT2	Debug Counter 2 Debug Event Set to 1 if a DCNT2 debug event occurred
25	CIRPT	Critical Interrupt Taken Debug Event Set to 1 if a Critical Interrupt Taken debug event occurred.
26	CRET	Critical Return Debug Event Set to 1 if a Critical Return debug event occurred
27	VLES	VLE Status Set to 1 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 3‘b000 and DBSR[IDE] is set to 1. Normally set to 3‘b000 by a non-DVC DAC. A DVC DAC may set this field to any value.
31	CNT1TRG	Counter 1 Triggered Set to 1 if Debug Counter 1 is triggered by a trigger event.

11.3.4 Debug External Resource Control Register (DBERC0)

The debug external resource control register (DBERC0) controls resource allocation when DBCR0[EDM] is set to ‘1’. 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. However, 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 by 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 (MSRDE] = 1 DBCR0[IDM] = 1 and DBCR0[EDM] = 0, or MSRDE] = 1, DBCR0[IDM] = 1 and DBCR0[EDM] = 1 and software is allocated resource(s) by DBERC0), a set bit in DBSR by an event that is software-owned (other than MRR 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).

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**, thus 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.

The DBERC0 register is shown in Figure 11-12.

0	IDM	RST	UDE	ICMP	BRT	IRPT	TRAP	IAC1	IAC2	IAC3	IAC4	DAC1	0	DAC2	0	RET	IAC5	IAC6	IAC7	IAC8	DEVT1	DEVT2	DCNT1	DCNT2	CIRPT	CRET	BKPT	DQM	0		FT
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

SPR - 569; Read-only by Software; Reset - Unaffected by **p_reset_b**, cleared by **m_por** or while in the test-logic-reset OnCE controller state

Figure 11-12. DBERC0 Register

Table 11-13 provides bit definitions for the debug external resource control register. Note that DBERC0 controls are disabled when DBCR0[EDM] = 0.

Table 11-14. 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$_{ICMP}$ field. 1 Event owned by software debug. DBCR0$_{ICMP}$ is software readable/writable.

Table 11-14. DBERC0 Bit Definitions (Continued)

Bit(s)	Name	Description
5	BRT	Branch Taken Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to $DBCR0_{BRT}$ field. 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 $DBCR0_{IRPT}$ field. 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 $DBCR0_{TRAP}$ field. 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.
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 $DBCR0_{RET}$ field. 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 fields are software readable/writable.

Table 11-14. DBERC0 Bit Definitions (Continued)

Bit(s)	Name	Description
21	DEVT1	External Debug Event Input 1 Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to DBCR0$_{DEVT1}$ field. 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$_{DEVT2}$ field. 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$_{CIRPT}$ field. 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$_{CRET}$ field. 1 Event owned by software debug. DBCR0$_{CRET}$ is software readable/writable.
27	BKPT	Breakpoint Instruction Debug Control 0 Breakpoint owned by hardware debug. Execution of a bkpt instruction (all 0's opcode) results in entry into debug mode. 1 Breakpoint owned by software debug. Execution of a bkpt instruction (all 0's 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$_{DQTAG}$ field 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$_{DQTAG}$ field 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 11-15 shows which resources are controlled by DBERC0 settings.

Table 11-15. 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$_{IDM}$

Table 11-15. 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	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	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$_{IAC2XM}$
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$_{IAC56M}$

Table 11-15. 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	—	—	—	—	—	—	—	—	—	—	—	—	—	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$_{DVC1M,DVC1BE}$ DBCR4$_{DVC1C,DAC1XM}$
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$_{DEVT2}$

Table 11-15. 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	—	—	—	—	—	—	DBCR0$_{DCNT1}$, 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	—	—	
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 11-15 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.

11.3.5 Debug Event Select Register (DEVENT)

The debug event select register allows instrumented software to internally generate signals when an **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 SoC User’s Manual for more information on SoC use cases.

The DEVENT[DEVNT] register field value is undefined on a read; it may or may not remain set to the last value written. Since 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 12.13.1, “Data Acquisition ID Tag Field,” for more detail on the DQTAG.

The DEVENT register is shown in Figure 11-13.

DQTAG	0	DEVNT
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

SPR - 975; Reset[1] - 0x0

Figure 11-13. DEVENT 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*. Note that DEVNT field is shared by hardware and software but is always reset by *p_reset_b*.

Table 11-16 provides bit definitions for the debug event register.

Table 11-16. 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

11.3.6 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+) by means of an **mtspr** instruction to this register. See Section 12.13, “Data Acquisition Messaging,” for details.

The DDAM register is shown in Figure 11-14.

DDAM																															
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

SPR - 576; Reset[1] - 0x0

Figure 11-14. DDAM 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 11-17 provides bit definitions for the debug data acquisition message register.

Table 11-17. 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)

11.4 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 the EDBCR0[EDM]/DBCR0[EDM] 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 as opposed to generating debug interrupts, unless the specific events are allocated to software by the settings in DBERC0. In debug mode, the CPU is halted at a recoverable boundary; an external debug control module may control CPU operation through the OnCE.

Note that the descriptions of events in the subsections of Section 11.2, “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.

NOTES

On the initial setting of [E]DBCR0[EDM] to ‘1’, other bits in DBCR0 will remain unchanged. After [E]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 OCRDR 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 which could 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 by the OCRDR, either by performing go+noexit commands or by assertion of the *jd_de_b* signal.

DBCR0[EDM] operation is blocked if OnCE operation is disabled (*jd_en_once* negated) regardless of whether it is set or cleared. This means that if DBCR0EDM was previously set, and then *jd_en_once* is 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) or DBCR0[EDM]. Register read data is synchronized back to the *j_tclk* clock domain. The OnCE control register provides the capability to signal the system level clock controller that the CPU clock should be activated if not already active.

Updates to the DBCRx, DBSR, and DBCNT registers by 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, it is possible that 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 by DBERC0, updates to the DBERC0, DBCRx, DBCNT, and DBSR registers must be performed with the CPU in debug mode because simultaneous updates of register portions could otherwise be attempted, which is not guaranteed to occur properly. The results of such an attempt are undefined.

11.4.1 External Debug Registers

The external debug registers are used for controlling several debug aspects of the core and reporting status while e200z446n3 is in external debug mode.

11.4.1.1 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.

EDBCR0 is shown in Figure 11-15.

EDM	DNH_EN	DTF		0																											
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

Reset[1] - 0x0

Figure 11-15. EDBCR0 Register

[1] EDBCR0 is affected (reset) by *j_trst_b* or *m_por* assertion, and remains reset while in the Test_Logic_Reset state, but is not affected by *p_reset_b*.

Table 11-18 provides bit definitions for external debug control register 0.

Table 11-18. 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

11.4.1.2 External Debug Status Register 0 (EDBSR0)

The external debug status register 0 (EDBSR0) contains the status on debug events owned by hardware. The external debug status register 0 is set by the hardware, and read and cleared by OnCE access by the debugger. Clearing is done by writing to the external debug status register through the OnCE port, with a ‘1’ in any bit position that is to be cleared and ‘0’ 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.

The EDBSR0 register is shown in Figure 11-16.

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
IDE	UDE	DNH	0	ICMP	BRT	IRPT	TRAP	IAC1	IAC2	IAC3	IAC4-8	DAC1R	DAC1W	DAC2R	DAC2W	RET	0				DEVT1	DEVT2	DCNT1	DCNT2	CIRPT	CRET	VLES	DAC_OFST			CNT1TRG

Read/Write; Reset[1] - 0x0000_0000

Figure 11-16. EDBSR0 Register

[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.

Table 11-19 provides bit definitions for external debug status register 0.

Table 11-19. EDBSR0 Bit Definitions

Bit(s)	Name	Description
0	IDE	Imprecise Debug Event Set to 1 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 SPE FP round exception.
1	UDE	Unconditional Debug Event Set to 1 if a hardware-owned Unconditional debug event occurred.
2	DNH	Debugger Notify Halt Event Set to 1 if a debugger notify halt instruction was executed and caused a debug halt.
3	—	Reserved
4	ICMP	Instruction Complete Debug Event Set to 1 if a hardware-owned Instruction Complete debug event occurred.
5	BRT	Branch Taken Debug Event Set to 1 if a hardware-owned Branch Taken debug event occurred.
6	IRPT	Interrupt Taken Debug Event Set to 1 if a hardware-owned Interrupt Taken debug event occurred.
7	TRAP	Trap Taken Debug Event Set to 1 if a hardware-owned Trap Taken debug event occurred.
8	IAC1	Instruction Address Compare 1 Debug Event Set to 1 if a hardware-owned IAC1 debug event occurred.
9	IAC2	Instruction Address Compare 2 Debug Event Set to 1 if a hardware-owned IAC2 debug event occurred.
10	IAC3	Instruction Address Compare 3 Debug Event Set to 1 if a hardware-owned IAC3 debug event occurred.
11	IAC4-8	Instruction Address Compare 4–8 Debug Event Set to 1 if a hardware-owned IAC4, IAC5, IAC6, IAC7, or IAC8 debug event occurred.

Table 11-19. EDBSR0 Bit Definitions (Continued)

Bit(s)	Name	Description
12	DAC1R	Data Address Compare 1 Read Debug Event Set to 1 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 to 1 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 to 1 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 to 1 if a hardware-owned write-type [DAC2] debug event occurred while DBCR0[DAC2] = 0b01 or DBCR0[DAC2] = 0b11
16	RET	Return Debug Event Set to 1 if a hardware-owned Return debug event occurred
17–20	—	Reserved
21	DEVT1	External Debug Event 1 Debug Event Set to 1 if a hardware-owned DEVT1 debug event occurred
22	DEVT2	External Debug Event 2 Debug Event Set to 1 if a hardware-owned DEVT2 debug event occurred
23	DCNT1	Debug Counter 1 Debug Event Set to 1 if a hardware-owned DCNT1 debug event occurred
24	DCNT2	Debug Counter 2 Debug Event Set to 1 if a hardware-owned DCNT2 debug event occurred
25	CIRPT	Critical Interrupt Taken Debug Event Set to 1 if a hardware-owned Critical Interrupt Taken debug event occurred.
26	CRET	Critical Return Debug Event Set to 1 if a hardware-owned Critical Return debug event occurred
27	VLES	VLE Status Set to 1 if a hardware-owned 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 hardware-owned DAC Debug exception, unless a simultaneous DTLB or DSI error occurs, in which case this field is set to 3‘b000 and EDBSR0[IDE] is set to 1. Normally set to 3‘b000 by a non-DVC DAC. A DVC DAC may set this field to any value.
31	CNT1TRG	Counter 1 Triggered Set to 1 if hardware-owned Debug Counter 1 is triggered by a trigger event.

11.4.2 OnCE Introduction

The e200z446n3 on-chip emulation circuitry (OnCE™/Nexus Class 1 interface) provides a means of interacting with the e200 core and integrated system so that a user may examine registers, memory, or on-chip peripherals facilitating hardware/software development. OnCE operation is controlled by 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 e200z446n3 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 e200z446n3 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.

Figure 11-17 shows the TAP controller state model and the TAP registers implemented by the OnCE logic.

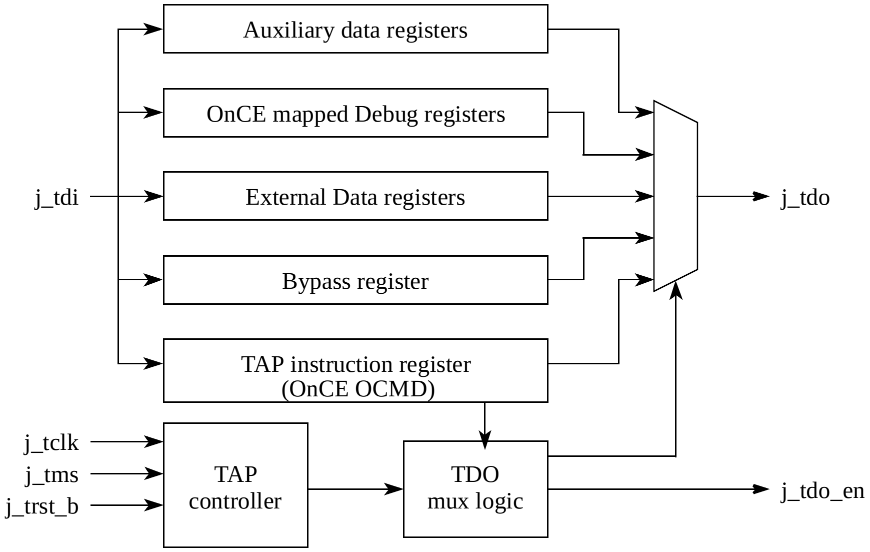


Figure 11-17. OnCE TAP Controller and Registers

Figure 11-18 shows the OnCE controller implemented as a 16-state FSM, with a one-to-one correspondence to the states defined for the JTAG TAP controller.

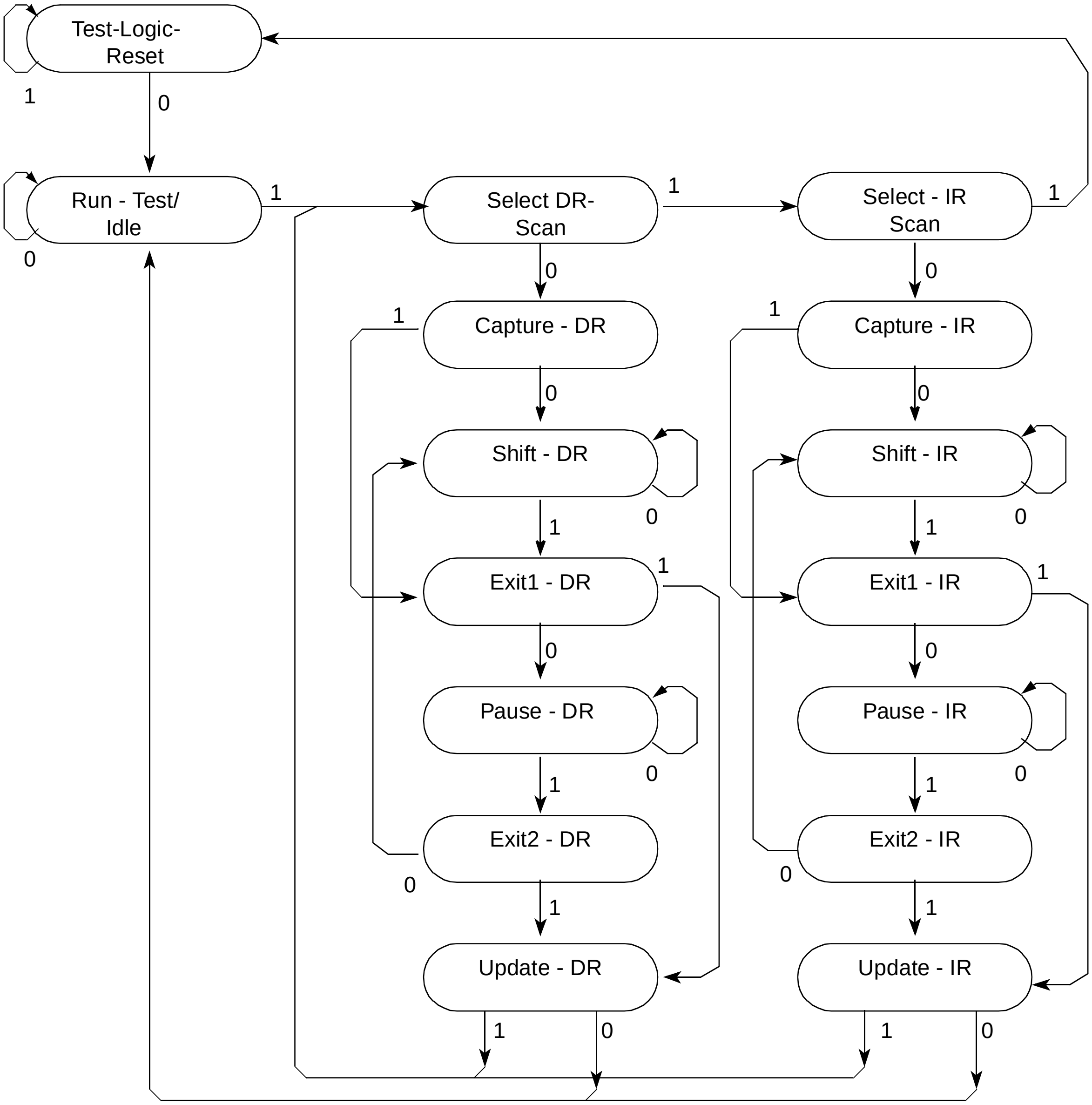


Figure 11-18. OnCE TAP Controller and Registers (16-State FSM)

Access to processor registers and the contents of memory locations happens through the following sequence:

1. Enable external debug mode (setting DBCR0[EDM] to ‘1’).
2. Place the processor into debug mode.
3. Scan instructions and data into and out of the CPU Scan Chain (CPUSCR).

4. Access required data by CPU execution of scanned instructions. Memory locations may be read by scanning a load instruction into the e200 core (which references the desired memory location), executing the load instruction, and then scanning out the result of the load. Other resources are accessed in a similar manner.

A debug session begins with the CPU's initial entry into debug mode from normal, waiting, stopped, or halted states (all indicated by the OnCE Status Register (OSR) as discussed in Section 11.4.6.1, "e200 OnCE Status Register,") by assertion of one or more debug requests. 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 then be single-stepped by scanning new values into the CPUSCR and performing a OnCE go+noexit command (See Section 11.4.6.2, "e200 OnCE Command Register (OCMD)"). The CPU then temporarily exits the debug state—but not the debug session—to execute the instruction It next returns to the debug state, again indicated by the 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. A scan into the CPUSCR is required prior to executing each go+exit or go+noexit OnCE command.

11.4.3 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 but 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. This causes the CPU to finish the 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 11-20 details the primary JTAG/OnCE interface signals.

Table 11-20. 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 13.3.23, "JTAG Support Signals—Primary Interface."

11.4.4 OnCE Internal Interface Signals

The following list describes the e200z446n3 OnCE interface signals to other internal blocks associated with the OnCE controller.

- CPU Debug Request (dbg_dbgrq)
  - Asserted by the OnCE control logic to request that the CPU enters the debug state.
  - May be asserted for a number of different conditions
  - Causes the CPU to finish the instruction being executed, save the instruction pipeline information, enter the debug mode, and wait for further commands.
- CPU Debug Acknowledge (cpu_dbgack)
  - Asserted by the CPU upon entering the debug state.
  - 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.
- CPU Address, Attributes
  - Used by a Nexus class 2-4 debug unit with information for real-time address trace information.
- CPU Data
  - Used to supply a Nexus class 2-4 debug unit with information for real-time data trace capability.

11.4.5 OnCE Interface Signals

The following paragraphs describe additional OnCE interface signals to other external blocks such as a Nexus Controller and external blocks which may need information pertaining to debug operation.

11.4.5.1 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 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. The OnCE status register is not visible when OnCE operation is disabled. In addition, OnCE control register 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 •, “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 affect after one additional *j_tclk* cycle of synchronization. In addition, *jd_enable_once* must not change state during a debug session, or undefined activity may occur.

11.4.5.2 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 e200z4 interface) provides a fast means of entering debug mode from an external command controller (when input) as well as a fast means of acknowledging to an external command controller (when output) that debug mode has been entered.

The assertion of this pin by a command controller causes the CPU core to finish the 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 then *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 is 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 which 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.

11.4.5.3 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. This may involve the CPU executing a sequence of instructions that are not part of the normal instruction stream that the CPU executes in other modes solely for the purpose of visibility/system control. 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 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 11.4.6.2, “e200 OnCE Command Register (OCMD),” for more information about the function of the GO and EX bits.

This signal is not normally used by the CPU.

11.4.5.4 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. It is synchronized to the *j_tclk* (scan clock) clock domain and presented as a status flag on the *j_tdo* output during the Shift_IR state. External firmware may use this signal to ensure that proper scan sequences will occur to access debug resources in the *m_clk* clock domain.

11.4.5.5 Watchpoint Events (jd_watchpt[0:21])

The *jd_watchpt[0:21]* signals may be asserted by the OnCE control logic to signal that a watchpoint condition has occurred. Watchpoints do not affect the CPU. 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 control register settings.

11.4.6 e200 OnCE Controller and Serial Interface

The OnCE controller contains the OnCE command register, the OnCE decoder, and the status/control register. In operation, the OnCE command register acts as the IR for the e200 TAP controller, and all other OnCE resources are treated as data registers (DR) by the TAP controller. The command register is loaded during the Update-IR state by serially shifting in commands during the TAP controller Shift-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 11-19 is a block diagram of the OnCE controller and serial interface.

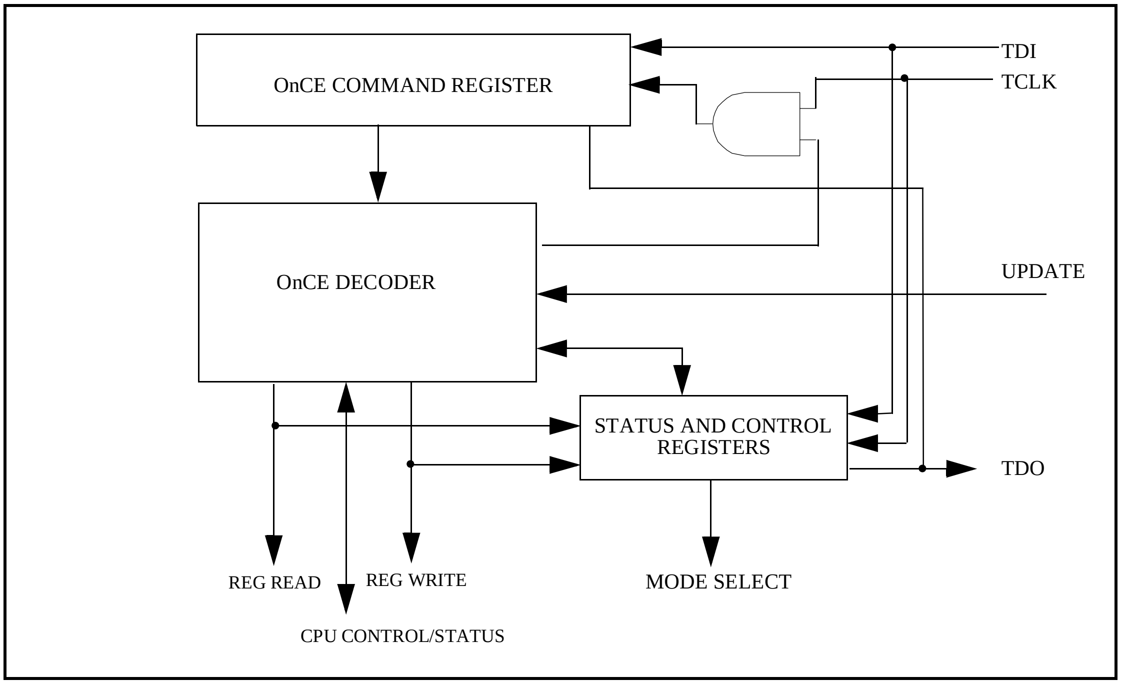


Figure 11-19. e200 OnCE Controller and Serial Interface

11.4.6.1 e200 OnCE Status Register

Status information regarding the state of the CPU is latched into the OnCE status register 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.

Figure 11-20 shows the OnCE status register.

MCLK	ERR	0	RESET	HALT	STOP	DEBUG	WAIT	0	1
0	1	2	3	4	5	6	7	8	9

Figure 11-20. OnCE Status Register

Table 11-21 provides bit definitions for the OnCE status register.

Table 11-21. 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 0
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 for IEEE 1149.1 conformity
9	1	Reserved, set to 1 for IEEE 1149.1 conformity

11.4.6.2 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 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. 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.

The OnCE command register is shown in Figure 11-21.

R/W	GO	EX	RS[0:6]						
0	1	2	3	4	5	6	7	8	9

Reset - 10’b1000000010 on assertion of *j_trst_b* or *m_por*, or while in the Test_Logic_Reset state

Figure 11-21. OnCE Command Register

Table 11-22 provides bit definitions for the OnCE command register.

Table 11-22. 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. The OSR$_{ERR}$ bit 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 11.4.9.6, “Exiting Debug Mode and Interrupt Blocking.”

Table 11-22. 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 EDBSR0 is set and $DBCR0_{EDM}$=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 11.4.9.6, “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 11-23 indicates the e200 OnCE register addresses. Attempted writes to read-only registers are ignored.

Table 11-23 shows the e200 OnCE register addresses.

Table 11-23. 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)
010 0111	Data Value Compare 2 (DVC2)
010 1000	Instruction Address Compare 5 (IAC5)

Table 11-23. e200 OnCE Register Addressing

RS[0–6]	Register Selected
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 1100	Reserved (do not access)
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	Shared Nexus Control Register Select
111 0000–111 1001	General Purpose register selects [0–9] (*j_gp_regsel[0:9]*)
111 1010	Cache Debug Access Control Register (CDACNTL)(See Section 9.15, "Cache Memory Access For Debug / Error Handling)
111 1011	Cache Debug Access Data Register (CDADATA)(See Section 9.15, "Cache Memory Access For Debug / Error Handling)
111 1100	Nexus2/3–Access
111 1101	LSRL Select (see Test Specification)
111 1110	Enable_OnCE[1]
111 1111	Bypass

[1] Causes assertion of the *j_en_once_regsel* output. Refer to Section 13.2.22.15, "Enable Once Register Select (j_en_once_regsel)"

The OnCE decoder receives the 10-bit command from the OCMD and status signals from the processor as input. It generates all 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 CPUre-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, and so on) and may prevent the desired instruction from being successfully executed. 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, [E]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 [E]DBCR0[EDM]. They may still be modified during a debug session by a single-stepped **mtspr** instruction or by OnCE access if [E]DBCR0[EDM] is set.

11.4.6.3 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 11.6, “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).

Figure 11-22 shows the OCR.

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								I_DMDIS	0		I_DVLE	I_DI	I_DM	0	I_DE	D_DMDIS	0		D_DW	D_DI	D_DM	D_DG	D_DE	0					WKUP	FDB	DR

Reset - 0x000_0000 on **m_por**, **j_trst_b**, or entering Test_logic_Reset state

Figure 11-22. OnCE Control Register

Table 11-24 provides bit definitions for the OnCE Control Register.

Table 11-24. 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 to‘1’ to allow full access. When disabled, no TLB miss or TLB exceptions are generated for Instruction accesses. External access errors can still occur.

Table 11-24. OnCE Control Register Bit Definitions (Continued)

Bit(s)	Name	Description
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 to‘1’ 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.
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 Zen *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.

Table 11-24. OnCE Control Register Bit Definitions (Continued)

Bit(s)	Name	Description
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 that 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 will indicate 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 will enter Debug mode at the next instruction boundary.

11.4.7 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 core is running without interfering with processor execution. Accesses to other resources such as the CPUSCR require that the core 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 core prior to access. Note that a scan operation to update the CPUSCR is required prior to exiting debug mode once 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, and so on) 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 11-25 provides a list of access requirements for OnCE registers.

Table 11-25. 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	*DBCR0[EDM] access only requires *jd_en_once* asserted
EDBSR0	Y	Y	N	N	N	—
IAC1–8	Y	Y	Y	N	*[1]	—
JTAG ID	N	N	—	N	—	Read only

Table 11-25. 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	
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
Nexus2/3-Access	Y	N	N	N	N	—
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.

11.4.8 Methods of Entering Debug Mode

The OnCE status register indicates that the CPU has entered the debug mode by means of the debug status bit. The following sections describe how debug mode is entered assuming the OnCE circuitry has been enabled. In the e200, OnCE operation is enabled by the assertion of the *jd_en_once* input (see Section 11.4.5.1, “OnCE Enable (jd_en_once)”).

11.4.8.1 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 by the OnCE status register debug bit, the external command controller negates 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. 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 the debug controller must cause the appropriate reset to be re-asserted.

11.4.8.2 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 the debug controller must cause the appropriate reset to be re-asserted.

11.4.8.3 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.

11.4.8.4 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 its prior state.

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 either OCRDR 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.

11.4.8.5 Software Request During Normal Activity

Upon executing a ‘**bkpt’** pseudo-instruction (for e200, defined to be an all zeros 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 debug mode after the instruction following the ‘**bkpt’** pseudo-instruction has entered the instruction register.

11.4.8.6 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. For the e200z4, 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.

11.4.9 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 11-23 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.

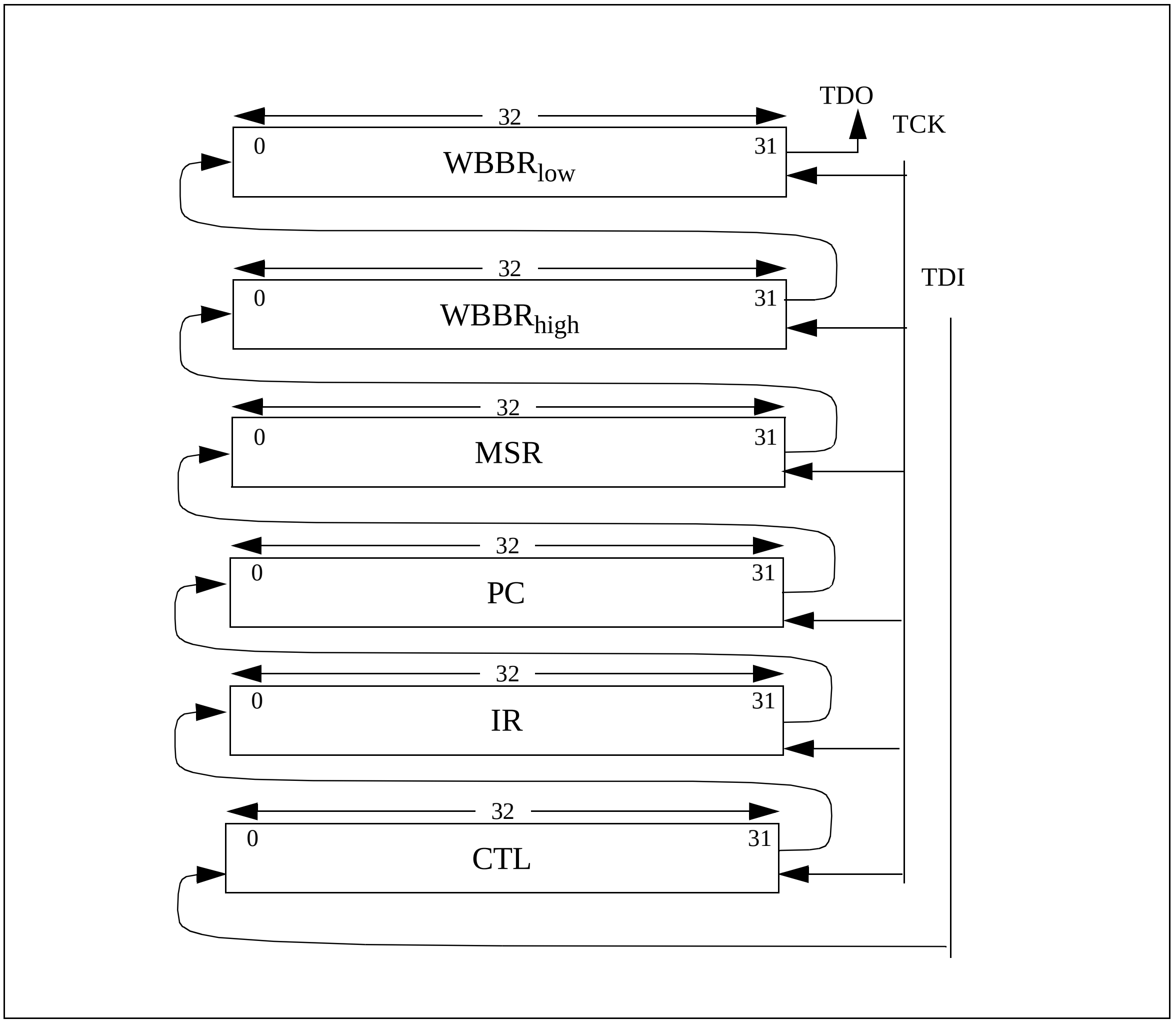


Figure 11-23. CPU Scan Chain Register (CPUSCR)

11.4.9.1 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. When the scan-out of this chain begins, this register contains the opcode of the next instruction to be executed upon entering debug mode. This value should be saved for later restoration if continuation of the normal instruction stream is desired.

On scan-in, this register is filled with an instruction opcode selected by debug control software in preparation for exiting debug mode. 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.

11.4.9.2 Control State Register (CTL)

The control state register (CTL) 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.

Figure 11-24 shows the control state register.

*											IRSTAT13	IRSTAT12	IRSTAT11	IRSTAT10	WAITING	PCOFST				PCINV	FFRA	IRSTAT0	IRSTAT1	IRSTAT2	IRSTAT3	IRSTAT4	IRSTAT5	IRSTAT6	IRSTAT7	IRSTAT8	IRSTAT9
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

Figure 11-24. Control State Register (CTL)

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 in Table 11-26.

Table 11-26. CTL Emulation Firmware Modifications

Bit(s)	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, i.e 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 11.4.5.3, “e200 OnCE Debug Output (jd_debug_b)”) which is not part of the normal program execution flow, these bits should be set to a 0.
11	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.
12	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.
13	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.
14	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.

Table 11-26. CTL Emulation Firmware Modifications (Continued)

Bit(s)	Name	Description
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	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 nop instruction instead of the original IR value, other wise 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 registers — 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
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.

Table 11-26. CTL Emulation Firmware Modifications (Continued)

Bit(s)	Name	Description
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 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 the 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 as follows.

- If set, the PC and IR should be initialized for performing whatever recovery sequence is appropriate for a faulted exception vector fetch.
- If 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. However, that instruction is not re-executed; the no-op instruction is executed instead, and instruction fetch and execution resumes at location PC+4. IRStat8 is used determine the type of instruction present in the IR and should be cleared in this case. Note that debug events that may occur on the no-op (ICMP) are generated (and optionally counted) if enabled.

For the CTL register, the internal state bits should be restored to their original value. The IRStatus bits should be set to zeros 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 is re-entered prior to any further instruction execution.

11.4.9.3 Program Counter Register (PC)

The PC is a 32-bit register that stores the value of the program counter that 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. 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.

11.4.9.4 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 that generate 64-bit results, WBBR[high] holds the high-order 32 bits of the result. It is undefined for other instructions.

For 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 then 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 that uses this value as a substitute data value is executed. The control state register FFRA bit forces the value of 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 11.4.9.2, “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. Due to control issues, they are 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 rn,rn,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.

11.4.9.5 Machine State Register (MSR)

The MSR is a 32-bit register that defines the state of the machine. Whenever the external command controller needs to save or modify the state of the machine, 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.

11.4.9.6 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; the CPU re-enters 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.

11.4.10 Instruction Address FIFO Buffer (PC FIFO)

To assist debugging and keep 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.

11.4.10.1 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 11-25 shows the block diagram of the PC FIFO.

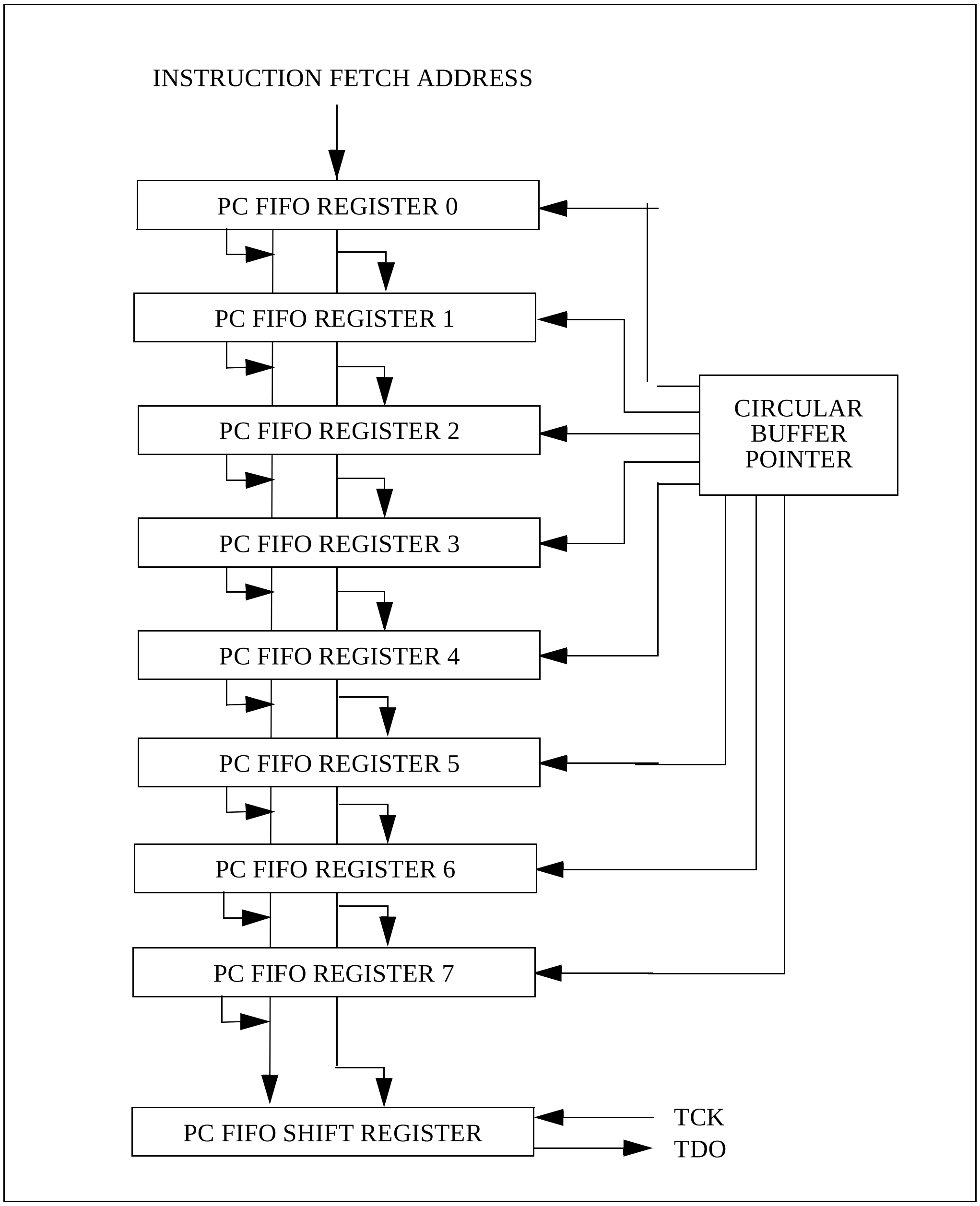


Figure 11-25. OnCE PC FIFO

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 returns 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). This allows 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.

11.4.11 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.

11.5 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 (DBCR0EDM = 1). Watchpoints are indicated with a dedicated set of interface signals. The *jd_watchpoint[0:21]* 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) and debug counter event (DCNT1–2), as well as 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 (other than 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.

Table 11-27 shows the watchpoint output signal assignments.

Table 11-27. 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
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]

Table 11-27. Watchpoint Output Signal Assignments

Signal Name	Type	Description
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
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]

[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.

11.6 MMU and Cache Operation During Debug

Normal operation of the MMU may be modified during a debug session by the OnCE control register (OCR). A debug session begins when the CPU initially enters debug mode, and ends when a 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 substitute default values for the access protection bits (UX, UR, UW, SX, SR, SW) and values obtained from the OnCE control register for page attribute bits (VLE, W, I, M, G, E) normally provided by a matching TLB entry. In addition, no address translation is performed. 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 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) remains in effect.

The OCR control bits are used when debug mode is entered. Refer to the bit definitions in the OCR (Section 11.4.6.3, “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.

11.7 Cache Array Access During Debug

The cache arrays may be read and written during debug mode by the CDACNTL and CDADATA debug registers. This functionality is described in detail in Section 9.15, “Cache Memory Access For Debug/Error Handling.”

11.8 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. This simplified flow is intended to illustrate basic operations, but does not cover all potential methods in depth.

To enable external debug mode and initialize debug registers:

1. The debugger should ensure that the *jd_en_once* control signal is asserted in order to enable OnCE operation
2. Select the OCR and write a value to it in which OCR[DR], OCR[WKUP], are set to ‘1’. 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
3. 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.
4. Select the DBCR0 register and update it with the DBCR0[EDM] bit set
5. Clear the DBSR status bits
6. Write appropriate values to the DBCRx, 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.

- Optionally, set OCR[I_DMDIS] and/or OCR[D_DMDIS] to ensure that no TLB misses 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:

1. 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 11.4.9.2, “Control State Register (CTL)”) with a Go+Noexit OnCE command value.
2. The debugger scans out the OSR with “no-register selected”, Go cleared, and 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):

1. The OCR[I_DMDIS, D_DMDIS], OCR[DR], control bits should be cleared, leaving the OCR[WKUP] bit set.
2. 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 11.4.9.2, “Control State Register (CTL)”) with a Go+Exit OnCE Command value.
3. OCR[WKUP] may then be cleared.

To exit external debug mode:

1. The debugger should place the CPU in the debug state by the OCR[DR] with OCR[WKUP] asserted, scanning out and saving the CPUSCR.
2. The debugger should write the DBCRx registers as needed, likely clearing every enable except DBCR0[EDM].
3. The debugger should write the DBSR to a cleared state.
4. The debugger should re-write the DBCR0 with all bits including EDM cleared.
5. The debugger should clear OCR[DR].
6. 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 11.4.9.2, “Control State Register (CTL)”) with a Go+Exit OnCE Command value.
7. OCR[WKUP] may then be cleared.

NOTE

These steps are examples. They are not an exact template for debugger operation.

11.9 Parallel Signature Unit

To support applications requiring system integrity checking during operation, the e200 core provides a parallel signature unit, which is capable of monitoring the CPU data read and data write AHB buses, and accumulating a pair of 32-bit MISR signatures of the data values transferred over these buses.

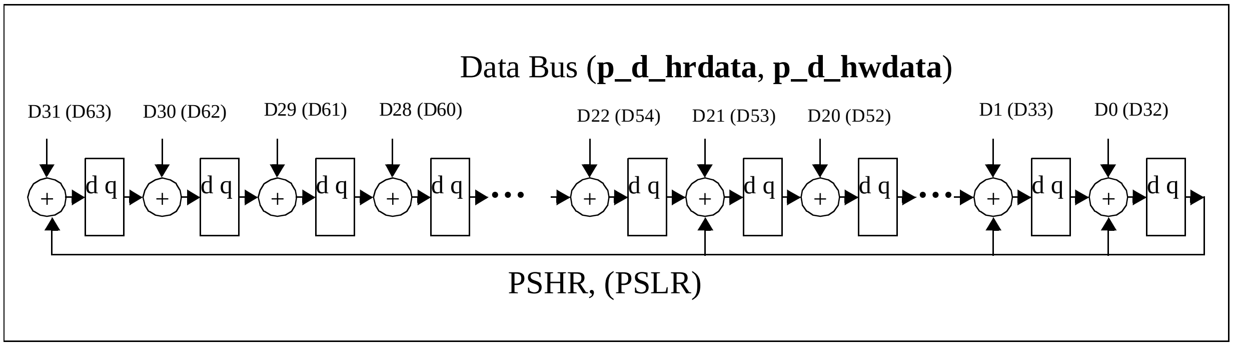
The primitive polynomial used is $P(X) = 1 + X^{10} + X^{30} + X^{31} + X^{32}$. Values are accumulated based on an initially programmed seed value and qualified based on active byte lanes of the data read and data write buses *(p_d_hrdata[63:0]*, *p_d_hwdata[63:0]*) as indicated by the *p_d_hbstrb[7:0]* signals. Inactive byte lanes use a value of all zeros as input data to the MISRs. Refer to Table 13-12 for active byte lane information. If a transfer 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. Errors occurring on data writes are not logged because the data driven by the CPU is valid.

The unit may be independently enabled for read cycles and write cycles, allowing for flexible usage. Software may also control accumulation of software provided values by a pair of update registers. In addition, a counter is provided for software use to monitor the number of beats of data that have been compressed.

Updates are performed when the parallel signature registers are initialized, when a qualified 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 an **mtdcr** instruction.

NOTE

Updates due to qualified bus transfers are suppressed for the duration of a debug session.



The parallel signature unit consists of seven registers as described below. 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 an **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**).

11.9.1 Parallel Signature Control Register (PSCR)

The parallel signature control register (PSCR) controls the operation of the parallel signature unit.

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																										CNTEN	0		RDEN	WREN	INIT

DCR - 272; Read/Write; Reset - 0x0

Figure 11-26. Parallel Signature Control Register (PSCR)

Table 11-28 shows the PSCR field descriptions.

Table 11-28. PSCR Field Descriptions

Bits	Name	Description
0–25	—	These bits are 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	—	These bits are reserved
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 ‘0’s (0x00000000). This bit always reads as ‘0’.

11.9.2 Parallel Signature Status Register (PSSR)

The parallel signature status register (PSSR) provides status relative to operation of the parallel signature unit.

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																															TERR

DCR - 273; Read/Write; Reset -Unaffected

Figure 11-27. Parallel Signature Status Register (PSSR)

Table 11-28 shows the PSSR field descriptions.

Table 11-29. 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 will cause it to be cleared.

11.9.3 Parallel Signature High Register (PSHR)

The parallel signature high register (PSHR) provides signature information for the high word (bits 63–32) of the AHB data read and data write buses. It may be written by an **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, it should be initialized by software prior to performing parallel signature operations.

High Signature
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

DCR - 274; Read/Write; Reset -Unaffected

Figure 11-28. Parallel Signature High Register (PSHR)

11.9.4 Parallel Signature Low Register (PSLR)

The parallel signature low register (PSLR) provides signature information for the low word (bits 31–0) of the AHB data read and data 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; thus, it should be initialized by software prior to performing parallel signature operations.

Low Signature
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

DCR - 275; Read/Write; Reset -Unaffected

Figure 11-29. Parallel Signature Low Register (PSLR)

11.9.5 Parallel Signature Counter Register (PSCTR)

The parallel signature counter register (PSCTR) provides count information for signature accumulation. The counter is incremented on every accumulated transfer or on an **mtdcr psulr, Rn** instruction. It may be written by an **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; thus, it should be initialized by software prior to performing parallel signature operations.

Counter
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

DCR - 276; Read/Write; Reset -Unaffected

Figure 11-30. Parallel Signature Counter Register (PSCTR)

11.9.6 Parallel Signature Update High Register (PSUHR)

The parallel signature update high register (PSUHR) provides a means for updating the high signature value by software. It may be written by 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.

High Signature Update Data
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

DCR - 277; Write-only; Reset -Unaffected

Figure 11-31. Parallel Signature Update High Register (PSUHR)

11.9.7 Parallel Signature Update Low Register (PSULR)

The parallel signature update low register (PSULR) provides a means for updating the low signature value by software. It may be written by an **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.

Low Signature Update Data
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

DCR - 278; Write-only; Reset -Unaffected

Figure 11-32. Parallel Signature Update Low Register (PSULR)

Chapter 12 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 proposed IEEE-ISTO Nexus 5001-2008™ standard. The development features supported are Program Trace, Data Trace, Watchpoint Messaging, Ownership Trace, Data Acquisition Messaging, and Read/Write Access through the JTAG interface. The Nexus 3+ module also supports two Class 4 features: Watchpoint Triggering and Processor Overrun Control.

12.1 Introduction

The Nexus 3+ module provides real-time development capabilities for e200 processors in compliance with the proposed IEEE-ISTO Nexus 5001-2008 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 5001-2008 standard defines an extensible auxiliary port that is used in conjunction with the JTAG port in e200z4 processors.

The Nexus modules are coupled to the CPU core and monitor a variety of signals including addresses, data, control signals, status signals, and so forth. Some SoC designs may use a single shared Nexus module with the capability of selectively monitoring more than one CPU. Control over this selection of the source if information is provided by a SoC-level shared Nexus control module, which is accessed through JTAG by the Nexus 1 shared Nexus control register. Specifics of this module are provided in a separate document. The CPU provides an interface signal to communicate the selection of this register.

12.1.1 Terms and Definitions

Table 12-1 contains a set of terms and definitions associated with the Nexus 3+ module.

Table 12-1. Terms and Definitions

Term	Description
IEEE-ISTO 5001	Consortium and standard for real-time embedded system design. World wide Web documentation at http://www.nexus5001.org
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 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.

Table 12-1. Terms and Definitions (Continued)

Term	Description
Data Acquisition Messaging (DQM)	Data Acquisition Messaging (DQM) allows code to be instrumented to export customized information to the Nexus Auxiliary Output Port.
JTAG Compliant	Device complying to IEEE 1149.1 JTAG standard
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 by an 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 conforms to Class1 of the IEEE-ISTO 5001-2008 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 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.

12.1.2 Feature List

The Nexus 3+ module conforms to Class 3 of the IEEE-ISTO 5001-2008 standard, with additional Class 4 features available. The following features are implemented:

- Program Trace through Branch Trace Messaging (BTM).
  - Branch trace messaging displays program flow discontinuities (direct and indirect branches, exceptions, and so on), allowing the development tool to interpolate what transpires between the discontinuities. Thus static code may be traced.
- Data Trace through 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 through Ownership Trace Messaging (OTM).
  - OTM facilitates ownership trace by providing visibility of which process ID or operating system task is activated. An Ownership Trace Message is transmitted when a new process/task is activated, allowing the development tool to trace ownership flow.
- Run-time access to embedded processor memory map through the JTAG port, which allows enhanced download/upload capabilities.

- Watchpoint Messaging through 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 through 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_wevt[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

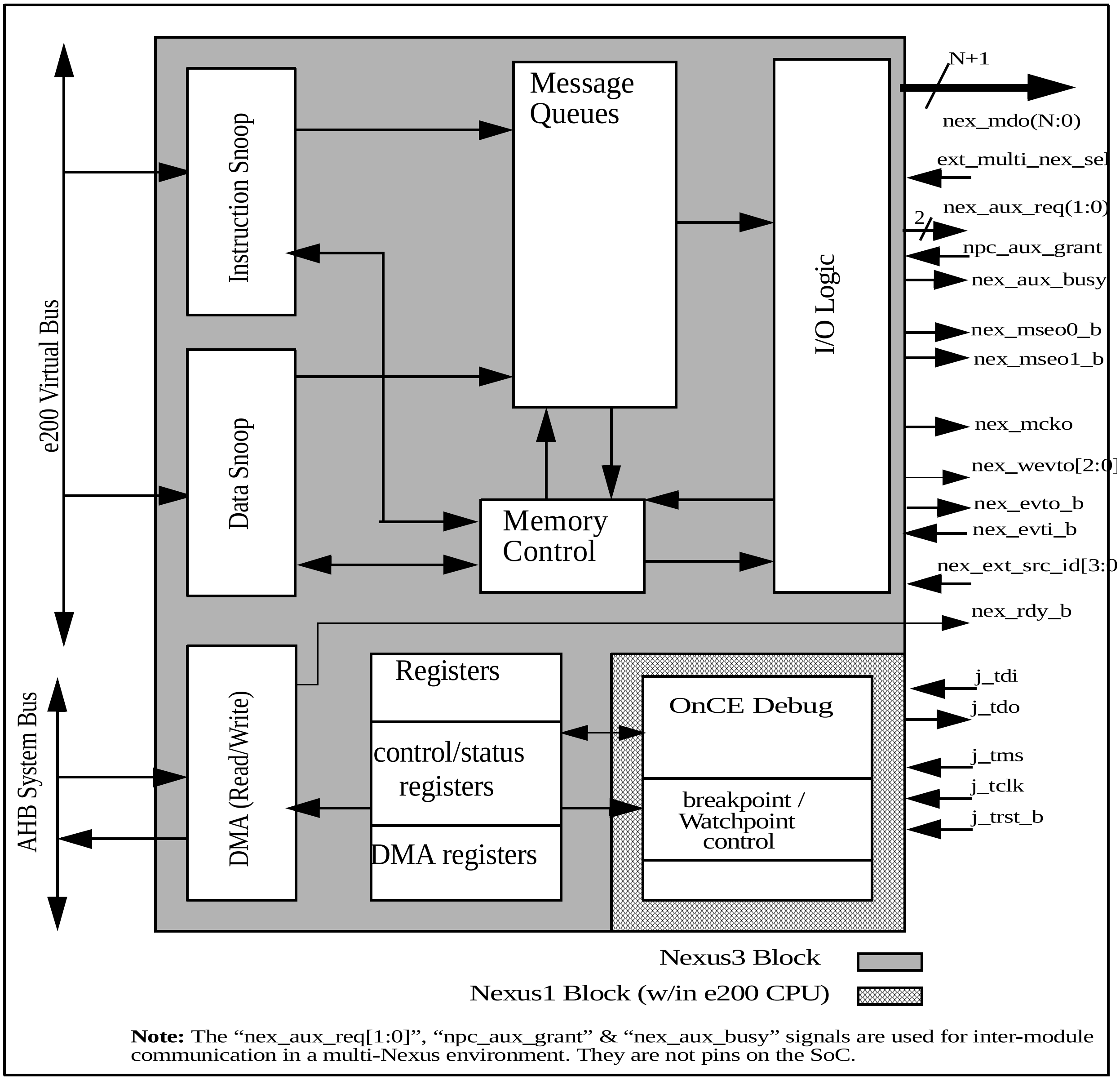
For multi-Nexus implementations, the configuration of the Message Data Out pins is controlled by the port control register (at the SoC level). For single Nexus implementations, this configuration is controlled by development control register 1 (DC1) within the Nexus 3+ module.

In either implementation, both full port mode (maximum number of MDO pins) or reduced port mode (minimum number of MDO pins) are supported. This setting should not be changed 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 will be hard-wired based on SOC bandwidth requirements.

12.1.3 Functional Block Diagram

Figure 12-1 shows the Nexus 3+ functional block diagram.



Note: The “nex_aux_req[1:0]”, “npc_aux_grant” & “nex_aux_busy” signals are used for inter-module communication in a multi-Nexus environment. They are not pins on the SoC.

Figure 12-1. Nexus 3+ Functional Block Diagram

12.2 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 will be ready to accept control input by 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 be 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 are not be available for reads or writes.

NOTE

Please refer to the “Nexus 3 Integration Guide” for details on IEEE-ISTO 5001 compliance with regard to output pins and multiple Nexus module configurations.

12.3 TCODEs Supported

The Nexus 3+ pins allow for flexible transfer operations through 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 5001-2008 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 12-2.

Table 12-2. Supported TCODEs

Message Name	Minimum Field Size (bits)	Maximum 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	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	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

Table 12-2. Supported TCODEs (Continued)

Message Name	Minimum Field Size (bits)	Maximum 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 12-7)
	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
	3	3	DSZ	fixed	data size (Refer to Table 12-7)
	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 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 12-7)
	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 DEVENT$_{DQTAG}$ 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	EVCODE	fixed	error code
Program Trace Direct Branch Message w/ Sync	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)

Table 12-2. Supported TCODEs (Continued)

Message Name	Minimum Field Size (bits)	Maximum Field Size (bits)	Field Name	Field Type	Field Description
Program Trace Indirect Branch Message w/ Sync	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)
Data Trace Data Write Message w/ 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 12-7)
	1	32	F-ADDR	variable	full access address (leading zeros truncated)
	1	64	DATA	variable	data write value(s) (see Data Trace section for details)
Data Trace Data Read Message w/ 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 12-7)
	1	32	F-ADDR	variable	full access address (leading zeros truncated)
	1	64	DATA	variable	data read value(s) (see Data Trace section for details)
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 12-5) indicates which resource is the cause of this message
	1	32	RDATA	variable	branch/predicate instruction history (see Section Section 12.11.4, “Resource Full Messages”)

Table 12-2. Supported TCODEs (Continued)

Message Name	Minimum Field Size (bits)	Maximum Field Size (bits)	Field Name	Field Type	Field Description
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 12.11.1, “Branch Trace Messaging Types”)
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 12.11.1, “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 with program flow (Refer to Table 12-6)
	2	2	CDF	fixed	# fields of information in CDATA. 01 one field (CDATA1), 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 12.11.5, “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 12.11.5, “Program Correlation Messages”)
	0	32	CDATA3	variable	correlation data field 3 - TLB info -ADDR_P for physical address (see Section 12.11.5, “Program Correlation Messages”)

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 12.11.1, “Branch Trace Messaging Types.” If the branch history method is selected, the shaded TCODES above will not be messaged out.

Table 12-3 shows the error code encodings used when reporting an error via the Nexus 3+ error message.

Table 12-3. 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 12-3 shows the error type encodings used when reporting an error through the Nexus 3+ error message.

Table 12-4. 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 12-5 shows the encodings used for resource codes for certain messages.

Table 12-5. 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 12-6 shows the event code encodings used for certain messages.

Table 12-6. Event Code Encoding (TCODE = 33)

Event Code	Description
0000	Entry into Debug Mode
0001	Entry into Low Power Mode (CPU only)
0010–0011	Reserved for future functionality
0100	Disabling Program Trace
0101	New process ID value is established in PID0 by **mtspr PID0**, or new value for $\mathrm{MSR}_{\mathrm{IS}}$ is established by an **mtmsr** instruction
0110–1001	Reserved for future functionality
1010	Branch and link occurrence (direct branch function call)[1]
1011	New Address Translation established in the TLB by **tlbwe**
1100	Address Translation entries invalidated in the TLB by **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)

[1] Only used for Program Trace—History Mode

Table 12-7 shows the data trace size encodings used for certain messages.

Table 12-7. 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	Reserved
0100	Word (4 bytes)
0100–0111	Reserved
1000	Double word (8 bytes)
1001–1111	Reserved

12.4 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 12.5, “Nexus 3+ Register Access via JTAG/OnCE,” 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 12-8 details the register map for the Nexus 3+ module.

Table 12-8. 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
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
Program Trace End Trigger Control (PTETC)	0x36	R/W	0x6C	0x6D

Table 12-8. Nexus 3+ Register Map (Continued)

Nexus Register	Nexus Access Opcode	Read/ Write	Read Address	Write Address
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 "the e200 Nexus 3 Integration Guide" for more information on how this parameter is implemented for each Nexus module.

12.4.1 Client Select Control (CSC)—reference only

The CSC register determines which Nexus client is under development. This register is present at the top-level SOC Nexus controller to select one of multiple on-chip Nexus 3+ units.

Figure 12-2 shows the client select control register.

7	6	5	4	3	2	1	0
Reserved				CS			

Nexus Reg# - 0x1;
Read-only; Reset - 0x0

Figure 12-2. Client Select Control Register

Table 12-9 shows the CSC fields.

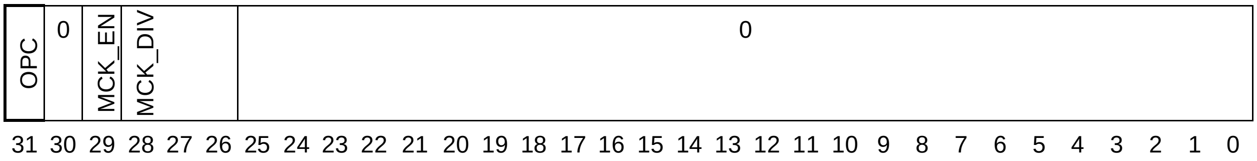
Table 12-9. Client Select Control Register Fields

CSC[7–4]	RES Reserved for future Nexus Clients (read as 0)
CSC[3–0]	CSC Client Select Control 0xX Nexus client (SoC level)

12.4.2 Port Configuration Register (PCR)—reference only

The port configuration register (PCR) 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.

Figure 12-3 shows the port configuration register.



Nexus Reg# - PCR_INDEX; Read/Write; Reset - 0x0

Figure 12-3. Port Configuration Register

Table 12-10 shows the PCR fields.

Table 12-10. Port Configuration Register Fields

Bits	Name	Description
31	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	MCKO Clock Enable (SoC Level) 0 **nex_mcko** is disabled 1 **nex_mcko** is enabled
28–26	MCK_DIV	MCKO Clock Divide Ratio (see note below) (SoC Level) 000 **nex_mcko** is 1x processor clock freq. 001 **nex_mcko** is 1/2x processor clock freq. 010 Reserved (default to 1/2x processor clock freq.) 011 **nex_mcko** is 1/4x processor clock freq. 100–110 Reserved (default to 1/2x processor clock freq.) 111 **nex_mcko** is 1/8x processor clock freq.
25–0	—	Reserved for future functionality

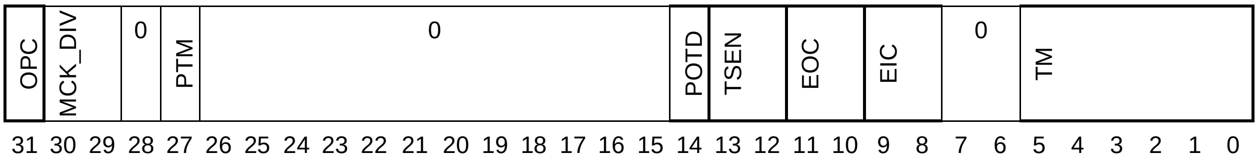
NOTE

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. Instead, the e200 Nexus 3+ defined development control register 1 (DC1) is used to control the SoC-level Nexus port functionality.

12.4.3 Nexus Development Control Register 1 (DC1)

Nexus development control register 1 is used to control basic development features of the Nexus 3+ module.

Development control register 1 is shown in Figure 12-4.



Nexus Reg# - 0x2; Read/Write; Reset - 0x0

Figure 12-4. Development Control Register 1

Development control register 1's fields are described in Table 12-11.

Table 12-11. Development Control Register 1 Fields

Bits	Name	Description
31	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	MCKO Clock Divide Ratio (see note below) 00 **nex_mcko** is 1x processor clock frequency 01 **nex_mcko** is 1/2x processor clock frequency 10 **nex_mcko** is 1/4x processor clock frequency 11 **nex_mcko** is 1/8x processor clock frequency
28–15	—	Reserved for future functionality
27	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	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	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 12-11. Development Control Register 1 Fields (Continued)

Bits	Name	Description
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 12.4.7, “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 be modified only 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 those two modes.

12.4.4 Nexus Development Control Registers 2 and 3 (DC2, DC3)

Nexus development control registers 2 and 3 are used to control output signaling on the Nexus 3+ module. A table of watchpoints can be found in Table 11-27.

Figure 12-5 shows development control register 2.

0				WEVTO[2]C				WEVTO[1]C				WEVTO[0]C				EWC															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Nexus Reg# - 0x3; Read/Write; Reset - 0x0

Figure 12-5. Development Control Register 2

Table 12-12 describes development control register 2’s fields.

Table 12-12. Development Control Register 2 Fields

Bits	Name	Description
31–28	—	Reserved
27–24	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	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 12-12. Development Control Register 2 Fields (Continued)

Bits	Name	Description
19–16	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	EVTO Watchpoint Configuration[1] 0000000000000000 No Watchpoints 0–15 trigger **nex_evto_b** XXXXXXXXXXXXXXX1Watchpoint #0 triggers **nex_evto_b** XXXXXXXXXXXXXX1XWatchpoint #1 triggers **nex_evto_b** XXXXXXXXXXXXX1XXWatchpoint #2 triggers **nex_evto_b** XXXXXXXXXXXX1XXXWatchpoint #3 triggers **nex_evto_b** XXXXXXXXXXX1XXXXWatchpoint #4 triggers **nex_evto_b** XXXXXXXXXX1XXXXXWatchpoint #5 triggers **nex_evto_b** XXXXXXXXX1XXXXXXWatchpoint #6 triggers **nex_evto_b** XXXXXXXX1XXXXXXXWatchpoint #7 triggers **nex_evto_b** XXXXXXX1XXXXXXXXWatchpoint #8 triggers **nex_evto_b** XXXXXX1XXXXXXXXXWatchpoint #9 triggers **nex_evto_b** XXXXX1XXXXXXXXXXWatchpoint #10 triggers **nex_evto_b** XXXX1XXXXXXXXXXXWatchpoint #11 triggers **nex_evto_b** XXX1XXXXXXXXXXXXWatchpoint #12 triggers **nex_evto_b** XX1XXXXXXXXXXXXXWatchpoint #13 triggers **nex_evto_b** X1XXXXXXXXXXXXXXWatchpoint #14 triggers **nex_evto_b** 1XXXXXXXXXXXXXXXWatchpoint #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.

Figure 12-5 shows the development control register 3.

0				WEVTO[2]C				WEVTO[1]C				WEVTO[0]C				0										EWC					
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Nexus Reg# - 0x4; Read/Write; Reset - 0x0

Figure 12-6. Development Control Register 3

Table 12-12 shows the development control register 3’s fields.

Table 12-13. Development Control Register 3 Fields

Bits	Name	Description
31–28	—	Reserved
27–24	WEVTO[2]C	Watchpoint Event Out 2 Configuration 0000 No Watchpoints 15–21 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]** 1000–1111 Reserved
23–20	WEVTO[1]C	Watchpoint Event Out 1 Configuration 0000 No Watchpoints 15–21 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]** 1000–1111 Reserved
19–16	WEVTO[0]C	Watchpoint Event Out 0 Configuration 0000 No Watchpoints 15–21 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]** 1000–1111 Reserved
15–6	—	Reserved for watchpoint expansion
5–0	EWC	EVTO Watchpoint Configuration[1] 000000 No Watchpoints 16–21 trigger **nex_evto_b** XXXXX1 Watchpoint #16 triggers **nex_evto_b** XXXX1X Watchpoint #17 triggers **nex_evto_b** XXX1XX Watchpoint #18 triggers **nex_evto_b** XX1XXX Watchpoint #19 triggers **nex_evto_b** X1XXXX Watchpoint #20 triggers **nex_evto_b** 1XXXXX Watchpoint #21 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.

12.4.5 Nexus Development Control Register 4 (DC4)

Nexus development control register 4 is used to control masking of events that initiate program correlation messages on the Nexus 3+ module.

Figure 12-7 shows the development control register 4.

0	EVCDM
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Nexus Reg# - 0x5; Read/Write; Reset - 0x0

Figure 12-7. Development Control Register 4

Table 12-14 shows the development control register 4’s fields.

Table 12-14. Development Control Register 4 Fields

Bits	Name	Description
31–16	—	Reserved
15–0	EVCDM	Event Code (EVCODE) Mask[1] 0000000000000000No EVCODEs masked for Program Correlation Messages XXXXXXXXXXXXXXX1EVCODE #0 is masked for Program Correlation Messages XXXXXXXXXXXXXX1XEVCODE #1 is masked for Program Correlation Messages XXXXXXXXXXXXX1XXEVCODE #2 is masked for Program Correlation Messages XXXXXXXXXXXX1XXXEVCODE #3 is masked for Program Correlation Messages XXXXXXXXXXX1XXXXEVCODE #4 is masked for Program Correlation Messages XXXXXXXXXX1XXXXXEVCODE #5 is masked for Program Correlation Messages XXXXXXXXX1XXXXXXEVCODE #6 is masked for Program Correlation Messages XXXXXXXX1XXXXXXXEVCODE #7 is masked for Program Correlation Messages XXXXXXX1XXXXXXXXEVCODE #8 is masked for Program Correlation Messages XXXXXX1XXXXXXXXXEVCODE #9 is masked for Program Correlation Messages XXXXX1XXXXXXXXXXEVCODE #10 is masked for Program Correlation Messages XXX1XXXXXXXXXXXEVCODE #11 is masked for Program Correlation Messages XXX1XXXXXXXXXXXXEVCODE #12 is masked for Program Correlation Messages XX1XXXXXXXXXXXXXEVCODE #13 is masked for Program Correlation Messages X1XXXXXXXXXXXXXXEVCODE #14 is masked for Program Correlation Messages 1XXXXXXXXXXXXXXXEVCODE #15 is masked for Program Correlation Messages

[1] Refer to Table 12-6 for implemented EVCODEs

12.4.6 Development Status Register (DS)

The Development Status Register 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 (see note below), a debug status message is transmitted with DS[31–24]. The external tool can read this register at any time.

Figure 12-8 shows the development status register.

DBG	LPS	LPC	0
31	30 29 28	27 26	25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Nexus Reg# - 0x30; Read-only; Reset - 0x0

Figure 12-8. Development Status Register

Table 12-15 shows the development status register’s fields.

Table 12-15. Development Status Register Fields

Bits	Name	Description
31	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	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	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)

12.4.7 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 4’b1111 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 but start and stop triggers occur simultaneously, tracing will be 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 11-27.

Figure 12-9 shows the watchpoint trigger register.

PTS	PTE	DTS	DTE	0

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Nexus Reg# - 0xB; Read/Write; Reset - 0x0

Figure 12-9. Watchpoint Trigger (WT) Register

Table 12-16 details the watchpoint trigger register fields.

Table 12-16. Watchpoint Trigger Register Fields

Bits	Name	Description
31–28	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	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	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
19–16	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 12-10, is used for extended program trace start trigger control.

0	PTST
31 30 29 28 27 26 25 24 23 22	21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Nexus Reg# - 0x35; Read/Write; Reset - 0x0

Figure 12-10. Program Trace Start Trigger Control (PTSTC) Register

Table 12-17 details the PTSTC register fields.

Table 12-17. Program Trace Start Trigger Control Register Fields

Bits	Name	Description
31–22	—	Reserved for future functionality (read as 0)
21–0	PTST	Program Trace Start Trigger Control 0000000000000000000000Trigger disabled XXXXXXXXXXXXXXXXXXXXX1Use Watchpoint #0 XXXXXXXXXXXXXXXXXXXX1XUse Watchpoint #1 XXXXXXXXXXXXXXXXXXX1XXUse Watchpoint #2 XXXXXXXXXXXXXXXXXX1XXXUse Watchpoint #3 XXXXXXXXXXXXXXXXX1XXXXUse Watchpoint #4 XXXXXXXXXXXXXXXX1XXXXXUse Watchpoint #5 XXXXXXXXXXXXXXX1XXXXXXUse Watchpoint #6 XXXXXXXXXXXXXX1XXXXXXXUse Watchpoint #7 XXXXXXXXXXXXX1XXXXXXXXUse Watchpoint #8 XXXXXXXXXXXX1XXXXXXXXXUse Watchpoint #9 XXXXXXXXXXX1XXXXXXXXXXUse Watchpoint #10 XXXXXXXXXX1XXXXXXXXXXXUse Watchpoint #11 XXXXXXXXX1XXXXXXXXXXXXUse Watchpoint #12 XXXXXXXX1XXXXXXXXXXXXXUse Watchpoint #13 XXXXXXX1XXXXXXXXXXXXXXUse Watchpoint #14 XXXXXX1XXXXXXXXXXXXXXXUse Watchpoint #15 XXXXX1XXXXXXXXXXXXXXXXUse Watchpoint #16 XXXX1XXXXXXXXXXXXXXXXXUse Watchpoint #17 XXX1XXXXXXXXXXXXXXXXXXUse Watchpoint #18 XX1XXXXXXXXXXXXXXXXXXXUse Watchpoint #19 X1XXXXXXXXXXXXXXXXXXXXUse Watchpoint #20 1XXXXXXXXXXXXXXXXXXXXXUse Watchpoint #21

The PTETC register, shown in Figure 12-11, is used for extended program trace end trigger control.

0	PTET
31 30 29 28 27 26 25 24 23 22	21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Nexus Reg#0x36; Read/Write; Reset - 0x0

Figure 12-11. Program Trace End Trigger Control (PTETC) Register

Table 12-18 details the PTETC register fields.

Table 12-18. Program Trace End Trigger Control Register Fields

Bits	Name	Description
31–22	—	Reserved for future functionality (read as 0)
21–0	PTET	Program Trace End Trigger Control 0000000000000000000000 Trigger disabled XXXXXXXXXXXXXXXXXXXXX1Use Watchpoint #0 XXXXXXXXXXXXXXXXXXXX1XUse Watchpoint #1 XXXXXXXXXXXXXXXXXXX1XXUse Watchpoint #2 XXXXXXXXXXXXXXXXXX1XXXUse Watchpoint #3 XXXXXXXXXXXXXXXXX1XXXXUse Watchpoint #4 XXXXXXXXXXXXXXXX1XXXXXUse Watchpoint #5 XXXXXXXXXXXXXXX1XXXXXXUse Watchpoint #6 XXXXXXXXXXXXXX1XXXXXXXUse Watchpoint #7 XXXXXXXXXXXXX1XXXXXXXXUse Watchpoint #8 XXXXXXXXXXXX1XXXXXXXXXUse Watchpoint #9 XXXXXXXXXXX1XXXXXXXXXXUse Watchpoint #10 XXXXXXXXXX1XXXXXXXXXXXUse Watchpoint #11 XXXXXXXXX1XXXXXXXXXXXXUse Watchpoint #12 XXXXXXXX1XXXXXXXXXXXXXUse Watchpoint #13 XXXXXXX1XXXXXXXXXXXXXXUse Watchpoint #14 XXXXXX1XXXXXXXXXXXXXXXUse Watchpoint #15 XXXXX1XXXXXXXXXXXXXXXXUse Watchpoint #16 XXXX1XXXXXXXXXXXXXXXXXUse Watchpoint #17 XXX1XXXXXXXXXXXXXXXXXXUse Watchpoint #18 XX1XXXXXXXXXXXXXXXXXXXUse Watchpoint #19 X1XXXXXXXXXXXXXXXXXXXXUse Watchpoint #20 1XXXXXXXXXXXXXXXXXXXXXUse Watchpoint #21

The DTSTC register, shown in Figure 12-12, is used for extended data trace start trigger control.

0										DTST																					
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Nexus Reg# - 0x37; Read/Write; Reset - 0x0

Figure 12-12. Data Trace Start Trigger Control (DTSTC) Register

Table 12-17 details the DTSTC register fields.

Table 12-19. Data Trace Start Trigger Control Register Fields

Bits	Name	Description
31–22	—	Reserved for future functionality (read as 0)
21–0	DTST	Data Trace Start Trigger Control 0000000000000000000000 Trigger disabled XXXXXXXXXXXXXXXXXXXXX1Use Watchpoint #0 XXXXXXXXXXXXXXXXXXXX1XUse Watchpoint #1 XXXXXXXXXXXXXXXXXXX1XXUse Watchpoint #2 XXXXXXXXXXXXXXXXXX1XXXUse Watchpoint #3 XXXXXXXXXXXXXXXXX1XXXXUse Watchpoint #4 XXXXXXXXXXXXXXXX1XXXXXUse Watchpoint #5 XXXXXXXXXXXXXXX1XXXXXXUse Watchpoint #6 XXXXXXXXXXXXXX1XXXXXXXUse Watchpoint #7 XXXXXXXXXXXXX1XXXXXXXXUse Watchpoint #8 XXXXXXXXXXXX1XXXXXXXXXUse Watchpoint #9 XXXXXXXXXXX1XXXXXXXXXXUse Watchpoint #10 XXXXXXXXXX1XXXXXXXXXXXUse Watchpoint #11 XXXXXXXXX1XXXXXXXXXXXXUse Watchpoint #12 XXXXXXXX1XXXXXXXXXXXXXUse Watchpoint #13 XXXXXXX1XXXXXXXXXXXXXXUse Watchpoint #14 XXXXXX1XXXXXXXXXXXXXXXUse Watchpoint #15 XXXXX1XXXXXXXXXXXXXXXXUse Watchpoint #16 XXXX1XXXXXXXXXXXXXXXXXUse Watchpoint #17 XXX1XXXXXXXXXXXXXXXXXXUse Watchpoint #18 XX1XXXXXXXXXXXXXXXXXXXUse Watchpoint #19 X1XXXXXXXXXXXXXXXXXXXXUse Watchpoint #20 1XXXXXXXXXXXXXXXXXXXXXUse Watchpoint #21

The DTETC register, shown in Figure 12-13, is used for extended data trace end trigger control.

0	DTET
31 30 29 28 27 26 25 24 23 22	21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Nexus Reg# - 038; Read/Write; Reset - 0x0

Figure 12-13. Data Trace End Trigger Control (DTETC) Register

Table 12-18 details the DTETC register fields.

Table 12-20. Data Trace End Trigger Control Register Fields

Bits	Name	Description
31–22	—	Reserved for future functionality (read as 0)
21–0	DTET	Data Trace End Trigger Control 0000000000000000000000 Trigger disabled XXXXXXXXXXXXXXXXXXXXX1Use Watchpoint #0 XXXXXXXXXXXXXXXXXXXX1XUse Watchpoint #1 XXXXXXXXXXXXXXXXXXX1XXUse Watchpoint #2 XXXXXXXXXXXXXXXXXX1XXXUse Watchpoint #3 XXXXXXXXXXXXXXXXX1XXXXUse Watchpoint #4 XXXXXXXXXXXXXXXX1XXXXXUse Watchpoint #5 XXXXXXXXXXXXXXX1XXXXXXUse Watchpoint #6 XXXXXXXXXXXXXX1XXXXXXXUse Watchpoint #7 XXXXXXXXXXXXX1XXXXXXXXUse Watchpoint #8 XXXXXXXXXXXX1XXXXXXXXXUse Watchpoint #9 XXXXXXXXXXX1XXXXXXXXXXUse Watchpoint #10 XXXXXXXXXX1XXXXXXXXXXXUse Watchpoint #11 XXXXXXXXX1XXXXXXXXXXXXUse Watchpoint #12 XXXXXXXX1XXXXXXXXXXXXXUse Watchpoint #13 XXXXXXX1XXXXXXXXXXXXXXUse Watchpoint #14 XXXXXX1XXXXXXXXXXXXXXXUse Watchpoint #15 XXXXX1XXXXXXXXXXXXXXXXUse Watchpoint #16 XXXX1XXXXXXXXXXXXXXXXXUse Watchpoint #17 XXX1XXXXXXXXXXXXXXXXXXUse Watchpoint #18 XX1XXXXXXXXXXXXXXXXXXXUse Watchpoint #19 X1XXXXXXXXXXXXXXXXXXXXUse Watchpoint #20 1XXXXXXXXXXXXXXXXXXXXXUse Watchpoint #21

12.4.8 Nexus Watchpoint Mask Register (WMSK)

The Nexus watchpoint mask register, shown in Figure 12-14, controls which watchpoint events are enabled to produce watchpoint trace messages. DC1[TM] must also be programmed to generate watchpoint trace messages.

0	WEM
31 30 29 28 27 26 25 24 23 22	21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Nexus Reg# - 0x33; Read/Write; Reset - 0x0

Figure 12-14. Watchpoint Mask Register

Table 12-16 details the Watchpoint Trigger register fields.

Table 12-21. Watchpoint Mask Register Fields

Bits	Name	Description
31–22	—	Reserved for future functionality (read as 0)
21–0	WEM	Watchpoint Enable for Messaging 0000000000000000000000 No Watchpoints enabled for Watchpoint Trace Messaging XXXXXXXXXXXXXXXXXXXXX1Watchpoint #0 enabled for WTM XXXXXXXXXXXXXXXXXXXX1XWatchpoint #1 enabled for WTM XXXXXXXXXXXXXXXXXXX1XXWatchpoint #2 enabled for WTM XXXXXXXXXXXXXXXXXX1XXXWatchpoint #3 enabled for WTM XXXXXXXXXXXXXXXXX1XXXXWatchpoint #4 enabled for WTM XXXXXXXXXXXXXXXX1XXXXXWatchpoint #5 enabled for WTM XXXXXXXXXXXXXXX1XXXXXXWatchpoint #6 enabled for WTM XXXXXXXXXXXXXX1XXXXXXXWatchpoint #7 enabled for WTM XXXXXXXXXXXXX1XXXXXXXXWatchpoint #8 enabled for WTM XXXXXXXXXXXX1XXXXXXXXXWatchpoint #9 enabled for WTM XXXXXXXXXXX1XXXXXXXXXXWatchpoint #10 enabled for WTM XXXXXXXXXX1XXXXXXXXXXXWatchpoint #11 enabled for WTM XXXXXXXXX1XXXXXXXXXXXXWatchpoint #12 enabled for WTM XXXXXXXX1XXXXXXXXXXXXXWatchpoint #13 enabled for WTM XXXXXXX1XXXXXXXXXXXXXXWatchpoint #14 enabled for WTM XXXXXX1XXXXXXXXXXXXXXXWatchpoint #15 enabled for WTM XXXXX1XXXXXXXXXXXXXXXXWatchpoint #16 enabled for WTM XXXX1XXXXXXXXXXXXXXXXXWatchpoint #17 enabled for WTM XXX1XXXXXXXXXXXXXXXXXXWatchpoint #18 enabled for WTM XX1XXXXXXXXXXXXXXXXXXXWatchpoint #19 enabled for WTM X1XXXXXXXXXXXXXXXXXXXXWatchpoint #20 enabled for WTM 1XXXXXXXXXXXXXXXXXXXXXWatchpoint #21 enabled for WTM

12.4.9 Nexus Overrun Control Register (OVCR)

The Nexus overrun control register, shown in Figure 12-15, controls Nexus behavior as the internal message queues fill up. Response options include suppressing selected message types or stalling processor instruction execution.

0	SPTHOLD	0	SPEN	0	STTHOLD	0	STEN
31 30	29 28	27 26 25 24 23 22	21 20 19 18 17 16	15 14	13 12	11 10 9 8 7 6 5 4 3 2 1	0

Nexus Reg# - 0x32; Read/Write; Reset - 0x0

Figure 12-15. Nexus Overrun Control Register

Table 12-16 shows the Nexus overrun control register fields.

Table 12-22. 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 1/4 full 01 Suppression threshold is when message queues are 1/2 full 10 Suppression threshold is when message queues are 3/4 full 11 Reserved
27–22	—	Reserved, should be cleared
21–16	SPEN	Suppression Enable 000000 Suppression is disabled xxxxx1 Ownership Trace message suppression is enabled xxxx1x Data Trace message suppression is enabled xxx1xx Program Trace message suppression is enabled xx1xxx Watchpoint Trace message suppression is enabled x1xxxx Reserved 1xxxxx Data Acquisition message suppression is enabled
15–14	—	Reserved, should be cleared
13–12	STTHOLD	Stall Threshold 00 Stall threshold is when message queues are 1/4 full 01 Stall threshold is when message queues are 1/2 full 10 Stall threshold is when message queues are 3/4 full 11 Reserved
11–1	—	Reserved, should be cleared
0	STEN	Stall Enable 0 Stalling is disabled 1 Stalling is enabled

12.4.10 Data Trace Control Register (DTC)

The data trace control registercontrols 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.

Figure 12-16 shows the data trace control register.

RWT1	RWT2	RWT3	RWT4	0	RC1	RC2	RC3	RC4	DI	0
31 30	29 28	27 26	25 24	23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8	7	6	5	4	3	2 1 0

Nexus Reg# - 0xD; Read/Write; Reset - 0x0

Figure 12-16. Data Trace Control Register

Table 12-23 details the data trace control register fields.

Table 12-23. Data Trace Control Register Fields

Bits	Name	Description
31–30	RWT1	Read/Write Trace 1 00 No trace enabled X1 Enable Data Read Trace 1X Enable Data Write Trace
29–28	RWT2	Read/Write Trace 2 00 No trace enabled X1 Enable Data Read Trace 1X Enable Data Write Trace
27–26	RWT3	Read/Write Trace 3 00 No trace enabled X1 Enable Data Read Trace 1X Enable Data Write Trace
25–24	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	Range Control 1 0 Condition trace on address within range 1 Condition trace on address outside of range
6	RC2	Range Control 2 0 Condition trace on address within range 1 Condition trace on address outside of range
5	RC3	Range Control 3 0 Condition trace on address within range 1 Condition trace on address outside of range
4	RC4	Range Control 4 0 Condition trace on address within range 1 Condition trace on address outside of range
3	DI	Data Access/Instruction Access Trace 0 Condition trace on data accesses 1 Condition trace on instruction accesses
2–0	—	Reserved for future functionality (read as 0)

12.4.11 Data Trace Start Address Registers (DTSA1–4)

Figure 12-17–Figure 12-20 show the data trace start address registers, which define the start addresses for each trace channel.

Figure 12-17. Data Trace Start Address 1 Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Nexus Reg#0xE; Read/Write; Reset - 0x0

Figure 12-17. Data Trace Start Address 1 Register

.

Data Trace Start Address																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Nexus Reg# - 0xF; Read/Write; Reset - 0x0

Figure 12-18. Data Trace Start Address 2 Register

..

Data Trace Start Address																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Nexus Reg# - 0x10; Read/Write; Reset - 0x0

Figure 12-19. Data Trace Start Address 3 Register

Data Trace Start Address																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Nexus Reg# - 0x11; Read/Write; Reset - 0x0

Figure 12-20. Data Trace Start Address 4 Register

12.4.12 Data Trace End Address Registers (DTEA1–4)

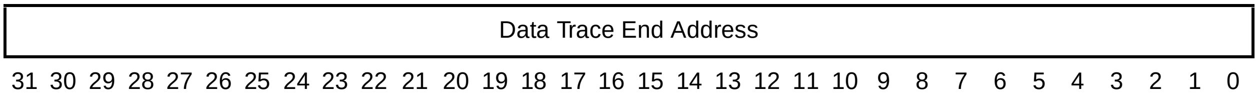
Figure 12-20–Figure 12-24 show the data trace end address registers, which define the end addresses for each trace channel.

Data Trace End Address																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Nexus Reg# - 0x12; Read/Write; Reset - 0x0

Figure 12-21. Data Trace End Address 1 Register

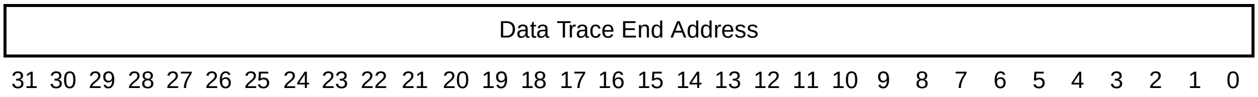
.



Nexus Reg# - 0x13; Read/Write; Reset - 0x0

Figure 12-22. Data Trace End Address 2 Register

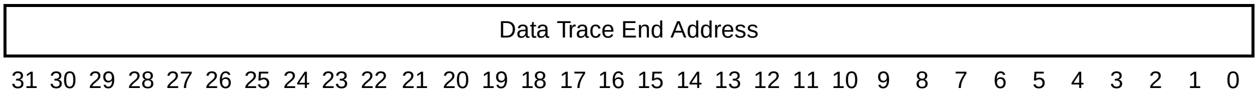
.



Nexus Reg# - 0x14; Read/Write; Reset - 0x0

Figure 12-23. Data Trace End Address 3 Register

.



Nexus Reg# - 0x15; Read/Write; Reset - 0x0

Figure 12-24. Data Trace End Address 4 Register

Table 12-24 illustrates the range that is selected for data trace for various cases of DTSA being less than, greater than, or equal to DTEA.

Table 12-24. 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 in order 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”, and are *exclusive* of the DTSA and DTEA addresses for Range Control settings indicating “outside of range”.

12.4.13 Read/Write Access Control/Status (RWCS)

The read write access control/status register 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. The RWCS register also provides read/write access status information per Table 12-26.

Figure 12-25 shows the read/write access control/status register.

AC	RW	SZ			MAP			PR		0						CNT														ERR	DV
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Nexus Reg# - 0x7; Read/Write[1]; Reset - 0x0

Figure 12-25. Read/Write Access Control/Status Register

[1] ERR and DV are read-only

Table 12-25 shows the RWCS fields.

Table 12-25. Read/Write Access Control/Status Register Fields

Bits	Name	Description
RWCS[31]	AC	Access Control 0 End access 1 Start access
RWCS[30]	RW	Read/Write Select 0 Read access 1 Write access
RWCS[29–27]	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 Select 000 Primary memory map 001–111 Reserved
RWCS[23–22]	PR[1]	Read/Write Access Priority 00 Reserved (default to highest access priority) 01 Reserved (default to highest access priority) 10 Reserved (default to highest access priority) 11 Highest access priority
RWCS[21–16]	—	Reserved for future functionality
RWCS[15–2]	CNT	Access Control Count hhhh Number of accesses of word size SZ
RWCS[1]	ERR[2]	Read/Write Access Error (see Table 12-26)
RWCS[0]	DV [2]	Read/Write Access Data Valid (see Table 12-26)

[1] The priority functionality is not currently implemented

[2] ERR and DV are read-only

Table 12-26 shows the read/write access status bit encoding.

Table 12-26. 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

12.4.14 Read/Write Access Data (RWD)

The read/write access data register (RWD) provides the data to/from system bus memory-mapped locations when initiating a read or a write access.

Figure 12-26 shows the read/write access data register.

Read/Write Data

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Nexus Reg# - 0xA; Read/Write; Reset - 0x0

Figure 12-26. 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 12-27 shows the proper placement of data into the RWD. Note that double-word transfers require two passes through RWD.

Table 12-27. RWD Data Placement For Transfers

Transfer Size and byte offset	RWA(2–0)	RWCS[SZ]	vRWD			
			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 12-27. RWD Data Placement For Transfers (Continued)

Transfer Size and byte offset	RWA(2–0)	RWCS[SZ]	vRWD			
			31–24	23–16	15–8	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

Table Notes:
“X” indicates byte lanes with valid data
“—” indicates byte lanes which will contain unused data.

Table 12-28 shows the mapping of RWD bytes to byte lanes of the AHB read and write data buses.

Table 12-28. RWD Byte Lane Mapping

Transfer Size and byte offset	RWA(2–0)	RWD			
		31–24	23–16	15–8	7–0
Byte at 000	0 0 0	—	—	—	AHB[7–0]
Byte at 001	0 0 1	—	—	—	AHB[15–8]
Byte at 010	0 1 0	—	—	—	AHB[23–16]
Byte at 011	0 1 1	—	—	—	AHB[31–24]
Byte at 100	1 0 0	—	—	—	AHB[39–32]
Byte at 101	1 0 1	—	—	—	AHB[47–40]
Byte at 110	1 1 0	—	—	—	AHB[55–48]
Byte at 111	1 1 1	—	—	—	AHB[63–56]
Half at 000	0 0 0	—	—	AHB[15–8]	AHB[7–0]
Half at 010	0 1 0	—	—	AHB[31–24]	AHB[23–16]
Half at 100	1 0 0	—	—	AHB[47–40]	AHB[39–32]
Half at 110	1 1 0	—	—	AHB[63–56]	AHB[55–48]
Word at 000	0 0 0	AHB[31–24]	AHB[23–16]	AHB[15–8]	AHB[7–0]
Word at 100	1 0 0	AHB[63–56]	AHB[55–48]	AHB[47–40]	AHB[39–32]
Double word at 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]

Table Notes–
“—” indicat es byte lanes which will contain unused dat a.

12.4.15 Read/Write Access Address (RWA)

The read/write access address register, shown in Figure 12-27, provides the system bus address to be accessed when initiating a read or a write access.

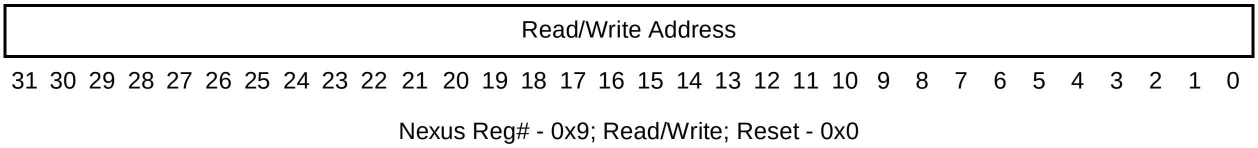


Figure 12-27. Read/Write Access Address Register

12.5 Nexus 3+ Register Access via JTAG/OnCE

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 12-8.

Reading/writing of a Nexus 3+ register then requires two passes through the data-scan (DR) path of the JTAG state machine (see Section 12.21, "IEEE 1149.1 (JTAG) RD/WR Sequences").

1. The first pass through the DR selects the Nexus 3+ register to be accessed by providing an index (see Table 12-8), 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:

(7bits)	(1 bit)
Nexus Register Index	R/W

RESET Value: 0x00

Nexus Register Index:	Selected from values in Table 12-8
Read/Write (R/W):	0 Read 1 Write

2. The second pass through the DR then shifts the data in or out of the JTAG port, LSB first.
   - During a read access, data is latched from the selected Nexus register when the JTAG state machine passes through the "Capture-DR" state.
   - During a write access, data is latched into the selected Nexus register when the JTAG state machine passes through the "Update-DR" state.

12.6 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 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.

12.6.1 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.

12.6.2 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. Multi-threaded processors may assign additional source ID information to indicate which thread a message is associated with. The e200z446n3 Nexus 3+ module implements a 4-bit fixed length Source ID field consisting of a Client Source ID.

12.6.3 Relative Address Field (U-ADDR)

The non-sync forms of the program and data trace messages include addresses that are relative to the address transmitted in the previous program or data trace message respectively. The relative address format conforms to the IEEE-ISTO 5001 standard 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 12-28 shows the relative address generation and recreation.

Previous Address (A1) =0x0003FC01, New Address (A2) = 0x0003F365

```
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 12-28. Relative Address Generation and Recreation

12.6.4 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.

12.6.5 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 only occurs on reset, on an exception, or by an **mtmsr**, **rfi**, **rfci**, **rfdi**, or **rfmci** instruction. A potential change in address space by an exception or by an **rfi**, **rfci**, **rfdi**, or **rfmci** instruction causes 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 by an **mtmsr** instruction causes 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**).

12.7 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:

- 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.

12.7.1 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 which contains information about the types of messages that were discarded due to the overrun condition.

12.7.2 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 1/4 full, the stall stays in effect until the queue utilization drops to empty. There may be significant skid from the time that the stall request is made until the processor is able to stop completing instructions. This skid should be taken into consideration when programming the threshold. Refer to Section 12.4.9, “Nexus Overrun Control Register (OVCR),” for complete programming options.

12.7.3 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, possibly avoiding an overrun condition. If an overrun condition occurs despite this message suppression, the queue responds according to the behavior described in Section 12.7.1, “Message Queue Overrun.” Once triggered, message suppression remains in effect until queue utilization drops to the threshold below the selected trigger suppression level.

12.7.4 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 12-29.

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 are combined.

Table 12-29 lists the various message types and their relative priority from highest to lowest.

Table 12-29. 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)	PCMPID or mtmsr IS update (Program Correlation Message)	2	OTM Error Message
OT (Ownership)	OTMPID 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

Table 12-29. Message Type Priority and Message Dropped Responses

Message Type	Message	Priority	Message Dropped Response
DT (Data Trace)	DTM (Data Trace Message)	6	Sync upgrade next DTM
OT (Ownership)	OTMPeriodic 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.

12.7.5 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.

12.7.6 Ownership Trace Message Priority Loss Response

If software updates to the Process ID state cause an ownership trace message (OTM) to lose arbitration due to contention with higher priority messages—other than a program correlation message with EVCODE = 0101 (PID or MSR[IS] update)—an error message is generated to indicate that an OTM has been lost due to contention. If the pending OTM is a periodic update, the event is dropped without generating an error message.

12.7.7 Program Trace Message Priority Loss Response

If a program trace message (PTM) loses arbitration due to contention with higher priority messages, and the discarded PTM is a program correlation message, a resource full message for instruction count/history buffer, or a Branch Trace message, then an error message is generated to indicate that branch trace information has been lost, and 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 will indicate a dropped OTM and a dropped program trace (error code = xxxx11xx).

12.7.8 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.

12.8 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 12-29:

(8 bits)	(4 bits)	(6 bits)
DS[31:24]	Src. Proc.	TCODE (000000)

Fixed length = 18 bits

Figure 12-29. Debug Status Message Format

12.9 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 will 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 as shown in Figure 12-30 (also see Table 12-3 and Table 12-4).

(6 bits)	(4 bits)	(4 bits)	(6 bits)
Error Code	Error Type	Src. Proc.	TCODE (001000)

Fixed length = 20 bits

Figure 12-30. Error Message Format

12.10 Ownership Trace

This section details the ownership trace features of the Nexus 3+ module.

12.10.1 Overview

Ownership trace provides a macroscopic view 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.

12.10.2 Ownership Trace Messaging (OTM)

Ownership trace information is messaged by the auxiliary port using an ownership trace message (OTM). The e200 processors contain a Power ISA defined “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); thus, the Nexus UBA functionality is not implemented.

There are two conditions that cause an ownership trace message when ownership trace is enabled:

- When new information is updated in the PID0 register by the e200 processor, the data is latched within Nexus and messaged out by 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, since the program correlation message will provide this PID0 update information.
- Periodically, at least once every 256 messages, the most recent state of the PID0 register is messaged out. The resulting ownership trace message indicates in the PID index subfield that PID0 status is being reported and the most recent value of the PID0 register is conveyed in the Process ID value subfield. These periodic ownership trace message events can be disabled by setting DC1[POTD].

Ownership trace information is messaged out as shown in Figure 12-31.

(1–8 bits)	(4 bits)	(4 bits)	(6 bits)
Process ID	PID Index (0000)	Src. Proc.	TCODE (000010)

Variable length = 15–22 bits

Figure 12-31. Ownership Trace Message Format

12.11 Program Trace

This section details the program trace mechanism supported by Nexus 3+ for the e200 processor. Program trace is implemented through branch trace messaging (BTM) as per the IEEE-ISTO 5001-2008 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]**).

12.11.1 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 that 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 will reflect the current value of msr[is].

Branch history messaging facilitates program trace by providing the following information.

- Messaging for taken indirect branches and exceptions includes:
  - 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 which are both reported as consecutive bits in the history field. Not-taken indirect branches generate a history bit with a value of zero. Instructions that generate history bits are not included in instruction counts. For taken indirect branches which 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 the MSR[IS] bit 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 reflect the current value of MSR[IS].

12.11.1.1 e200 Indirect Branch Message Instructions

Table 12-30 shows the types of instructions and events that cause indirect branch messages or branch history messages to be encoded.

Table 12-30. 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**
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

12.11.1.2 e200 Direct Branch Message Instructions

Table 12-31 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 12-31. 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**

12.11.1.3 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.

12.11.1.4 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. Since 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 is messaged out maintaining that order.

12.11.2 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.

12.11.2.1 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 following format:

(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 12-32. Indirect Branch Message (History) Format

12.11.2.2 Indirect Branch Messages (Traditional)

If DC1[PTM] is cleared, indirect branch information is messaged out in the following format:

(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 12-33. Indirect Branch Message Format

12.11.2.3 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 in the following format:

(1-8 bits)	(4 bits)	(6 bits)
Sequence Count	Src. Proc.	TCODE (000011)

Max length = 18 bits; Min length = 11bits

Figure 12-34. 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.

12.11.3 Program Trace Message Fields

The following subsections describe specific fields used for program trace messages.

12.11.3.1 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.

12.11.3.2 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, which acts as a stop bit (the most significant 1 in the history field is a termination bit for the field). The pre-loaded 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 12-32 shows the branch/predicate history events.

Table 12-32. 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

Table 12-32. Branch/Predicate History Events

Branch/Predicate History Event	History Bit(s)	Relevant Instructions
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 generate program correlation messages and are not logged in the history buffer.

12.11.3.3 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 12.11.5, "Program Correlation Messages," for more information on these cases.

12.11.4 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 the 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. This indicates 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 12-35 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 12-35. Resource Full Message Format

Table 12-33 shows the RCODE encodings and RDATA information used for resource full messages.

Table 12-33. 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.

12.11.5 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
  — 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
  — 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
  — 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**/**bcla**-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
  — 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 and a direct branch results in an execution mode switch into or out of a sequence of VLE instructions
  - 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 a new address translation is established in the TLB by an **tlbwe** instruction.
- When address translation(s) are invalidated in the TLB by an **tlbivax** instruction.
- When a new instruction address space setting (IS) is established in the MSR by an **mtmsr** instruction.
- When an update to the process ID register (PID0) is made by an **mtspr** PID0.

Refer to Table 12-6 for the event codes that are supported in this implementation. Event code masking is available by the EVCDM field of the DC4 register to allow for control over generation of program correlation messages for each event type.

Program correlation is messaged out as shown in Figure 12-36:

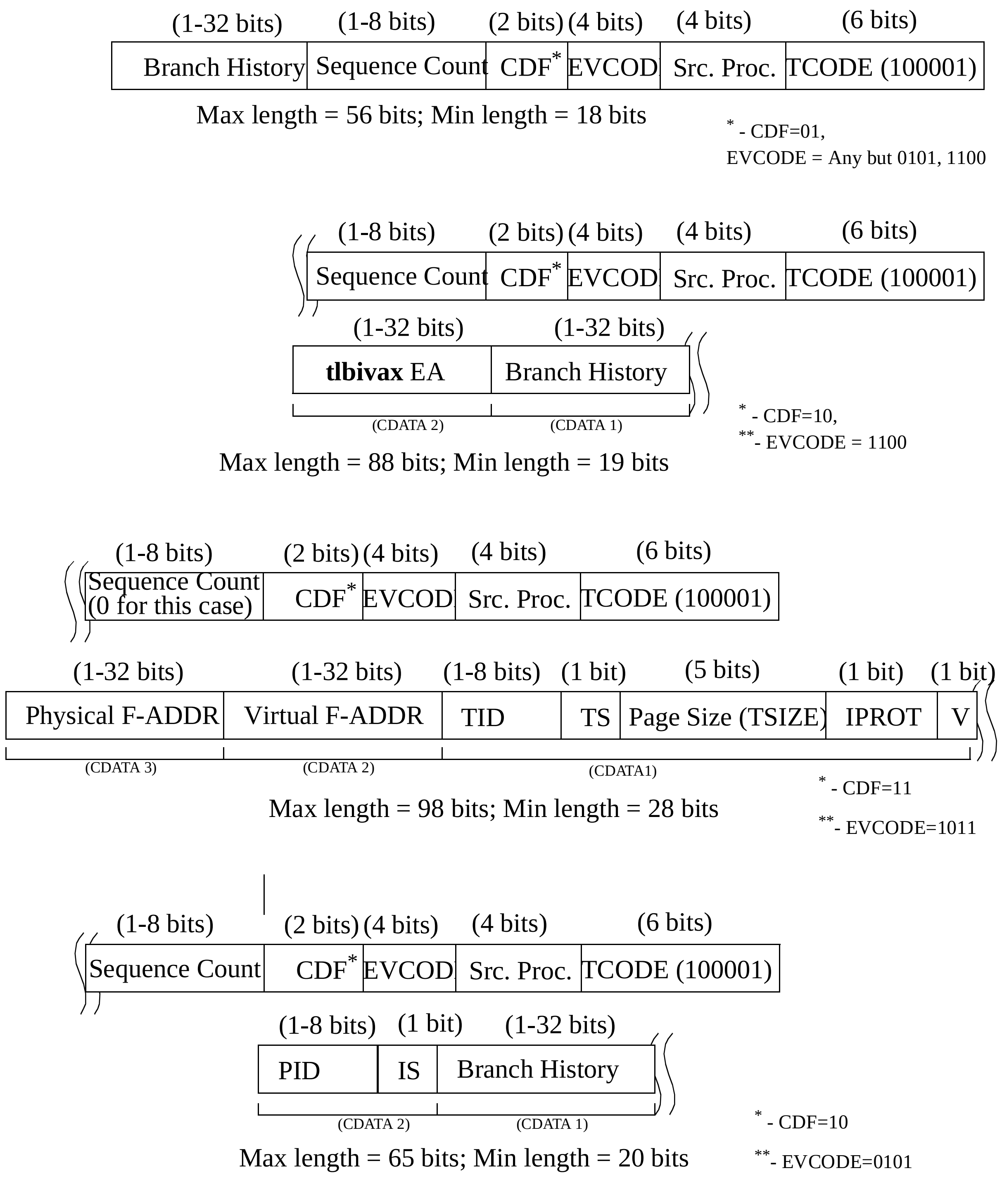


Figure 12-36. Program Correlation Message Formats

12.11.5.1 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.

12.11.5.2 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.

12.11.5.3 Program Correlation Message Generation for PID Updates or MSR[IS] Updates

When a (potentially) new value is established in the PID by an **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] by an **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.

12.11.6 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 will discard 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.

12.11.7 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 by the auxiliary port (provided program trace is enabled) for the following conditions (see Table 12-34):

- 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.

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 12-37.

(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 12-37. Direct/Indirect Branch with Sync Message Format

The format for program trace indirect branch history with sync messages is as as shown in Figure 12-38.

(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 12-38. Indirect Branch History w/ Sync. Message Format

Exception conditions that result in program trace synchronization are summarized in Table 12-34.

Table 12-34. 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 w/ Sync. 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.
Collision Priority	All Messages have the following priority: Instruction 0 (WPM -> DQM -> 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.

12.11.8 Enabling Program Trace

Program Trace Messaging can be enabled in one of two 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)

12.11.9 Program Trace Timing Diagrams (2 MDO/1 MSEO configuration)

Figure 12-39–Figure 12-42 shows the program trace timing diagrams.

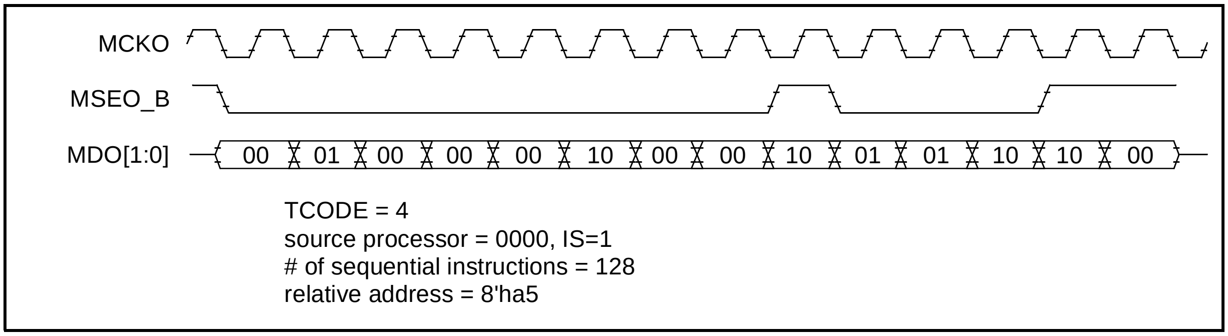


Figure 12-39. Program Trace—Indirect Branch Message (Traditional)

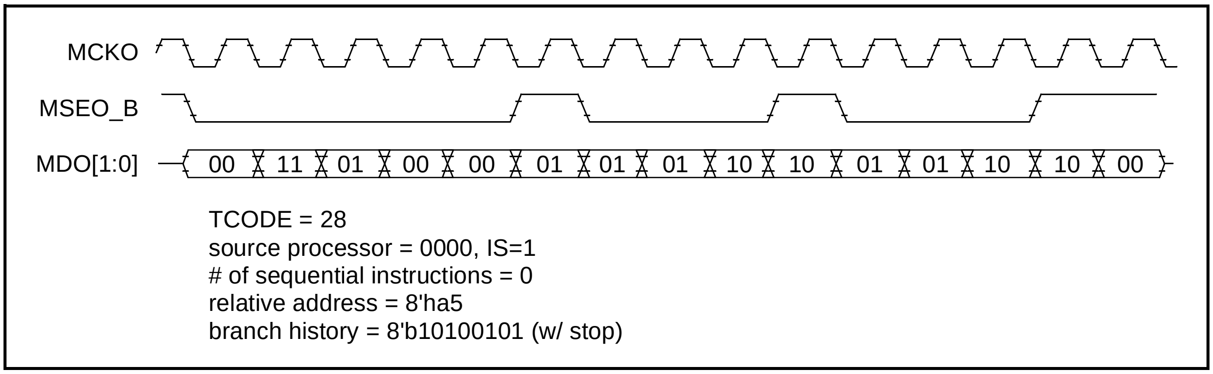
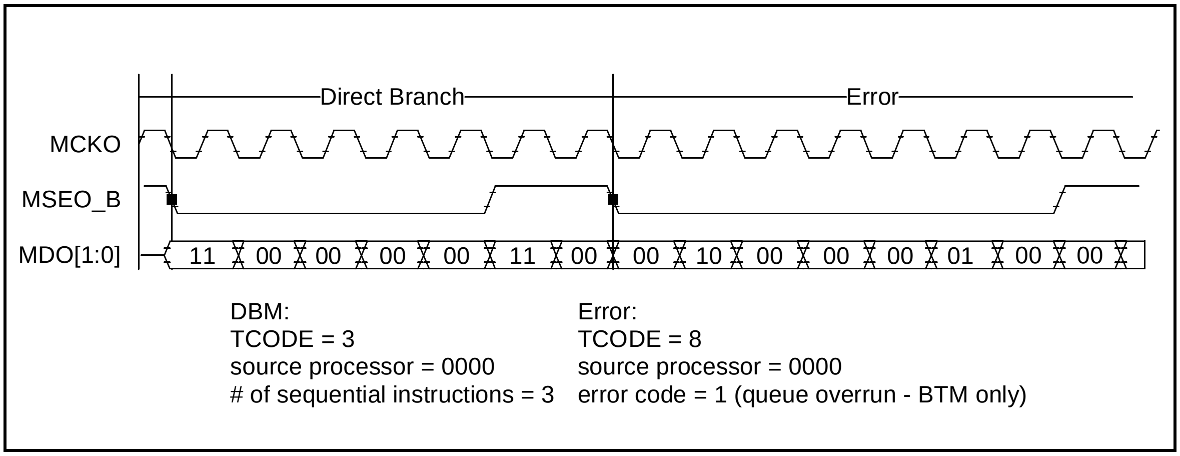

MCKO
MSEO_B
MDO[1:0]
00 11 01 00 00 01 01 01 10 10 01 01 10 10 00
TCODE = 28
source processor = 0000, IS=1
# of sequential instructions = 0
relative address = 8'ha5
branch history = 8'b10100101 (w/ stop)

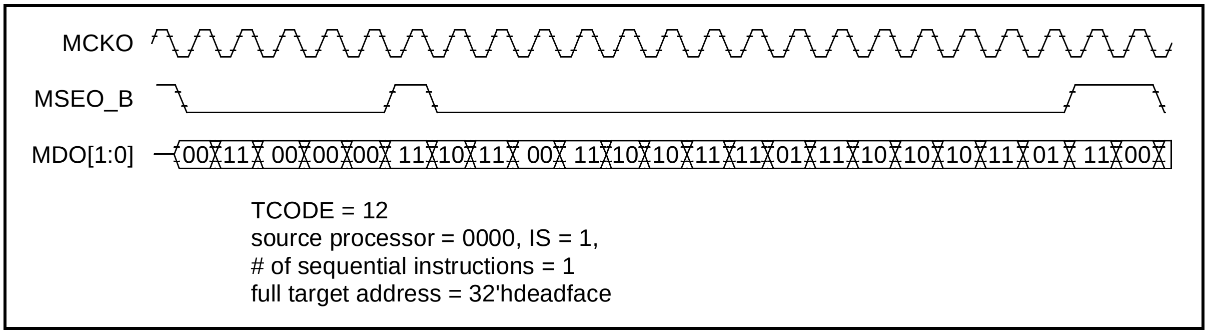
Figure 12-40. Program Trace—Indirect Branch Message (History)



DBM:
TCODE = 3
source processor = 0000
# of sequential instructions = 3

Error:
TCODE = 8
source processor = 0000
error code = 1 (queue overrun - BTM only)

Figure 12-41. Program Trace—Direct Branch (Traditional) and Error Messages



TCODE = 12
source processor = 0000, IS = 1,
# of sequential instructions = 1
full target address = 32'hdeadface

Figure 12-42. Program Trace - Indirect Branch w/ Sync. Message

12.12 Data Trace

This section deals with the data trace mechanism supported by the Nexus 3+ module. Data trace is implemented by data write messaging (DWM) and data read messaging (DRM), as per the IEEE-ISTO 5001-2008 standard.

12.12.1 Data Trace Messaging (DTM)

Data trace messaging for the 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 bus. This allows for data visibility for e200 processors which incorporate a data cache. Only e200 CPU initiated accesses will be traced. No DMA accesses to the AHB system bus will be 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 Nexus 1 module).

12.12.2 DTM Message Formats

The Nexus 3+ block supports the following five types of DTM messages:

- Data Write
- Data Read
- Data Write Synchronization
- Data Read Synchronization
- Error Messages

12.12.2.1 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 as shown in Figure 12-43.

(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 12-43. Data Write Message Format

12.12.2.2 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 as shown in Figure 12-44.

(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 12-44. Data Read Message Format

NOTE

e200 based CPUs are capable of generating two reads or writes per clock cycle in cases where multiple registers are accessed with a single instruction (**lmw/stmw**). These will have a double word pair size encoding (**p_tsiz** = 0b000). In these cases, the Nexus 3+ module sends one 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 will be 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 e200 processor.

12.12.2.3 Data Trace Synchronization Messages

A data trace write/read with synchronization message is messaged by the auxiliary port (provided data trace is enabled) for the following conditions (see Table 12-35):

- 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 will be 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 is shown in Figure 12-45.

(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 12-45. Data Write/Read with Synchronization Message Format

Exception conditions that result in data trace Synchronization are summarized in Table 12-35.

Table 12-35. 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 w/ Sync. 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 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 DTM message in the queue will be a Data Write/Read w/ Sync. Message.
Periodic Data Trace Sync.	A forced synchronization occurs periodically after 255 Data Trace Messages have been queued. A Data Write/Read w/ Sync. 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 w/ Sync. 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 SOC's which implement security, any attempted read or write to secure memory locations will temporarily disable Data Trace & cause the corresponding DTM to be lost. A subsequent read/write will queue a Data Trace Read/Write w/ Sync. Message.
Collision Priority	All Messages have the following priority: Instruction 0 (WPM → DQM → OTM → BTM → DTM)→ Instruction1 (WPM → DQM → PCM[PIDMSG] → OTM → BTM → DTM), where instruction0 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 w/ sync. message.

12.12.3 DTM Operation

The following sections explain DTM operation.

12.12.3.1 Data Trace Windowing

Data write/read messages are enabled by the RWT field in the data trace control register (DTC) for each DTM channel. Data trace windowing is achieved by 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.

12.12.3.2 Data Access/Instruction Access Data Tracing

The Nexus 3+ 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.

12.12.3.3 e200 Bus Cycle Special Cases

Table 12-36 shows the e200 bus cycle special cases.

Table 12-36. 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 Nexus3	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 DTM is 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 and a single DTM is 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 & 2nd cycle captured & a single DTM is 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 is sent with a size encoding indicating the size of the original access (such as word), and the address indicating the original misaligned accesses.

12.12.4 Data Trace Timing Diagrams(8 MDO/2 MSEO configuration)

Figure 12-46 shows the data write message timing diagram.

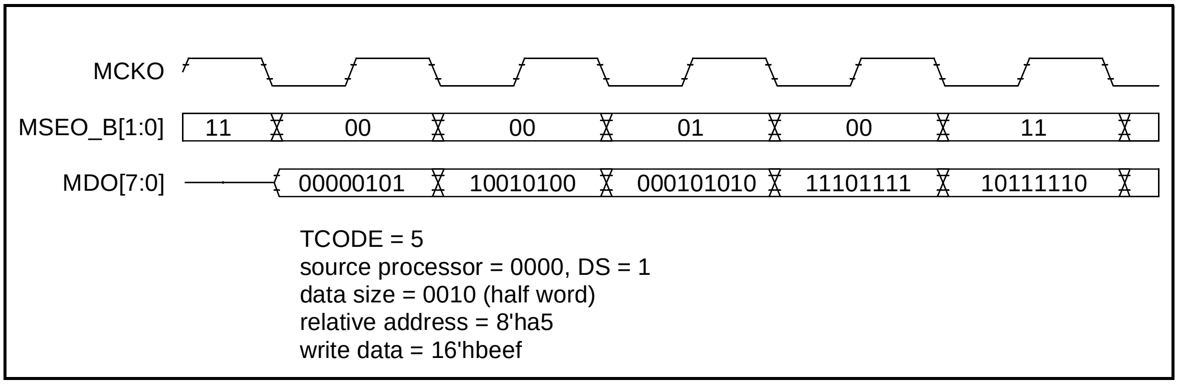


Figure 12-46. Data Trace—Data Write Message

Figure 12-47 shows the data read with synchornization message timing diagram.

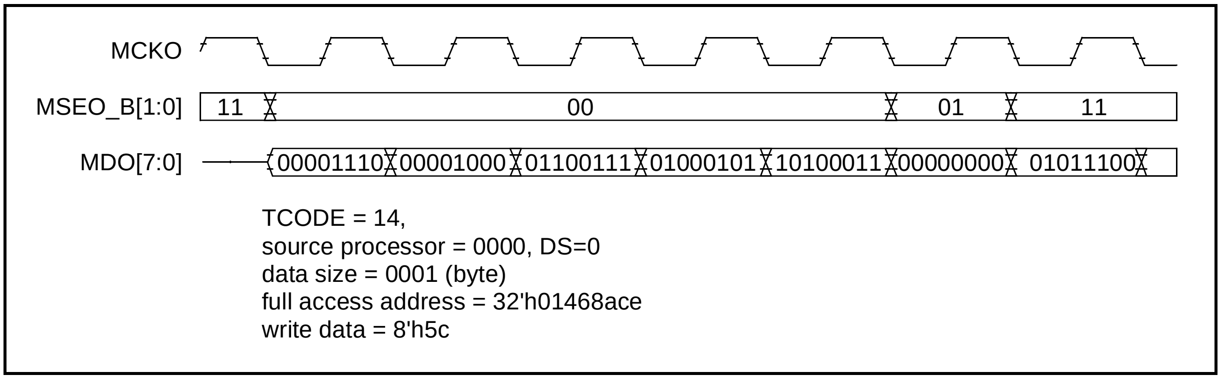


Figure 12-47. Data Trace—Data Read with Sync Message

12.13 Data Acquisition Messaging

This section details the data acquisition mechanisms supported by the e200z446n3 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.

12.13.1 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 by **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.

12.13.2 Data Acquisition Data Field

The data acquisition data field (DQDATA) is the data captured from the DDAM write operation by **mtspr** operations. Leading zeros are omitted from the message.

12.13.3 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 by control in DBERC0 while in external debug mode. Hardware always has access to these registers. Refer to Section 11.3.4, “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.

Figure 12-48 shows the data acquisition message format.

(1-32 bits)	(8 bits)	(4 bits)	(6 bits)
DQDATA	DQTAG	Src. Proc.	TCODE (011011)

Max length = 50 bits; Min length = 19 bits

Figure 12-48. Data Acquisition Message Format

12.14 Watchpoint Trace Messaging

Enabling watchpoint messaging is done by setting the watchpoint trace enable bit in the DC1 register. Setting the individual watchpoint sources is supported through the e200 Nexus 1 module. The e200 Nexus 1 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 settings of WMSK[WEM] 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 12-39 for details on **nex_evto_b**.

Watchpoint information is messaged out as shown in Figure 12-49.

(1-22 bits)	(4 bits)	(6 bits)
Watchpoint Source	Src. Proc.	TCODE (001111)

Variable length = 11-32 bits

Figure 12-49. Watchpoint Message Format.

Table 12-37 shows the watchpoint source encoding. The Watchpoint Source message field contains a ‘1’ for each asserted watchpoint. Leading zeros are truncated.

Table 12-37. Watchpoint Source Encoding

Watchpoint Source (1-22 bits)	Watchpoint Description
XXXXXXXXXXXXXXXXXXXXX1	Watchpoint #0 enabled for WTM
XXXXXXXXXXXXXXXXXXXX1X	Watchpoint #1 (I enabled for WTM
XXXXXXXXXXXXXXXXXXX1XX	Watchpoint #2 enabled for WTM
XXXXXXXXXXXXXXXXXX1XXX	Watchpoint #3 enabled for WTM
XXXXXXXXXXXXXXXXX1XXXX	Watchpoint #4 enabled for WTM
XXXXXXXXXXXXXXXX1XXXXX	Watchpoint #5 enabled for WTM
XXXXXXXXXXXXXXX1XXXXXX	Watchpoint #6 enabled for WTM
XXXXXXXXXXXXXX1XXXXXXX	Watchpoint #7 enabled for WTM
XXXXXXXXXXXXX1XXXXXXXX	Watchpoint #8 enabled for WTM
XXXXXXXXXXXX1XXXXXXXXX	Watchpoint #9 enabled for WTM
XXXXXXXXXXX1XXXXXXXXXX	Watchpoint #10 enabled for WTM
XXXXXXXXXX1XXXXXXXXXXX	Watchpoint #11 enabled for WTM
XXXXXXXXX1XXXXXXXXXXXX	Watchpoint #12 enabled for WTM
XXXXXXXX1XXXXXXXXXXXXX	Watchpoint #13 enabled for WTM
XXXXXXX1XXXXXXXXXXXXXX	Watchpoint #14 enabled for WTM
XXXXXX1XXXXXXXXXXXXXXX	Watchpoint #15 enabled for WTM
XXXXX1XXXXXXXXXXXXXXXX	Watchpoint #16 enabled for WTM
XXXX1XXXXXXXXXXXXXXXXX	Watchpoint #17 enabled for WTM
XXX1XXXXXXXXXXXXXXXXXX	Watchpoint #18 enabled for WTM
XX1XXXXXXXXXXXXXXXXXXX	Watchpoint #19 enabled for WTM
X1XXXXXXXXXXXXXXXXXXXX	Watchpoint #20 enabled for WTM
1XXXXXXXXXXXXXXXXXXXXX	Watchpoint #21 enabled for WTM

12.14.1 Watchpoint Timing Diagram (2 MDO/1 MSEO configuration)

Figure 12-50 shows the watchpoint message and watchpoint error message timing diagram.

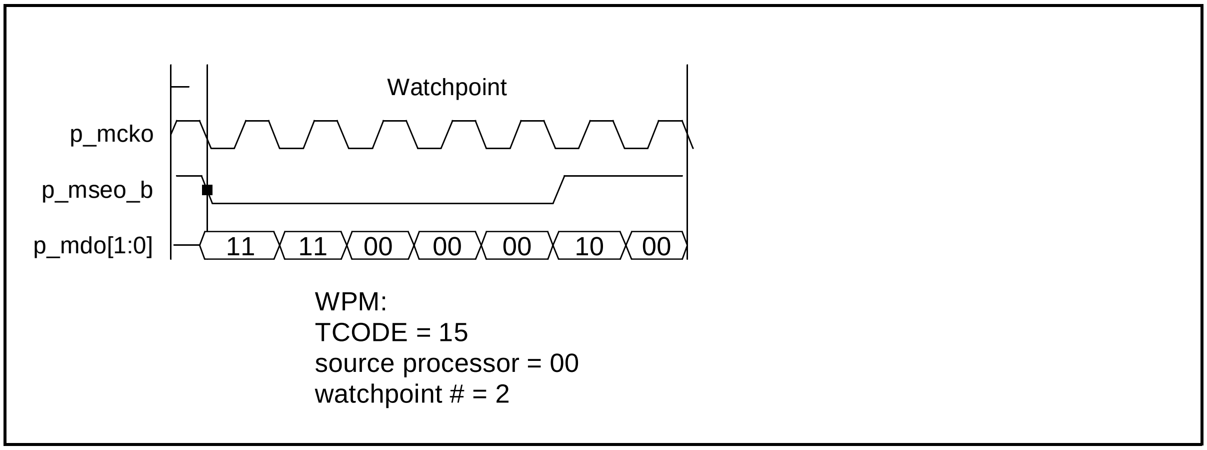


Figure 12-50. Watchpoint Message and Watchpoint Error Message

12.15 Nexus 3+ Read/Write Access to Memory-Mapped Resources

The read/write access feature allows access to memory-mapped resources by means of 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), with multiple configurable priority levels. Memory-mapped registers and other non-cached memory can be accessed by the standard memory map settings.

All accesses are setup and initiated by the read/write access control/status register as well as the read/write access address and read/write access data registers.

Using the read/write access registers, memory mapped e200 AHB resources can be accessed through Nexus 3+. The following subsections describe the steps 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 Nexus 3+ module.

12.15.1 Single Write Access

This section explains the steps required for single write access.

1. Initialize the read/write access address register (RWA) through the access method outlined in Section 12.5, “Nexus 3+ Register Access via JTAG/OnCE.” Configure as follows:
   — Write Address -> 32h’xxxxxxxx (write address)
2. Initialize the read/write access control/status register (RWCS) through the access method outlined in Section 12.5, “Nexus 3+ Register Access via JTAG/OnCE.” Configure the bits as follows:
   — Access Control (AC) -> 1b’1 (to indicate start access)

- Map Select (MAP) -> 3b’000 (primary memory map)
- Access Priority (PR) -> 2b’00 (lowest priority)
- Read/Write (RW) -> 1b’1 (write access)
- Word Size (SZ) -> 3b’0xx (32-bit, 16-bit, 8-bit)
- Access Count (CNT) -> 14h’0000 or 14h’0001(single access)

NOTE

Access Count (CNT) of 14’h0000 or 14’h0001 performs a single access.

3. Initialize the read/write access data register through the access method outlined in Section 12.5, “Nexus 3+ Register Access via JTAG/OnCE.” Configure as follows:
   - Write Data -> 32h’xxxxxxxx (write data)

The Nexus block then arbitrates for the AHB system bus and transfers 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 = 1’b0), Nexus asserts the **nex_rdy_b** pin (see Table 12-39 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.

12.15.2 Block Write Access

This section explains the steps required for block write access.

1. Follow Steps 1, 2, and 3 outlined in Section 12.15.1, “Single Write Access,” to initialize the registers, but use a value greater than one (14’h0001) for the CNT field in the RWCS register.
2. The Nexus block then arbitrates for the AHB system bus and transfers the first data value from the RWD register to the memory mapped address in the RWA register. When the transfer has completed without error (ERR=1’b0), 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 12.15.1, “Single Write Access until the internal CNT value is zero (0). When this occurs, the DV bit within the RWCS 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.

12.15.3 Single Read Access

This section explains the steps required for single read access.

1. Initialize the RWA register through the access method outlined in Section 12.5, “Nexus 3+ Register Access via JTAG/OnCE.” Configure as follows:
   - Read Address -> 32h’xxxxxxxx (read address)
2. Initialize the RWCS register through the access method outlined in Section 12.5, “Nexus 3+ Register Access via JTAG/OnCE.” Configure the bits as follows:
   - Access Control (AC) -> 1b’1 (to indicate start access)
   - Map Select (MAP) -> 3b’000 (primary memory map)
   - Access Priority (PR) -> 2b’00 (lowest priority)
   - Read/Write (RW) -> 1b’0 (read access)
   - Word Size (SZ) -> 3b’0xx (32-bit, 16-bit, 8-bit)
   - Access Count (CNT) -> 14h’0000 or 14h’0001(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=1’b0), Nexus asserts the **nex_rdy_b** pin (see Table 12-39 for detail on **nex_rdy_b**) and sets the DV bit in the RWCS register, indicating that the device is ready for the next access.
4. The data can then be read from the read/write access data register (RWD) through the access method outlined in Section 12.5, “Nexus 3+ Register Access via JTAG/OnCE.”

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.

12.15.4 Block Read Access

This section explains the steps required for block read access.

1. Follow Steps 1 and 2 outlined in Section 12.15.3, “Single Read Access,” to initialize the registers, but use a value greater than one (14’h0001) for the CNT field in the RWCS register.
2. 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 has completed without error (ERR=1’b0), 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. The data can then be read from the read/write access data register (RWD) through the access method outlined in Section 12.5, “Nexus 3+ Register Access via JTAG/OnCE.”
4. Repeat Steps 3 and 4 in Section 12.15.3, “Single Read Access,” until the CNT value is zero (0). When this occurs, the DV bit within the RWCS is set to indicate the end of the block read access.

NOTES

The data values must be shifted out 32-bits at a time LSB first (for example, double word read equals two word reads from the RWD).

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.

12.15.5 Error Handling

The Nexus 3+ module handles various error conditions as follows:

- 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:
    - – The access is terminated without retrying (AC bit is cleared)
    - – The ERR bit in the RWCS register is set
    - – The error message is sent (TCODE = 8) indicating read/write error
- Access Termination
  - — If the AC bit in the RWCS Register 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.
  - — If a block access is in progress (all cycles not completed), and the RWCS register is written, then the original block access is terminated at the boundary of the nearest completed access.
    - – If the RWCS is written with the AC bit set, the next read/write access begins and the RWD can be written to/ read from.
    - – 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.
- 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 12-50.

(5 bits)	(4 bits)	(6 bits)
Error Code (00011)	Src. Proc.	TCODE (001000)

Fixed length = 15 bits

Figure 12-51. Error Message Format

12.16 Nexus 3+ Pin Interface

The Nexus 3+ pin interface provides the function of transmitting messages from the message queues to the external tools. It is also responsible for handshaking with the message queues.

12.16.1 Pins Implemented

The Nexus 3+ module implements an auxiliary port consisting of one **nex_evti_b** and one **nex_mseo_b** or two **nex_mseo_b[1:0]**. It also implements a configurable number of **nex_mdo[n:0]** pins, one **nex_rdy_b** pin, one **nex_evto_b** pin, three **nex_wevto[2:0]** pins, and one clock output pin (**nex_mcko**), as well as additional configuration pins described in Table 12-39. 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 12.5, “Nexus 3+ Register Access via JTAG/OnCE,” for details). The JTAG pins are incorporated as I/O to the e200 processor and are further described in Section 11.4.3, “JTAG/OnCE Pins.”

Table 12-38. 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 12-39.

Table 12-39. 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]** & **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**).

Table 12-39. Nexus 3+ Auxiliary Pins (Continued)

Auxiliary Pins	Input/ Output	Description of Auxiliary Pins
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
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 Nexus3 is enabled, this signal is asserted upon successful (without error) completion of an AHB system bus transfer (Nexus read or write) & is held asserted until the JTAG/OnCE state machine reaches the “Capture_DR” state. Upon exit from system reset or if Nexus3 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) & EOC = 2’b00 2) debug mode was entered (jd_debug_b asserted from Nexus1) & EOC = 2’b01
nex_evti_b	I	Event In (**nex_evti_b**) is an input which, when asserted, will initiate one of two events based on the EIC bits in the DC1 Register (if the Nexus3 module is enabled at reset): 1) Program Trace & Data Trace synchronization messages (provided Program Trace & Data Trace are enabled & EIC = 2’b00). 2) Debug request to e200 Nexus1 module (provided EIC = 2’b01 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 that shares a single auxiliary output port. The arbitration is controlled by an SoC-level Nexus port control module (NPC). Refer to Section 12.18, “Auxiliary Port Arbitration,” for detail on Nexus port arbitration.

Table 12-40 shows the Nexus port arbitration signals.

Table 12-40. 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 to an SoC level Nexus arbiter a request for access to the shared Nexus auxiliary port in a multi-Nexus implementation. The priority encodings are determined by how many messages are currently in the message queues (see Table 12-42).
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.

Table 12-40. Nexus Port Arbitration Signals

Nexus Port Arbitration Pins	Input/ Output	Description of Arbitration Pins
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).

12.16.2 Pin Protocol

The protocol for the e200 processor transmitting messages via the auxiliary pins is accomplished with the MSEO pin function outlined in Table 12-41. 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 12-41. 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 1’s)	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 12-52 illustrates the state diagram for single pin MSEO transfers.

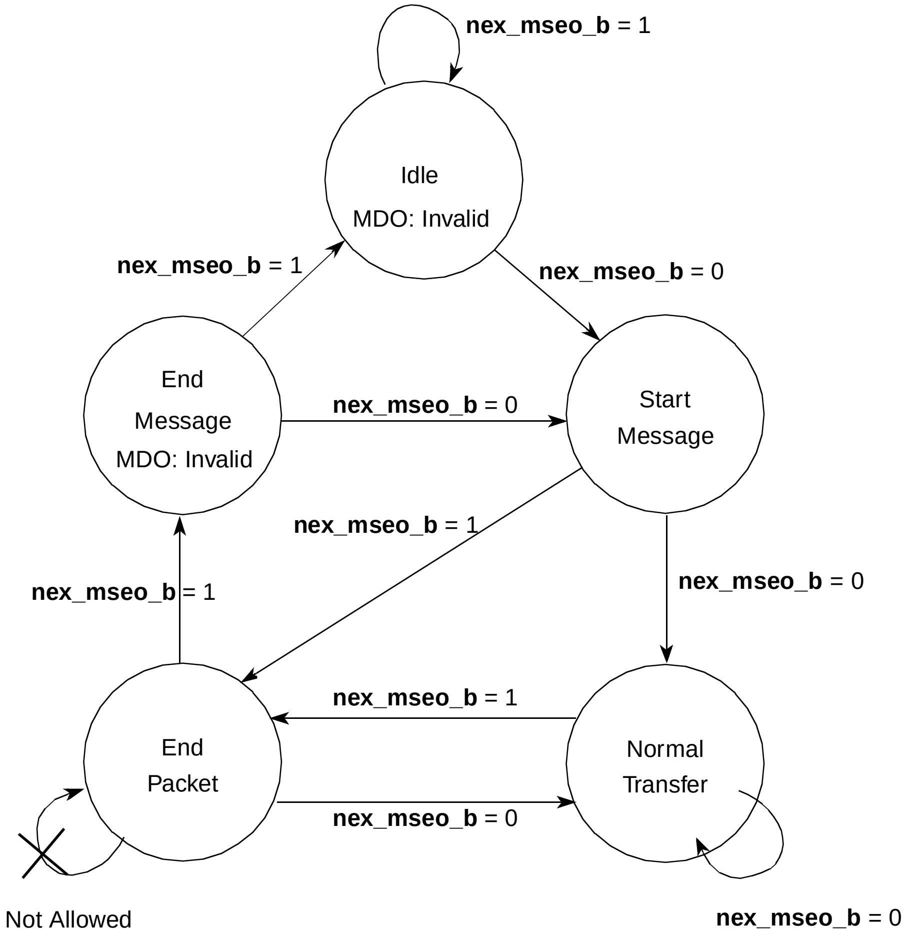


Figure 12-52. Single Pin MSEO Transfers

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 ones on the **nex_mseo_b** pin before the actual end of message.

Figure 12-53 illustrates the state diagram for dual pin MSEO transfers.

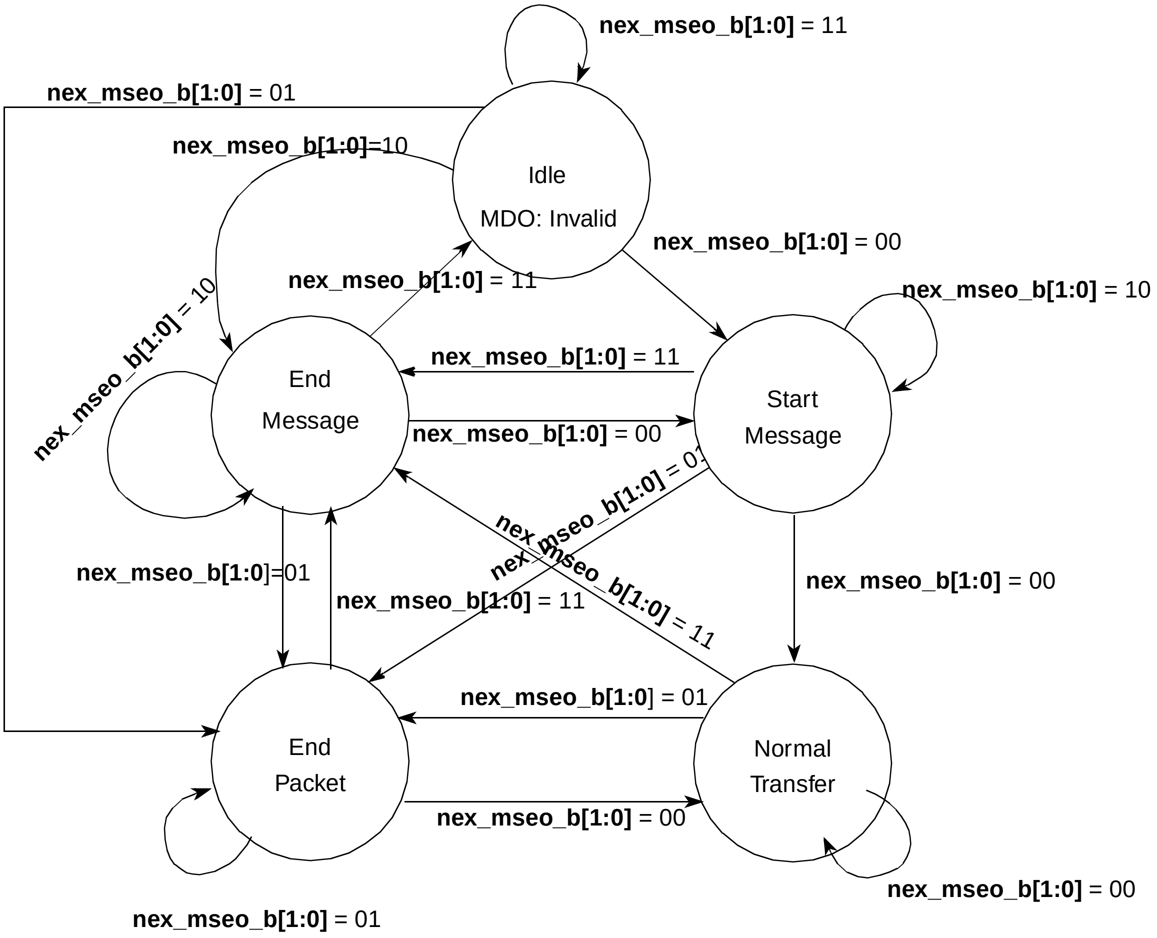


Figure 12-53. Dual Pin MSEO Transfers

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.

12.17 Rules for Output Messages

The e200-based Class 3 compliant embedded processors must provide messages through the auxiliary port in a consistent manner as described below:

- 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.

12.18 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 12-42 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 12.16.2, "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 12-42. 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

12.19 Examples

The following are examples of program trace and data trace messages.

Table 12-43 illustrates an example indirect branch message with 2 MDO/1 MSEO configuration. Table 12-44 illustrates the same example with an 8 MDO/2 MSEO configuration.

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 12-43. 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 12-44. 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 12-45 and Table 12-46 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 12-45. 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 12-46. 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 12-47 illustrates an example Data Write Message with 8 MDO/1 MSEO configuration. Table 12-48 illustrates the same DWM with 8 MDO/2 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 12-47. 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)
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 12-48. 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

12.20 Electrical Characteristics

For all electrical characteristics related to e200 and Nexus 3+ operation, please refer to the appropriate “e200 Integration Guide.”

12.21 IEEE 1149.1 (JTAG) RD/WR Sequences

This section contains example JTAG/OnCE sequences used to access resources.

12.21.1 JTAG Sequence for Accessing Internal Nexus Registers

Table 12-49 shows the JTAG sequence for accessing internal Nexus 3+ registers.

Table 12-49. Accessing Internal Nexus 3+ Registers through 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)
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)

12.21.2 JTAG Sequence for Read Access of Memory-Mapped Resources

Table 12-50 shows the JTAG sequence for read access of memory-mapped resources.

Table 12-50. 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

12.21.3 JTAG Sequence for Write Access of Memory-Mapped Resources

Table 12-50 shows the JTAG sequence for write access of memory-mapped resources.

Table 12-51. 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

Chapter 13 External Core Complex Interfaces

This chapter describes the external interfaces to the e200z446n3 core complex, including signal descriptions as well as the data transfer protocols.

13.1 Overview

The external interfaces encompass control and data signals supporting instruction and data transfers, 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 is comprised of a pair of 64-bit wide standard AHB 2.v6 system buses, one for instructions and the other for data. Both interfaces operate in a pipelined fashion and support misaligned transfers and true big- and little-endian operating modes. The data memory interface supports read and write transfers of 8, 16, 24, 32, and 64 bits, and the instruction memory interface supports read transfers of 16, 32, and 64 bits.

The memory interface supported by the bus interface units (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 e200z446n3 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 13.3.5, “Byte Lane Specification,” and Table 13-10.

Single-beat and misaligned transfers are supported for cache-inhibited read and write cycles. Burst transfers (double word aligned) of four double words are supported for cache line-fill operations.

Misaligned accesses are supported with one or more transfers to an interface. If an access is misaligned but contained within an aligned 64-bit double word, the core performs a single transfer. 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. The size and byte enable signals are driven to correspond to the number of remaining bytes to be transferred.

13.2 Signal Index

This section contains an index of the e200 signals.

The following prefixes are used for e200 signal mnemonics:

m	master clock and reset signals
p	processor or core-related signals
j	JTAG mode signals
jd	JTAG and Debug mode signals
ipt	Scan and test mode signals
nex	Nexus signals

NOTE

The “_b” suffix denotes an active low signal. Signals without the active-low suffix are active high.

Figure 13-1 groups core bus and control signals by function.

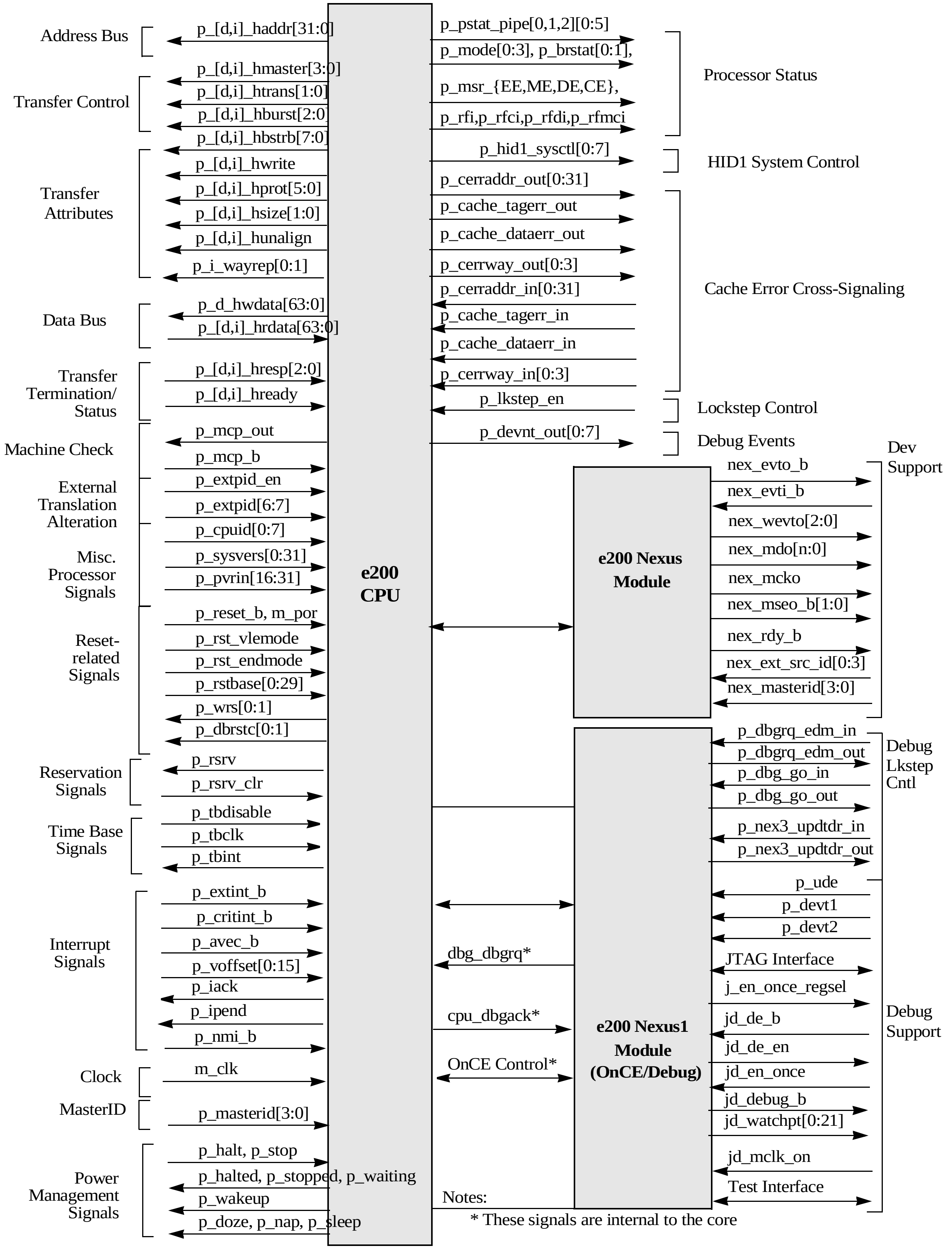


Figure 13-1. e200 Signal Groups

Table 13-1 shows e200 signal function and type, signal definition, and reset value. Signals are presented in functional groups.

Table 13-1. 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
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_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_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_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

Table 13-1. Interface Signal Definitions (Continued)

Signal Name	Type	Reset Value	Definition
Master ID Configuration Signals			
p_masterid[3:0]	I	—	CPU Master ID configuration
nex_masterid[3:0]	I	—	Nexus Master ID configuration
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_cerraddr_in[0:31]	I	—	Cache error address input
p_cerrway_in[0:3]	I	—	Cache error ways input
p_cache_tagerr_in	I	—	Cache tag error input
p_cache_dataerr_in	I	—	Cache data error input
p_cerraddr_out[0:31]	O	—	Cache error address output
p_cerrway_out[0:3]	O	—	Cache error ways output
p_cache_tagerr_out	O	0	Cache error update output
p_cache_dataerr_out	O	0	Cache data error 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

Table 13-1. Interface Signal Definitions (Continued)

Signal Name	Type	Reset Value	Definition
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
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 Debug Event Signals			
p_ude	I	—	Unconditional Debug Event
p_devt1	I	—	Debug Event 1 input

Table 13-1. Interface Signal Definitions (Continued)

Signal Name	Type	Reset Value	Definition
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
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[0:21]	O	0	Indicate 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_out	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[2:0]	O	—	Nexus 3 Watchpoint Event-Out Outputs
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
nex_ext_src_id[0:3]	I	—	Nexus 3 SRC ID Input
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

Table 13-1. Interface Signal Definitions (Continued)

Signal Name	Type	Reset Value	Definition
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
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[1]	—	—	Test Mode determination
Scan Test Interface[1]	—	—	Scan Configuration and Testing
Memory BIST Interface[1]	—	—	Memory BIST Configuration and Testing

[1] Please refer to the e200 Test Guide for information on the Test signals

13.3 Signal Descriptions

The following subsections provide descriptions of the signals.

13.3.1 e200 Processor Clock (m_clk)

The **m_clk** input is the synchronous clock source for the e200 processor core.

Since the e200 is designed for static operation, **m_clk** can be gated off to lower power dissipation (for example, during low-power stopped states).

13.3.2 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 and can be conditionally asserted by watchdog time-outs.

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.

13.3.2.1 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. 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. The resulting signal clears certain CPU registers. This is an asynchronous clear with a short assertion time requirement.

13.3.2.2 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.

13.3.2.3 Watchdog Reset Status (p_wrs[0:1])

The **p_wrs[0:1]** outputs are active-high reset output status signals from the e200 core which reflect the value of the TSR[WRS] status field. **p_wrs[0:1]** are conditionally asserted by the Watchdog Timer (Section 2.4.8, “Timer Control Register (TCR),” and Section 2.4.9, “Timer Status Register (TSR)”).

13.3.2.4 Debug Reset Control (p_dbrstc[0:1])

The **p_dbrstc[0:1]** outputs are active-high reset output control signals from the e200 core. They reflect the value of the DBCR0[RST] status field and are conditionally asserted by the debug control logic (Section 11.3.3.1, “Debug Control Register 0 (DBCR0)”).

13.3.2.5 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 occurs to the location [**p_rstbase[0:29]** || 2’b00].

13.3.2.6 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 causes the resultant entry ‘E’ bit to set to ‘0’, indicating a big-endian page. A high logic level on this signal causes the resultant entry ‘E’ bit to set to ‘1’, indicating a little-endian page.

13.3.2.7 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 causes the resultant entry ‘VLE’ bit to set to ‘0’, indicating a Power ISA page. A high logic level on this signal causes the resultant entry ‘VLE’ bit to set to ‘1’, indicating a VLE page.

13.3.2.8 JTAG/OnCE Reset (j_trst_b)

The **j_trst_b** signal (referred to in the *IEEE 1149.1 JTAG Specification* as the 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.

13.3.3 Address and Data Buses

Dual instruction and data interfaces are provided by the e200z446n3. They are described together, with appropriate differences denoted.

13.3.3.1 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.

13.3.3.2 Read Data Bus (p_d_hrdata[63:0], p_i_hrdata[63:0])

These inputs provide data to the e200z446n3 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 13-2 shows the relationship of byte addresses to read data bus signals.

Table 13-2. 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

13.3.3.3 Write Data Bus (p_d_hwdata[63:0])

These outputs transfer data from the e200z446n3 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 13-3 shows the relationship of byte addresses to write data bus signals.

Table 13-3. 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

13.3.4 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.

13.3.4.1 Transfer Type (p_d_htrans[1:0], p_i_htrans[1:0])

The processor drives these signals to indicate the current transfer type. Table 13-4 shows **p_[d,i]_htrans[1:0]** encoding.

Table 13-4. 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 the e200z4)
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 neither IDLE or BUSY, a transfer is being requested. The e200z446n3 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.

13.3.4.2 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.

13.3.4.3 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 13-5 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 13-11 and Table 13-12 for **p_[d,i]_hsize[1:0]** encodings used for aligned and misaligned transfers.

Table 13-5. 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)

13.3.4.4 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 13-6 shows the definitions of the **p_[d,i]_hburst[2:0]** encodings.

Table 13-6. 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 e200z446n3 only utilizes SINGLE and WRAP4 burst types. In addition, all WRAP4 bursts are of double-word size aligned to double-word boundaries.

13.3.4.5 Protection Control (p_d_hprot[5:0], p_i_hprot[5:0])

The e200z446n3 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 (such as 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 13-7 shows the definitions of the **p_d_hprot[5:0]** signals.

Table 13-7. 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, Writethrough
—	1	1	1	—	1	Cacheable, Writeback
0	—	—	—	—	1	Not Exclusive
1	—	—	—	—	1	Exclusive Access

Table 13-8 shows the definitions of the **p_i_hprot[5:0]** signals.

Table 13-8. 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

Table 13-8. 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	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. For example, p_d_hprot0 is always high).

The e200z446n3 maps the Power ISA embedded category storage attributes to the AHB data port **hprot** signals in the manner described in Table 13-9:

Table 13-9. 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, Writeback
0	0	1	1	1	0	Cacheable, Writethrough
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 Writethrough, and non-Guarded
—	—	—	1	1	1	Buffered Store and page marked copyback, and non-Guarded

13.3.4.6 Cache Way Replacement (p_i_wayrep[0:1])

The **p_i_wayrep[0:1]** control signals are driven valid during cache line-fills 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.

13.3.5 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.

13.3.5.1 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 size and low order address bits.

13.3.5.2 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 13-3 shows the relationship of byte addresses to the byte strobe signals.

Table 13-10. 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 13-11 lists all of the data transfer permutations. Note that misaligned data requests which 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 e200z446n3 performs the proper byte routing internally based on endianness.

Table 13-11. 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

Table 13-11. 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 @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
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 @100 (2 bus transfers)	1 0 0 +0 0 0	1 1* 1 0	— X	— X	— X	— X	X —	X —	X —	X —	1 0

Table Notes:
"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 13-12 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 which cross a double-word boundary are broken into a pair of accesses by the CPU. Double-word transfers are always word or double-word aligned.

Table 13-12. Big and Little Endian Memory 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	—	—	—	—	—	—	—	—	—	—	—	—	—
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	—	—	—

Table 13-12. Big and Little Endian Memory 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 @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	—	—	—	—	—	—	—	—	—	—	—
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	—	—	—	—	—	—	—	—

Table 13-12. Big and Little Endian Memory 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 @0101	0 1 0 1	1 0	—	—	—	—	—	E	F	G	—	—	—	—	—	—	—	—
	1 0 0 0	0 0	—	—	—	—	—	—	—	—	H	—	—	—	—	—	—	—
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	—	—	—	—	—	—	—
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	—	—	—	—	—

Table 13-12. Big and Little Endian Memory 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 @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 @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 @1000	1 0 0 0	1 1	—	—	—	—	—	—	—	—	A	B	C	D	E	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	—	—	—	—	—	—	—	—	—	—	—	—
L.E. Double Word @0000	0 0 0 0	1 1	H	G	F	E	D	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 @1000	0 0 0 0	1 1	—	—	—	—	—	—	—	—	H	G	F	E	D	C	B	A

Table 13-12. Big and Little Endian Memory 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 @1100	1 1 0 0	1 1	—	—	—	—	—	—	—	—	—	—	—	—	H	G	F	E
	+ 0 0 0 0 (next dword)	1 0	D	C	B	A	—	—	—	—	—	—	—	—	—	—	—	—

Notes:
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.

13.3.6 Transfer Control Signals

The following paragraphs describe the transfer control signals.

13.3.6.1 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 status of the transfer.

13.3.6.2 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 status of a terminating transfer on the respective interfaces. Table 13-13 and Table 13-14 show the definitions of the **p_i_hresp[1:0]** and **p_d_hresp[2:0]** encodings.

Table 13-13. 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)

Table 13-14. 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

Table 13-14. p_d_hresp[2:0] Transfer Response Encoding (Continued)

p_d_hresp[2:0]	Response Type
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

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. See Section 12.2.3.16, "Store Exclusive Failure (p_d_xfail_b)."

13.3.7 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 submultiples of the **m_clk** frequency. These signals do not affect non-AHB interface signals.

13.3.7.1 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.

13.3.7.2 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 because **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.

13.3.8 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.

13.3.8.1 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.

13.3.8.2 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.

13.3.9 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; thus the interrupt controller module must keep the interrupt request as well as the appropriate **p_voffset** or **p_avec_b** inputs 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 13.3.9.8, “Machine Check (p_mcp_b),” and Section 13.3.9.3, “Non-Maskable Input Interrupt Request (p_nmi_b).”

13.3.9.1 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; thus 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.

13.3.9.2 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 MSR[CE]. This signal is not internally synchronized by the e200 core; thus 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.

13.3.9.3 Non-Maskable Input Interrupt Request (p_nmi_b)

This active-low, transition sensitive signal provides a non-maskable interrupt request to the e200 core. This signal is not internally synchronized by the e200 core; thus 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. Instead, the **p_ipend** and **p_wakeup** signals are asserted to indicate to system logic that an interrupt is pending, so the clocks should be started and the **p_halt** and **p_stop** inputs negated so that the interrupt may be processed.

13.3.9.4 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 MCSR[NMI] 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 affects the **p_ipend** signal slightly differently; the **p_ipend** output asserts any time the **p_nmi_b** input is asserted or whenever the MCSR[NMI] syndrome bit is set.

13.3.9.5 Autovector (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.

13.3.9.6 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.

13.3.9.7 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 is 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 13-46 and Figure 13-47 for timing diagrams of operation.

13.3.9.8 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 the HID0[EMCP] bit. This signal is not internally synchronized by the e200 core; therefore, it 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 are not 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 •, "Processor Waiting (p_waiting)," Section •, "Processor Halted (p_halted)," and Section •, "Processor Stopped (p_stopped)."

13.3.10 Lockstep Enable Signal (p_lkstep_en)

The **p_lkstep_en** signal enables lockstep cross-signaling operation for the cache 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 13.4.3, "Cache Error Cross-Signaling Operation," and Section 13.3.21, "Debug Lockstep Cross-signaling Signals." Transitions on this signal must be properly coordinated by the SoC to ensure that enabling and disabling of lockstep operation is performed at appropriate operational boundaries, otherwise undefined behavior may result.

13.3.11 Cache Error Cross-signaling Signals

The following paragraphs describe the cache error cross-signaling interface signals. Examples of operation are provided in Section 13.4.3, “Cache Error Cross-Signaling Operation.”

13.3.11.1 Cache Tag Error Out (p_cache_tagerr_out)

The active-high **p_cache_tagerr_out** output signal is used to indicate a valid cache tag parity error has occurred during this cycle. It is only signaled if a cache operation or exception is signaled by the detected error condition. When L1CSR1[ICEA] indicates machine check generation on error, assertion of this signal indicates a machine check is signaled for the access, or for **icbi** operations, indicates that a remote invalidation of one or more cache lines should occur. When L1CSR1[ICEA] indicates auto-invalidation on error, assertion of this signal indicates that the cache inserts 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.

13.3.11.2 Cache Data Error Out (p_cache_dataerr_out)

The active-high **p_cache_dataerr_out** output signal issued to indicate a valid cache data array parity error has occurred during this cycle. It is only signaled if a cache operation or exception is signaled by the detected error condition. When L1CSR1[ICEA] indicates machine check generation on error, assertion of this signal indicates a machine check will be signaled for the access. When L1CSR1[ICEA] indicates auto-invalidation on error, assertion of this signal indicates that the cache forces a miss cycle to refill the cache line to correct the error data. The way with the data error to be reloaded is indicated by the assertion of one of the **p_cerrway_out[0:3]** signals in the following cycle. This signal is reset to 0.

13.3.11.3 Cache Error Address Out (p_cerraddr_out[0:31])

The active-high **p_cerraddr_out[0:31]** output signals are used to provide the physical address corresponding to a cache error signaled by the **p_cache_tagerr_out** and **p_cache_dataerr_out** output signals. These signals should be qualified with the assertion of **p_cache_tagerr_out** or **p_cache_dataerr_out** in the previous cycle. The **p_cerraddr_out[0:31]** values are provided to allow for proper updating of the MCAR on machine check exceptions by another cache. These signals are undefined following reset.

13.3.11.4 Cache Error Way(s) Out (p_cerrway_out[0:3])

The active-high **p_cerrway_out[0:3]** output signals are used to indicate which way(s) of the cache encountered a cache tag or data array error. These signals should be qualified with the assertion of **p_cache_tagerr_out** or assertion of **p_cache_dataerr_out** in the previous cycle. Uncorrectable tag errors are indicated by the assertion of one or more of the **p_cerrway_out[0:3]** signals in the cycle following assertion of **p_cache_tagerr_out**. If only correctable tag errors are found, these signals remain negated. When reloading of a cache way to correct a data error occurs, the way with the data error to be reloaded is indicated by the assertion of one of the **p_cerrway_out[0:3]** signals in the cycle following assertion of **p_cache_dataerr_out**. These signals are reset to 0.

13.3.11.5 Cache Tag Error In (p_cache_tagerr_in)

The active-high **p_cache_tagerr_in** input signal is used to indicate a cache tag parity error is being cross-signaled from another cache during this cycle. When cross-signaling operation is enabled via assertion of the **p_lkstep_en** input signal, assertion of this signal indicates the values of **p_cerraddr_in[0:31]** and **p_cerrway_in[0:3]** in the next cycle should be used to cause a cache parity error to be emulated for the indicated way(s) of the cache, using the index provided by **p_cerraddr_in[0:31]**. Depending on the settings of L1CSR1[ICEA], either a machine check or a possible invalidation should occur. MCAR should be updated with the value on **p_cerraddr_in[0:31]** if a machine check is signaled and MAV was previously clear. Normally, the value of **p_cerraddr_in[0:31]** is identical to the internal value of **p_cerraddr_out[0:31]**, although an undetected internal error causing a loss of synchronization between two (or more) processors operating in lockstep may cause the values to be different. This loss of synchronization will be detected at some future time.

13.3.11.6 Cache Data Error In (p_cache_dataerr_in)

The active-high **p_cache_dataerr_in** input signal is used to indicate a cache data array parity error is being cross-signaled from another cache during this cycle. When cross-signaling operation is enabled via assertion of the **p_lkstep_en** input signal, assertion of this signal indicates the values of **p_cerraddr_in[0:31]** and **p_cerrway_in[0:3]** in the next cycle should be used to cause a cache parity error to be emulated for the indicated way(s) of the cache, using the index provided by **p_cerraddr_in[0:31]**. Depending on the settings of L1CSR1, either a machine check or a refill should occur. MCAR should be updated with the value on **p_cerraddr_in[0:31]** if a machine check is signaled and MAV was previously clear. Normally, the value of **p_cerraddr_in[0:31]** is identical to the internal value of **p_cerraddr_out[0:31]**, although an undetected internal error causing a loss of synchronization between two (or more) processors operating in lockstep may cause the values to be different. This loss of synchronization will be detected at some future time.

13.3.11.7 Cache Error Way(s) In (p_cerrway_in[0:3])

When cross-signaling operation is enabled by the assertion of the **p_lkstep_en** input signal, the active-high **p_cerrway_in[0:3]** input signals are used to indicate whether the corresponding ways of the cache should emulate a cache error. These signals should be qualified with the assertion of **p_cache_tagerr_in** or **p_cache_dataerr_in** in the previous cycle.

13.3.12 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 Real-time Systems.”

13.3.12.1 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 the dynamic remapping capability to be used.

13.3.12.2 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**.

13.3.13 Timer Facility Signals

The following sub-sections describe the processor signals associated with the time base, watchdog, fixed-interval and decrementer facilities.

13.3.13.1 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. Therefore, it 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.

13.3.13.2 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**.

13.3.13.3 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.

13.3.14 Processor Reservation Signals

The following subsections describe processor reservation signals associated with the **lbarx**, **lharx**, **lwarx**, **stbcx.**, **sthcx.**, and **stwcx.** instructions.

13.3.14.1 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 3.6, “Memory Synchronization and Reservation Instructions”). This signal is provided as a status indicator for specialized system applications only.

13.3.14.2 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 3.6, “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. See Section 12.2.3.16, “Store Exclusive Failure (p_d_xfail_b),” for more details about **p_d_xfail_b**.

13.3.15 Miscellaneous Processor Signals

The following list describes miscellaneous active-high processor signals.

- CPU ID (p_cpuid[0:7])
  - Input signals used to provide an identity for a particular processor.
  - Reflected in the processor ID register (Section 2.4.2, “Processor ID Register (PIR)) following reset.
  - Intended to remain in a static condition and are not internally synchronized.
- PID0 outputs (p_pid0[0:7])
  - Output signals used to provide the current process ID in the process ID rRegister 0 (PID0).
  - Correspond to the low order eight bits of PID0.
- PID0 Update (p_pid0_updt)
  - Output signal used to indicate that the process ID register 0 (PID0) is being updated by a **mtspr** instruction.
  - Asserts during the clock cycle the **p_pid0[0:7]** outputs are changing.
- System Version (p_sysvers[0:31])
  - Input signals used to provide a version number for the particular system incorporating a e200 CPU.
  - Reflected in the system version register (Section 2.4.4, “System Version Register (SVR)”).
  - Intended to remain in a static condition and are <u>not</u> internally synchronized.
- Processor Version (p_pvrin[16:31])
  - Used to provide a portion of the version number for a particular e200 CPU.
  - Reflected in the processor version register (Section 2.4.3, “Processor Version Register (PVR)”).
  - Intended to remain in a static condition and are not internally synchronized.

- HID1 System Control (p_hid1_sysctl[0:7])
  - — Output signals used to provide a set of control output signals external to the CPU by means of values written to the HID1 special purpose register.
  - — 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).

13.3.16 Processor State Signals

The following sub-sections describe processor internal state signals.

13.3.16.1 Processor Mode (p_mode[0:3])

These signals indicate the global processor execution status. The timing is synchronous with **m_clk**. Table 13-16 shows **p_mode[0:3]** encoding.

Table 13-15. 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

13.3.16.2 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 13-16 shows **p_pstat_pipe{0,1}[0:5]** encodings.

Table 13-16. 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

13.3.16.3 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 t**m_clk**, so the indicated status may not apply to a current bus transfer. Table 13-17 shows **p_brstat[0:1]** encoding.

Table 13-17. 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

13.3.16.4 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 an **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.

13.3.16.5 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 un-nested interrupt exception handlers, and may 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.

13.3.16.6 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)."

13.3.17 Power Management Control Signals

The following active-high signals are provided for power management or other control functions by external control logic.

- Processor Waiting (p_waiting)
  - — Output signal used to indicate that the processor has entered the waiting state (Section 8.2, "Waiting State").
- Processor Halt Request (p_halt)
  - — Input signal used to request the processor to enter the halted state (Section 8.3, "Halted State").
- Processor Halted (p_halted)
  - — Output signal used to indicate that the processor has entered the halted state (Section 8.3, "Halted State").
- Processor Stop Request (p_stop)
  - — Input signal used to request the processor to enter the stopped state (Section 8.4, "Stopped State").
- Processor Stopped (p_stopped)
  - — Output signal used to indicate that the processor has entered the stopped state (Section 8.4, "Stopped State").

13.3.17.1 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 the MSR[WE] bit 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 13.3.13, "Timer Facility Signals."

13.3.17.2 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:

- 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 by 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.

13.3.18 Debug Event Input Signals

The following interface signals are provided to signal debug events to the e200 core.

13.3.18.1 Unconditional Debug Event (p_ude)

The active-high **p_ude** input signal is used to request an unconditional debug event. This event is described in detail in Section 11.2.13, "Unconditional Debug Event." This signal is not internally synchronized by the e200 core; thus 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 be held asserted until acknowledged by software or when external debug mode is enabled, by assertion of the **jd_debug_b** output for recognition to be guaranteed.

In addition, only a transition from the negated state to the asserted state of the **p_ude** signal causes an event to occur. The level on this signal is used, however, to cause assertion of the **p_wakeup** output.

13.3.18.2 External Debug Event 1 (p_devt1)

The active-high **p_devt1** input signal is used to request an external debug event. This event is described in detail in Section 11.2.12, "External Debug Event." This signal is not internally synchronized by the e200 core; thus it 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.

13.3.18.3 External Debug Event 2 (p_devt2)

The active-high **p_devt2** input signal is used to request an external debug event. This event is described in detail in Section 11.2.12, "External Debug Event." This signal is not internally synchronized by the e200 core; thus it 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 causes an event to occur. It is intended to signal e200 related events that are generated while the CPU is active.

13.3.19 Debug Event Output Signals (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.

13.3.20 Debug/Emulation (Nexus 1/ OnCE) Support Signals

The following interface signals are provided to assist in implementing an on-chip emulation capability with a controller external to the e200 core.

Table 13-18. 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

13.3.20.1 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; 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 this 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** and **j_key_in** signals are provided to assist external logic performing security checks.

Refer to Section •, “Enable Once Register Select (j_en_once_regsel),” for a description of the **j_en_once_regsel** output signal and to Section •, “Key Data In (j_key_in),” for a description of the **j_key_in** input 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.

13.3.20.2 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 is provided to allow system resources to be aware that access is occurring for debug purposes, thus allowing certain resource side effects to be frozen or otherwise controlled. Examples include FIFO state change control and control of side-effects of register or memory accesses. Refer to Section 11.4.5.3, "e200 OnCE Debug Output (jd_debug_b)," for additional information on this signal.

13.3.20.3 Debug Request (jd_de_b)

This signal is the debug mode request input. This signal is not internally synchronized by the e200 core; thus it 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 11.4.5.2, "OnCE Debug Request/Event (jd_de_b, jd_de_en)," for additional information on this signal.

13.3.20.4 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 11.4.5.2, "OnCE Debug Request/Event (jd_de_b, jd_de_en)," for additional information on this signal.

13.3.20.5 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.

13.3.20.6 Watchpoint Events (jd_watchpt[0:21])

The **jd_watchpoint[0:21]** 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 11.5, "Watchpoint Support," for the signal assignments of each watchpoint source.

13.3.21 Debug Lockstep Cross-signaling Signals

The following paragraphs describe the debug lockstep cross-signaling interface signals, which are used to enable lockstep debug operations. Examples of operation are provided in Section 13.4.4, "Debug Lockstep Cross-signaling Operation."

13.3.21.1 Debug Request EDM In (p_dbgrq_edm_in)

The active-high **p_dbgrq_edm_in** input signal is used to indicate 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 by means of a **tclk** domain mechanism, such as the setting of OCR[DR] 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.

13.3.21.2 Debug Request EDM Out (p_dbgrq_edm_out)

The active-high **p_dbgrq_edm_out** output signal is used to indicate that a request to enter a debug halted state has occurred. The request may have occurred by means of 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.

13.3.21.3 Debug Go Request In (p_dbg_go_in)

The active-high **p_dbg_go_in** input signal is used to indicate 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 by means of 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.

13.3.21.4 Debug Go Request Out (p_dbg_go_out)

The active-high **p_dbg_go_out** output signal is used to indicate that a request to exit a debug halted state has occurred. A request occurs by 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.

13.3.21.5 Debug Nexus 3 Update_DR state In (p_nex3_updtdr_in)

The active-high **p_nex3_updtdr_in** input signal is used to indicate that a synchronized Update_DR state has been recognized in another CPU and that Nexus 3 register updates may occur when the receiving CPU has an internally generated Update_DR request present. When the Update_DR state is entered in the **tclk** clock domain and is synchronized to the **m_clk** domain, this signal is used to qualify updates to Nexus 3 control registers to assure that updates occur in synchrony in all lockstep processors. 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_nex3_updtdr_in** input signal is qualified with the **p_lkstep_en** input, and is only used to condition Nexus 3 register updates via the Update_DR state when **p_lkstep_en** is asserted. If **p_lkstep_en** is negated, this input signal is ignored.

13.3.21.6 Debug Nexus 3 Update_DR state Out (p_nex3_updtdr_out)

The active-high **p_nex3_updtdr_out** output signal is used to indicate that a synchronized Update_DR state has been reached internally. When the Update_DR state is entered in the **tclk** clock domain and is synchronized to the **m_clk** domain, this output is asserted, and remains asserted until updates due to an Update DR state occur. This signal is synchronized to the **m_clk** clock domain.

13.3.22 Development Support (Nexus 3) Signals

Table 13-19 shows the interface signals provided to assist in implementing a real-time development tool capability with a controller external to the e200 core.

Table 13-19. e200 Development Support (Nexus 3) 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_evti_b	I	Nexus Event-In Input
nex_wevto[2:0]	O	Nexus Watchpoint Event-Out Outputs
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

13.3.23 JTAG Support Signals—Primary Interface

Table 13-20 lists the primary JTAG interface signals. These signals are usually connected directly to device pins (except for **j_tdo**, which needs three-state and edge support logic). However, this may not be the case when JTAG TAP controllers are concatenated together.

Table 13-20. 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.

The following list describes the JTAG primary interface signals in greater detail.

- JTAG/OnCE Serial Input (j_tdi)
  - — Provides data and commands to the OnCE controller. Data is latched on the rising edge of the **j_tclk** serial clock and shifted into the OnCE serial port least significant bit (LSB) first.
- JTAG/OnCE Serial Clock (j_tclk)
  - — Supplies the OnCE control block with the serial clock, which 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 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.
- 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**.
  - — Always driven.
  - — An external system-level TDO pin can be released to high impedance 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.
    - – For IEEE Std. 1149 conformity, the system level pin should change state on the falling edge of TCLK.
- JTAG/OnCE Test Mode Select (j_tms)
  - — Input 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.
- JTAG/OnCE Test Reset (j_trst_b)
  - — Input used to externally reset the OnCE controller by placing it in the Test-Logic-Reset state.

13.3.24 JTAG Support Signals—Support for External Registers

Table 13-21 details additional signals that may be used to support external JTAG data registers using the e200 TAP controller.

Table 13-21. 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_key_in	I	Serial data from external key logic
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

Table 13-21. JTAG Signals Used to Support External Registers (Continued)

Signal Name	Type	Description
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)

The following list describes the JTAG signals used to support external registers in greater detail.

- Test-Logic-Reset (j_tst_log_rst)
  - Indicates the TAP controller is in the Test-Logic-Reset state.
- Run-Test/Idle (j_rti)
  - Indicates the TAP controller is in the Run-Test/Idle state.
- Capture IR (j_capture_ir)
  - Indicates the TAP controller is in the Capture_IR state.
- Shift IR (j_shift_ir)
  - Indicates the TAP controller is in the Shift_IR state.
- Update IR (j_update_ir)
  - Indicates the TAP controller is in the Update_IR state.
- Capture DR (j_capture_dr)
  - Indicates the TAP controller is in the Capture_DR state.
- Shift DR (j_shift_dr)
  - Indicates the TAP controller is in the Shift_DR state.
- Key Data In (j_key_in)
  - 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.
  - Provided to assist with implementing security logic outside of the e200z446n3, 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**
    - After a two clock delay the data is internally routed to **j_tdo**, which allows provision of a key value through the **j_tdo** output following a transition from Capture_IR to Shift_IR. The key value is provided by the **j_key_in** input.
- Update DR with Write (j_update_gp_reg)
  - Indicates that the TAP controller is in the Update_DR state and 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.
- Register Select (j_gp_regsel)

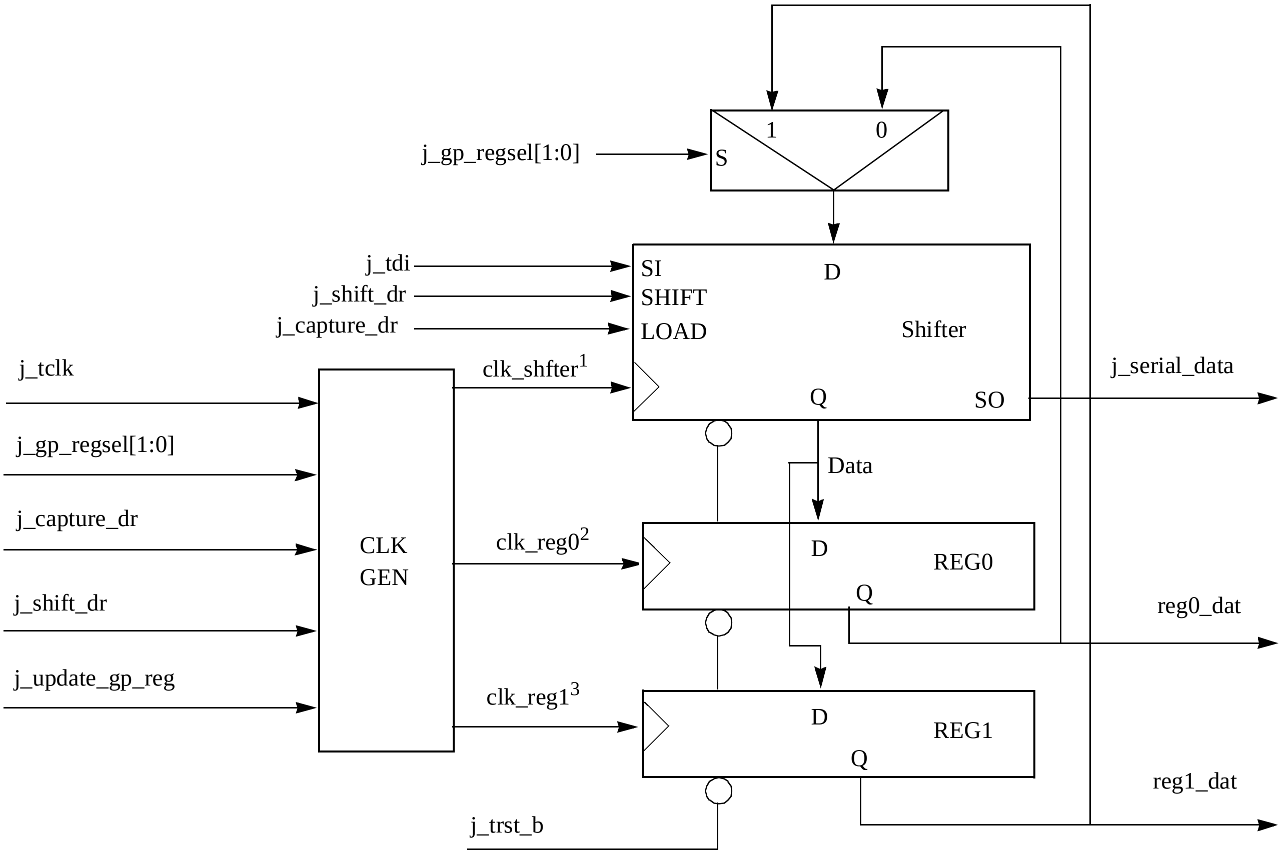
- — The outputs shown in Table 13-22 are a decode of the REGSEL[0:6] field in the once command register (OCMD). They are used to specify which external general purpose JTAG register to access through the e200 TAP controller.

Table 13-22. JTAG General Purpose Register Select Decoding

Signal Name	Type	Description
j_gp_regsel[0]	O	REGSEL[0:6]=7'h70
j_gp_regsel[1]	O	REGSEL[0:6]=7'h71
j_gp_regsel[2]	O	REGSEL[0:6]=7'h72
j_gp_regsel[3]	O	REGSEL[0:6]=7'h73
j_gp_regsel[4]	O	REGSEL[0:6]=7'h74
j_gp_regsel[5]	O	REGSEL[0:6]=7'h75
j_gp_regsel[6]	O	REGSEL[0:6]=7'h76
j_gp_regsel[7]	O	REGSEL[0:6]=7'h77
j_gp_regsel[8]	O	REGSEL[0:6]=7'h78
j_gp_regsel[9]	O	REGSEL[0:6]=7'h79

- Enable Once Register Select (j_en_once_regsel)
  - — 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.
  - — May be used by external security logic to assist with controlling the **jd_enable_once** input signal.
    - – The external enable_OnCE register should be muxed onto the **j_serial_data** input (see Section •, "Serial Data (j_serial_data)").
    - – During the Shift_DR state, **j_serial_data** is supplied to the **j_tdo** output.
- External Nexus Register Select (j_nexus_regsel)
  - — Asserted when a decode of the REGSEL[0–6] field in the OCMD indicates an external Nexus register is selected (0b1111100 encoding) for access through the e200 TAP controller.
- External LSRL Register Select (j_lsrl_regsel)
  - — Asserted when a decode of the REGSEL[0–6] field in the OCMD indicates an external LSRL register is selected (0b1111101 encoding) for access through the e200 TAP controller.
- Serial Data (j_serial_data)
  - — Receives serial data from external JTAG registers.
  - — All external registers share this 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. D
  - — Data is internally routed to **j_tdo**.

Figure 13-2 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 13-2. Example External JTAG Register Design

13.3.25 JTAG ID Signals

Table 13-23 shows the JTAG ID register unique to Freescale as specified by the IEEE 1149.1 JTAG Specification. Note that bit 31 is the MSB of this register.

Table 13-23. JTAG Register ID Fields

Bit Field	Type	Description	Value
[31–28]	Variable	Version Number	Variable
[27–22]	Fixed	Design Center Number (ZEN)	6’b011111
[21–12]	Variable	Sequence Number	Variable

Table 13-23. JTAG Register ID Fields (Continued)

Bit Field	Type	Description	Value
[11–1]	Fixed	Freescale Manufacturer ID	11’b00000001110
0	Fixed	JTAG ID Register Identification Bit	1’b1

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 13-24 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 13-24. 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

13.3.25.1 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.

13.3.25.2 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.

13.3.25.3 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.

13.4 Timing Diagrams

The following subsections provide timing diagrams.

13.4.1 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 13-3 shows an example of a free-running half-speed HCLK relative to **m_clk**.

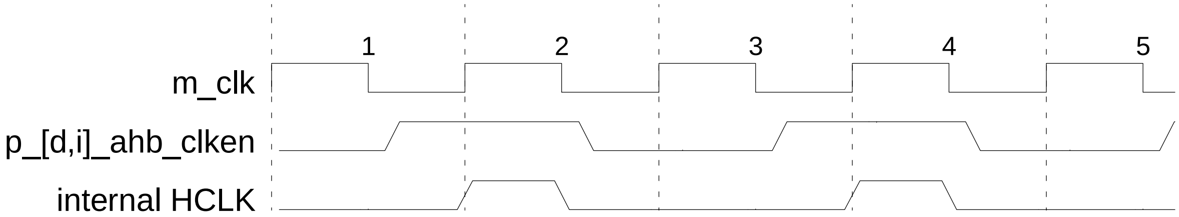


Figure 13-3. AHB Clock Enable Operation—1

Figure 13-4 shows an example of a free-running 1/3 speed HCLK relative to **m_clk**.

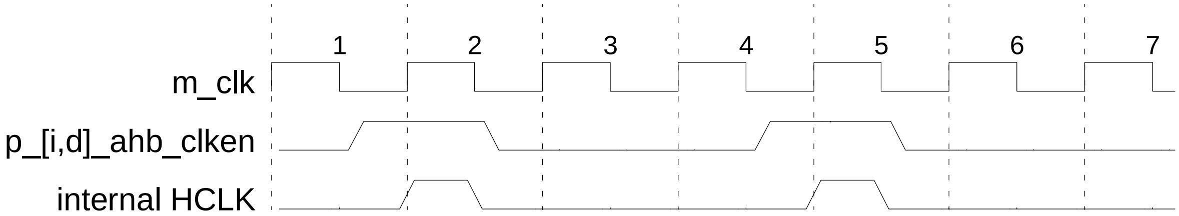


Figure 13-4. AHB Clock Enable Operation—2

Figure 13-5 shows an example of a non-periodic HCLK, used for power reduction, relative to **m_clk**.

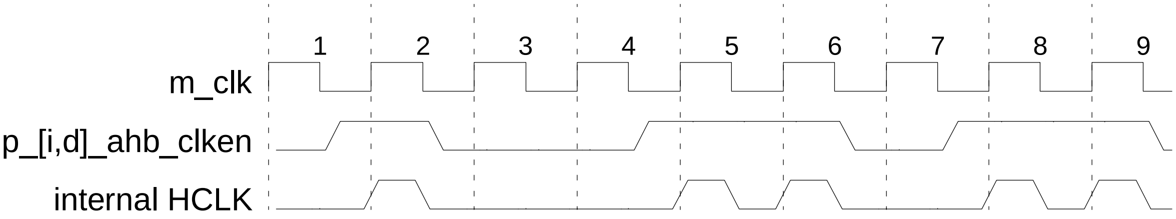


Figure 13-5. AHB Clock Enable Operation—3

13.4.2 Processor Instruction/Data Transfers

Transferring 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, which support 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 that 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 are 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, for example **m_clk** qualified by **p_[i,d]_ahb_clken**

13.4.2.1 Basic Read Transfer Cycles

During a read transfer, the core receives data from a memory or peripheral device. Figure 13-6 illustrates functional timing for basic read transfers, with clock-by-clock descriptions of the activity in the following subsections.

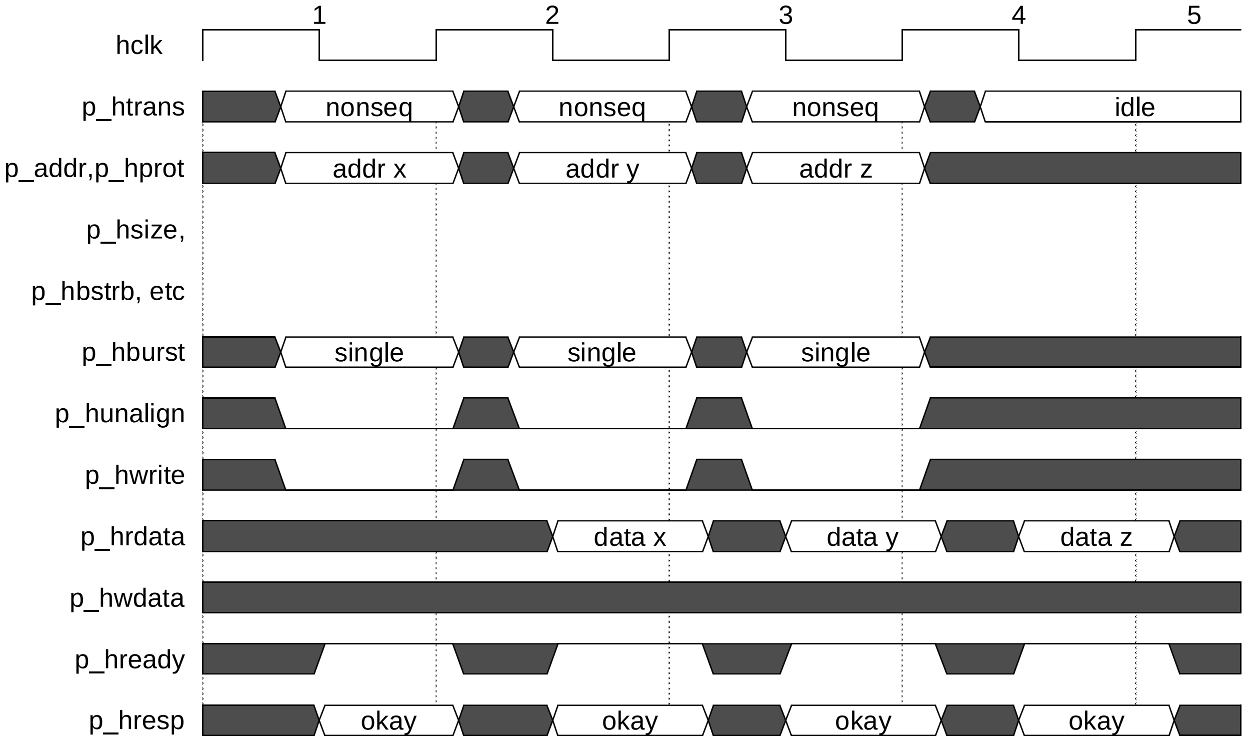


Figure 13-6. Basic Read Transfers

13.4.2.1.1 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 attribute (**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.

13.4.2.1.2 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.

13.4.2.1.3 Clock 3 (C3)

During C3, the $addr_y$ memory access takes place using the address and attribute values which were driven during C2 to enable reading of 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), and since the access to $addr_y$ is completing, it is considered taken at the end of C3.

13.4.2.1.4 Clock 4 (C4)

During C4, the $addr_z$ memory access takes place using the address and attribute values which were driven during C3 to enable reading of 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 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.

13.4.2.2 Read Transfer with Wait State

Figure 13-7 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 since 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 13-7 shows the read transfer with wait-state.

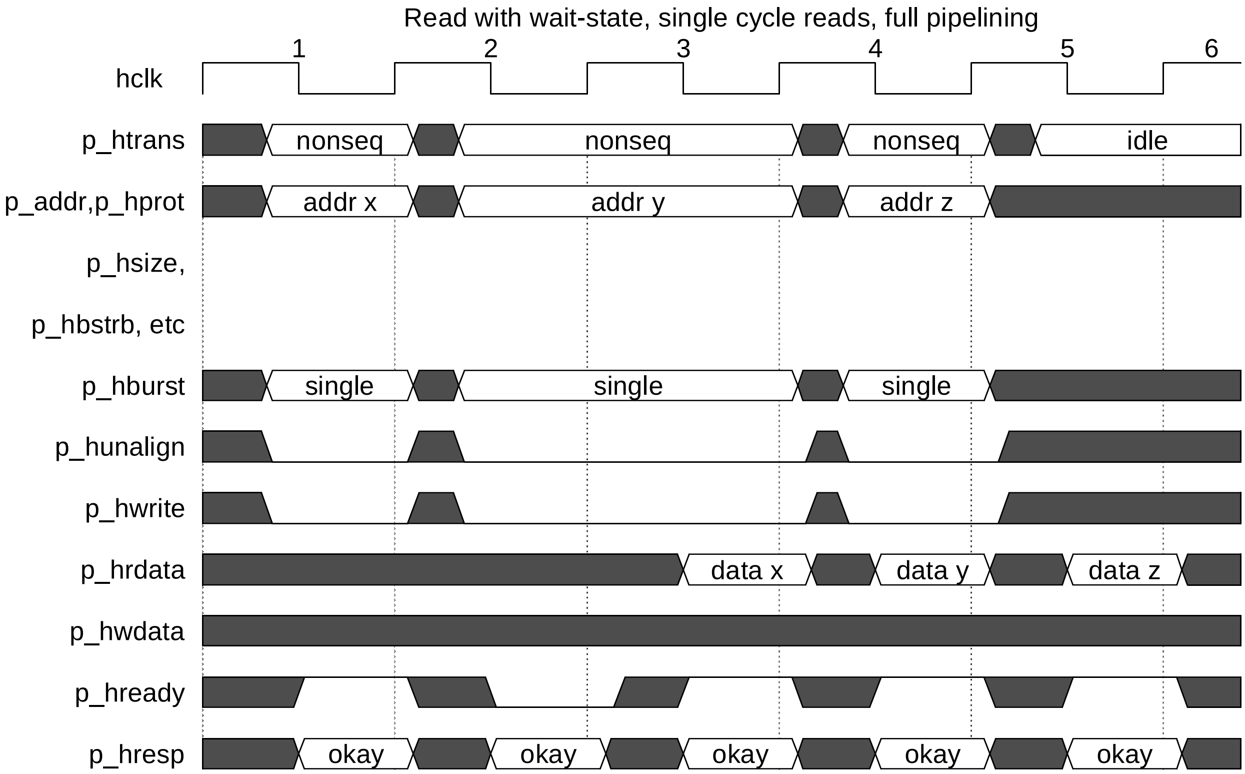


Figure 13-7. Read Transfer with Wait-State

13.4.2.3 Basic Write Transfer Cycles

During a write transfer, the core provides write data to a memory or peripheral device. Figure 13-8 illustrates functional timing for basic write transfers, with clock-by-clock descriptions of the activity in the following subsections.

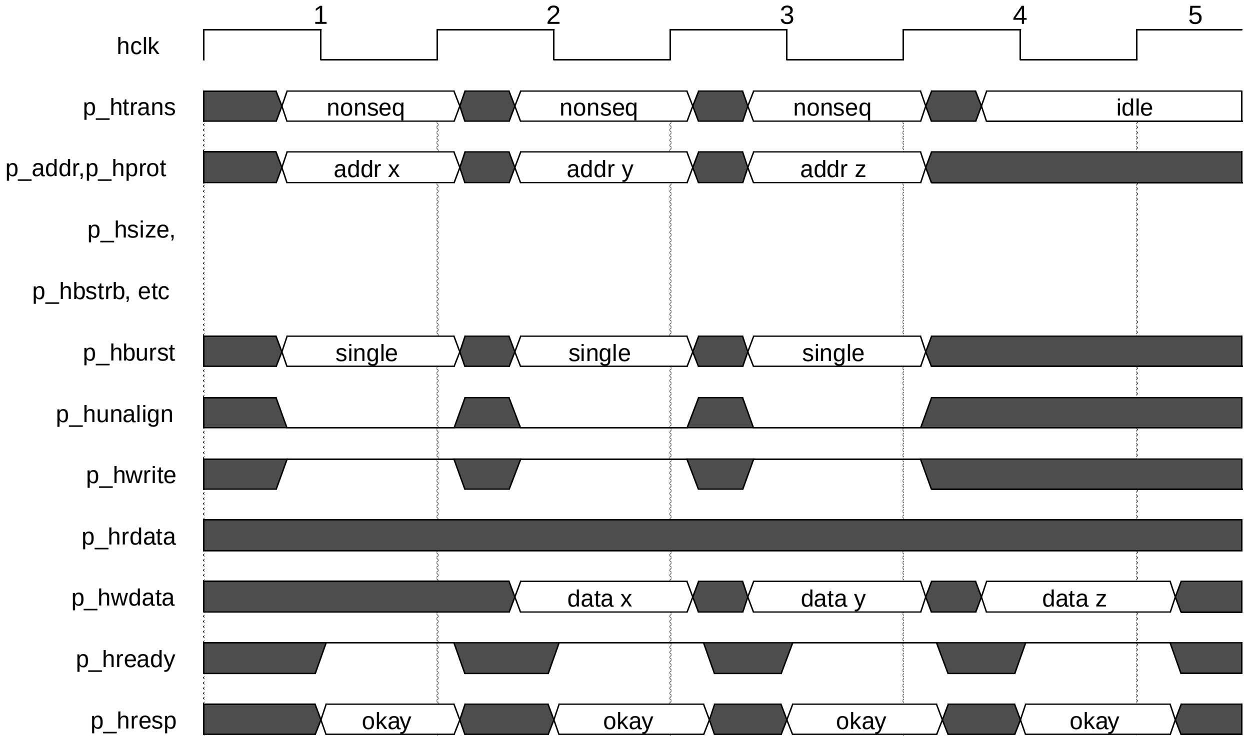


Figure 13-8. Basic Write Transfers

13.4.2.3.1 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]**) 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 high for a write 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 write 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.

13.4.2.3.2 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.

13.4.2.3.3 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 that 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.

13.4.2.3.4 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.

13.4.2.4 Write Transfer with Wait States

Figure 13-9 shows an example of write-wait-state operation.

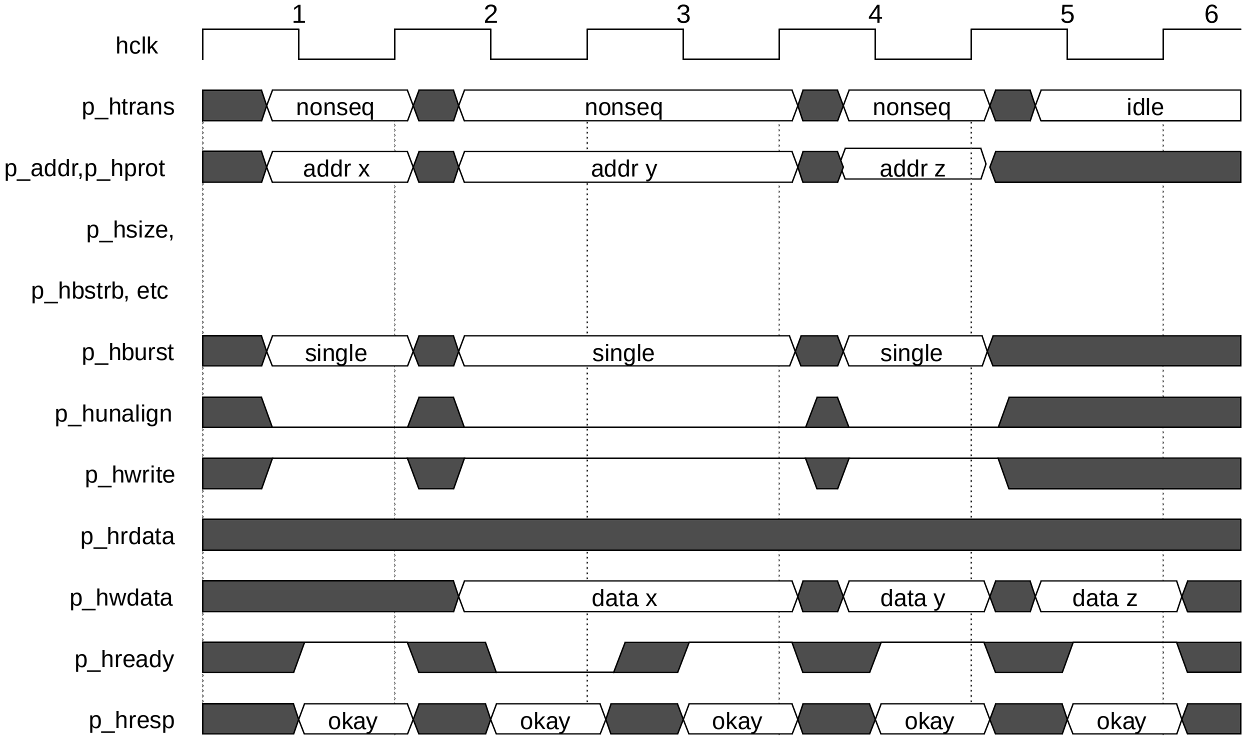


Figure 13-9. Write Transfer with Wait-State

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, 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.

13.4.2.5 Read and Write Transfers

Figure 13-10 shows a sequence of read and write cycles.

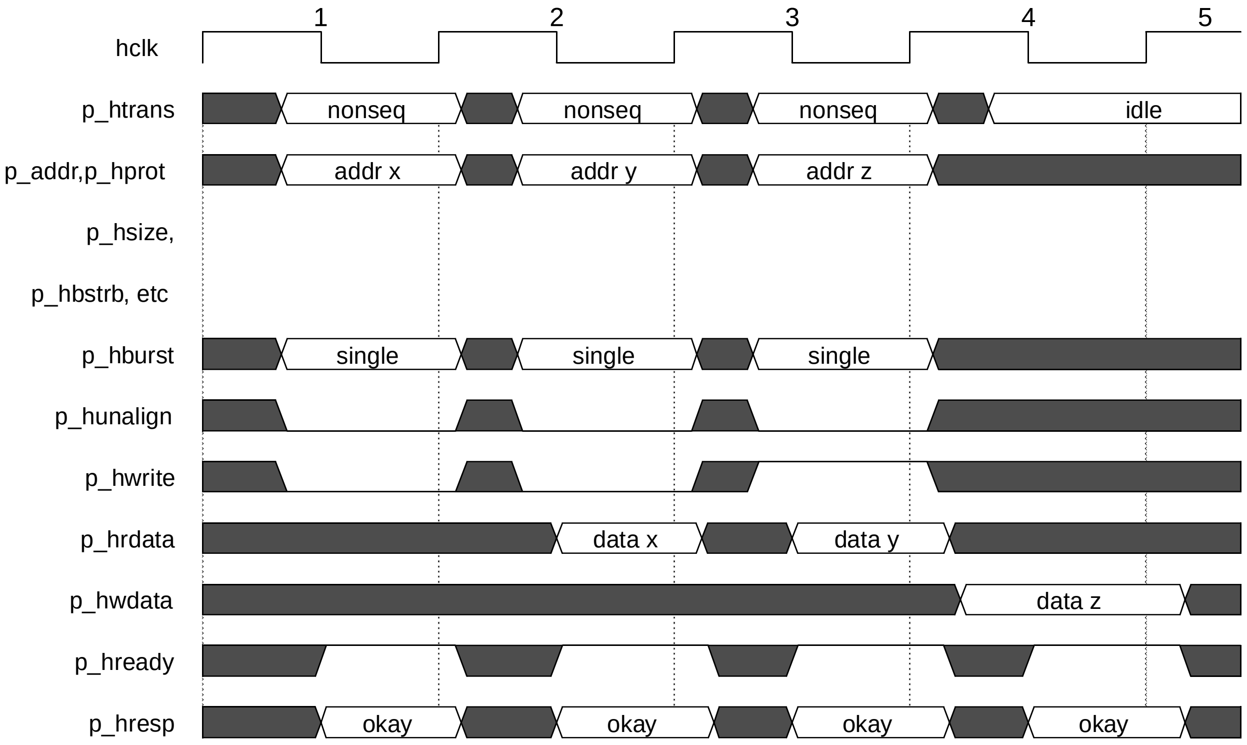


Figure 13-10. Single Cycle Read and Write Transfers

The sequence 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 taken at the end of C2 since a ready/OKAY response is asserted during C2 for the first read access (addr_x).
3. During C3, a request is generated for a write to addr_y, which is taken at the end of C3 because the second access is terminating.
4. 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 13-11 shows another sequence of read and write cycles. This example shows an interleaved write access between two reads.

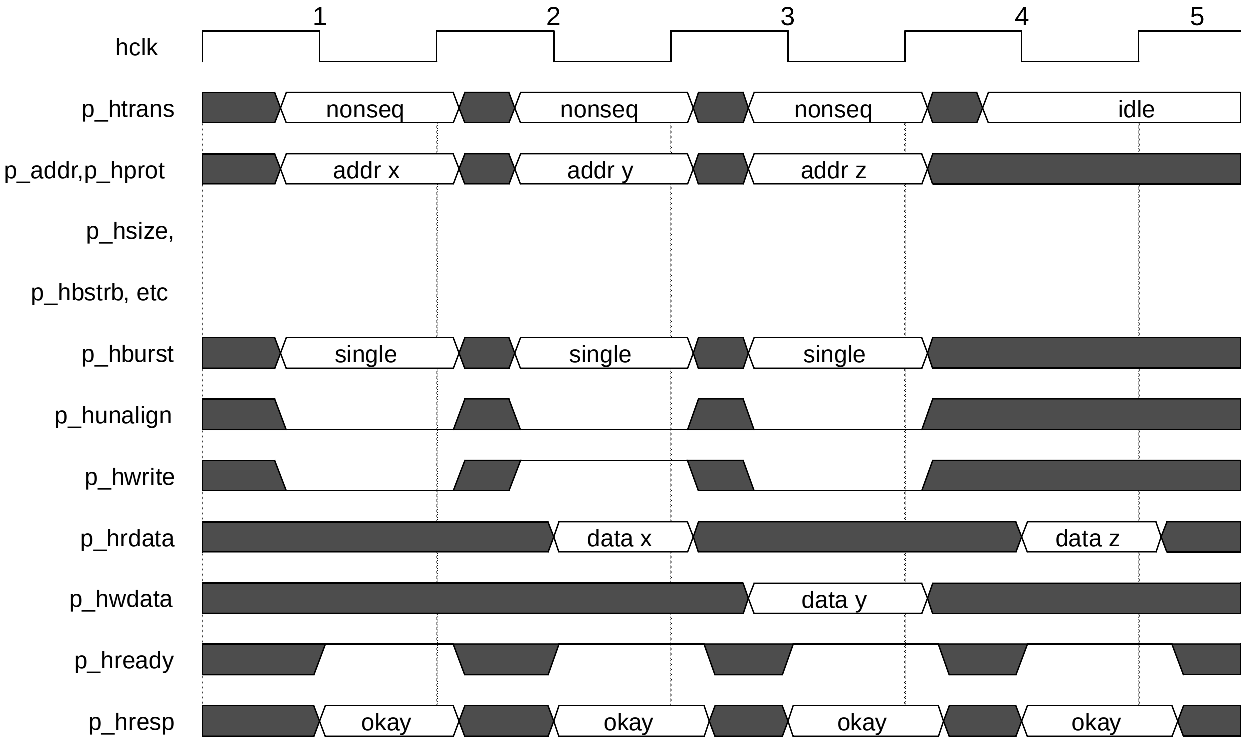


Figure 13-11. Single Cycle Read and Write Transfers—2

The sequence 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.
4. 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.
5. During C4, the $addr_y$ write access is terminated, and no further access is requested.

Figure 13-12 shows another sequence of read and write cycles. In this example, reads incur a single wait state.

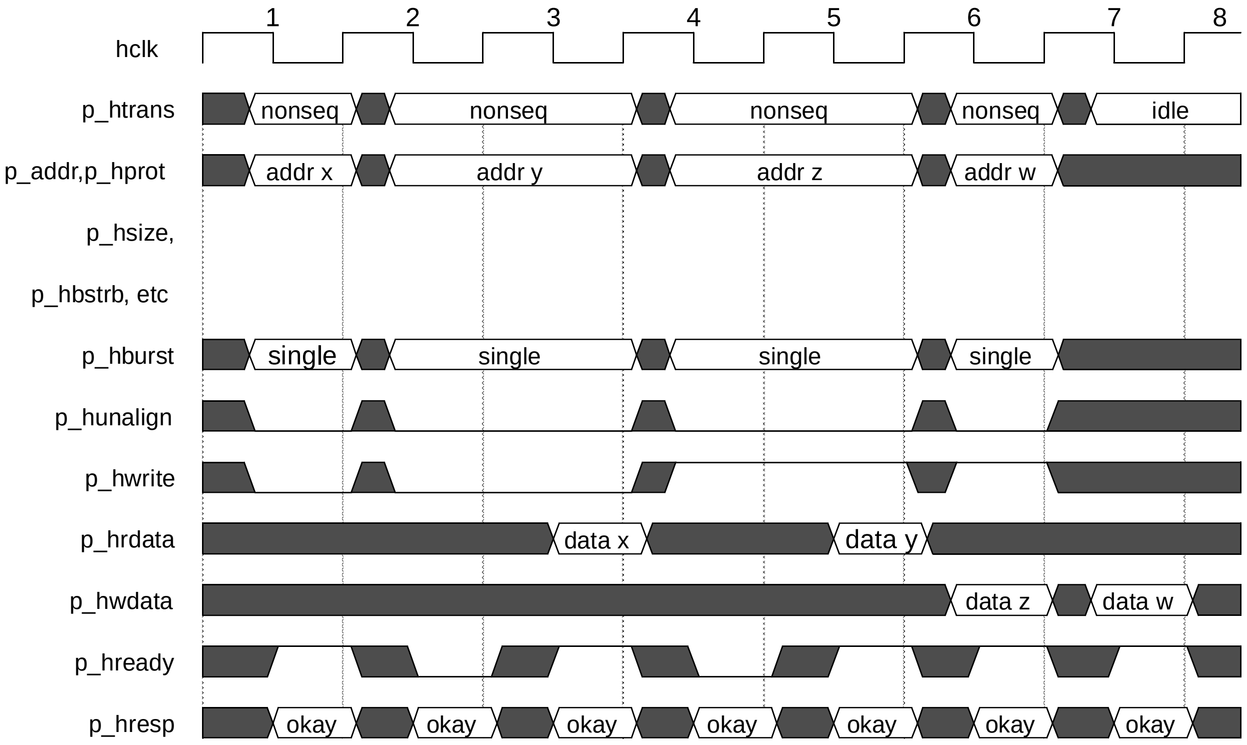


Figure 13-12. Multi-Cycle Read and Write Transfers

The sequence 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 ($addr_y$).
4. During C5, the request for a write to $addr_z$ is taken because the second access is terminating.
5. Data for the $addr_z$ write cycle is driven in C6, the cycle after the access is taken.
6. During C6, the $addr_z$ write access is terminated and the $addr_w$ write request is taken.
7. 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 13-13 shows another sequence of read and write cycles. In this example, reads incur a single wait state.

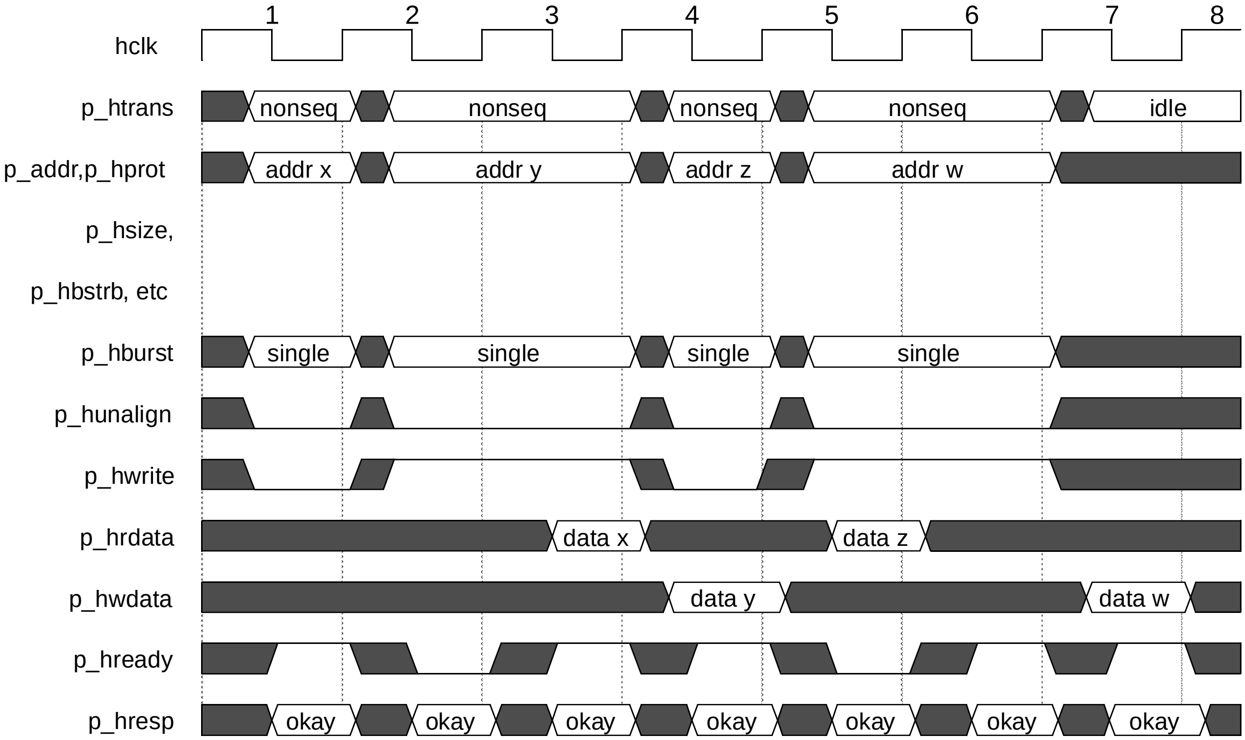


Figure 13-13. Multi-Cycle Read and Write Transfers—2

The sequence 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 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. 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 because 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.
8. 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.

13.4.2.6 Misaligned Accesses

Figure 13-14 illustrates functional timing for a misaligned read transfer. The read to addr$_x$ is misaligned across a 64-bit boundary.

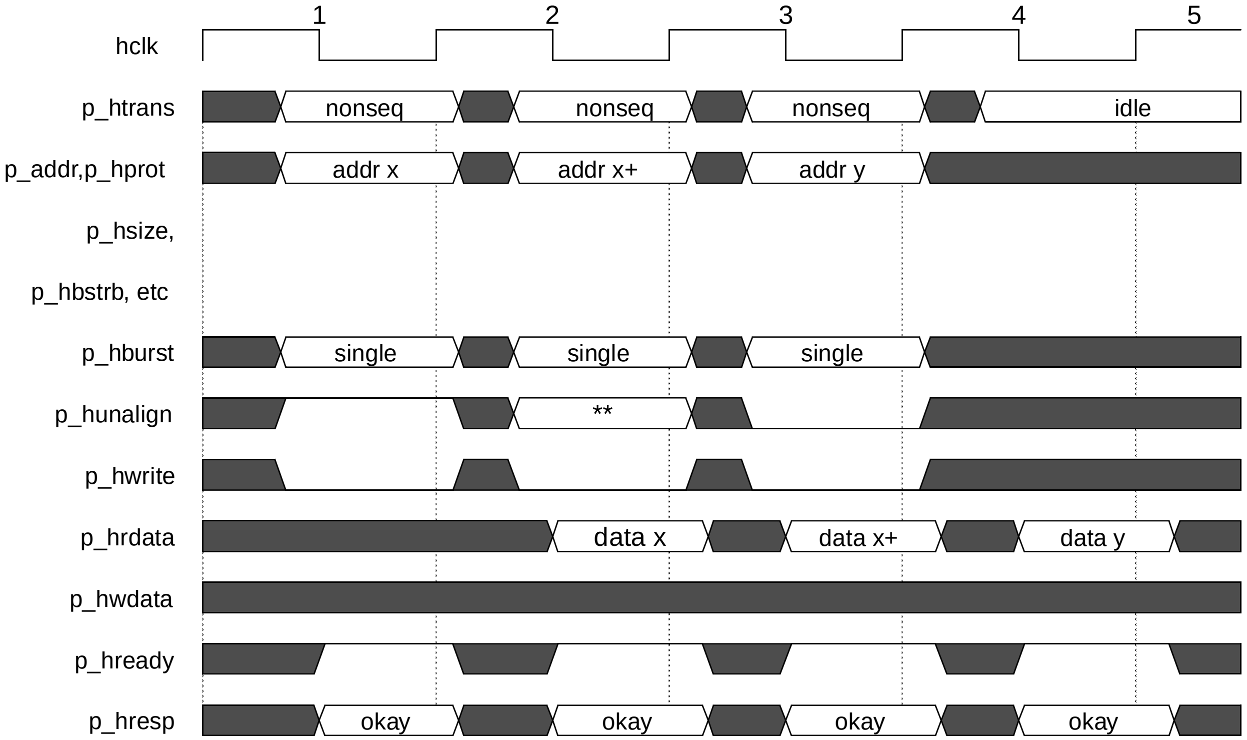


Figure 13-14. Misaligned Read Transfer

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 is aligned to the next higher 64-bit boundary), and because 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 13-15 illustrates functional timing for a misaligned write transfer. The write to addr$_x$ is misaligned across a 64-bit boundary.

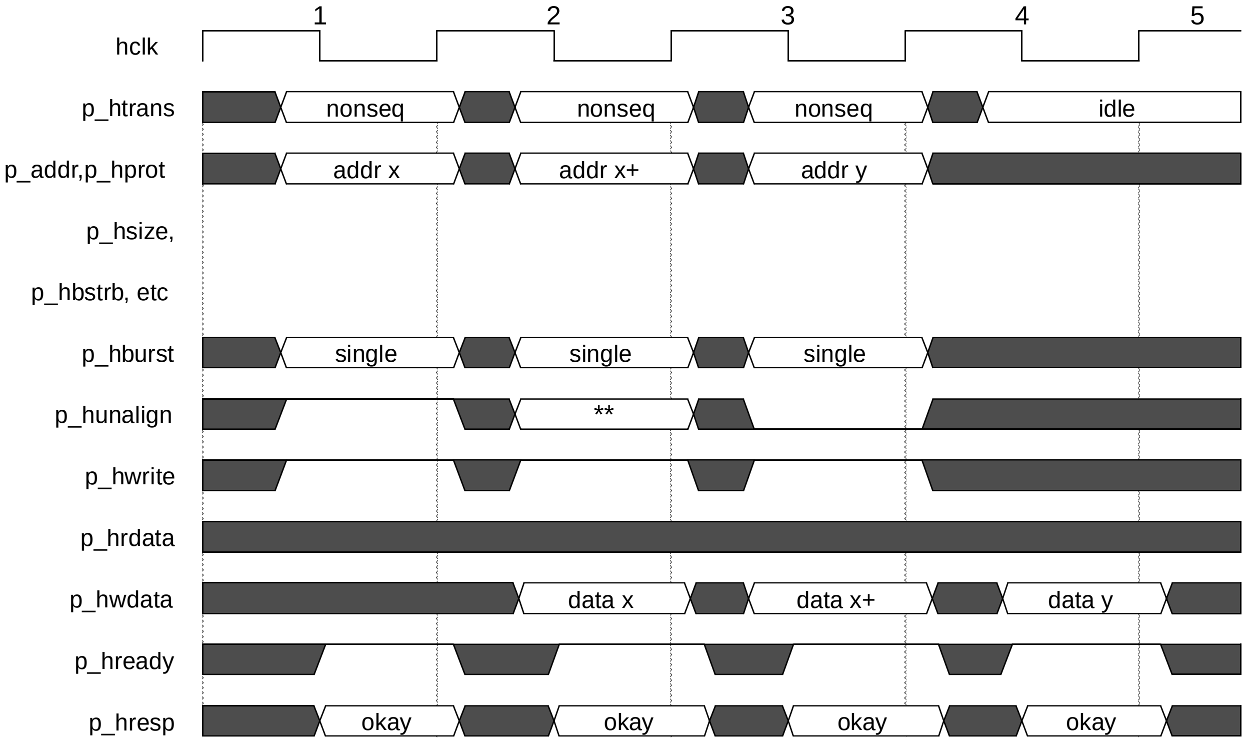


Figure 13-15. Misaligned Write Transfer

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 is 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, because this access is aligned.

An example of a misaligned write cycle followed by an aligned read cycle is shown in Figure 13-16. It is similar to the previous example in Figure 13-15.

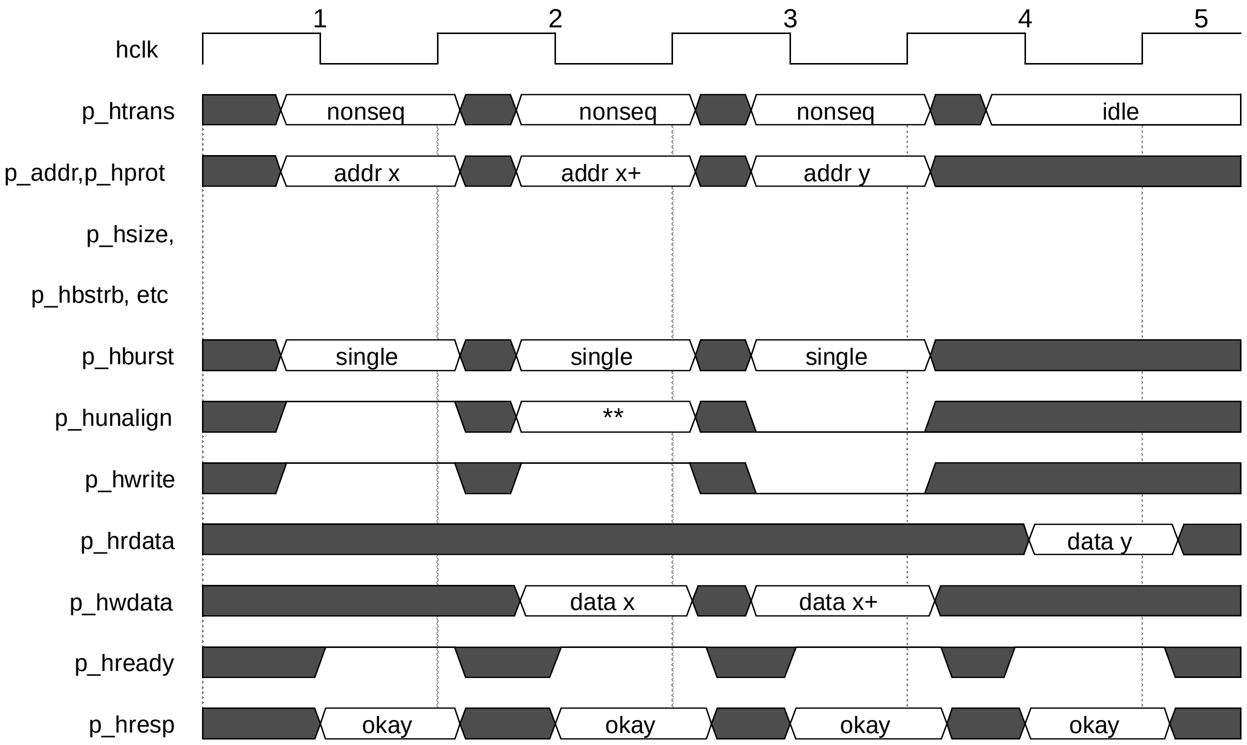


Figure 13-16. Misaligned Write, Single Cycle Read Transfer

13.4.2.7 Burst Accesses

The following paragraphs describe burst read and burst write accesses on the AHB. Burst write accesses are shown for reference only. The e200z446n3 does not generate burst write transfers.

Figure 13-17 illustrates functional timing for a burst read transfer.

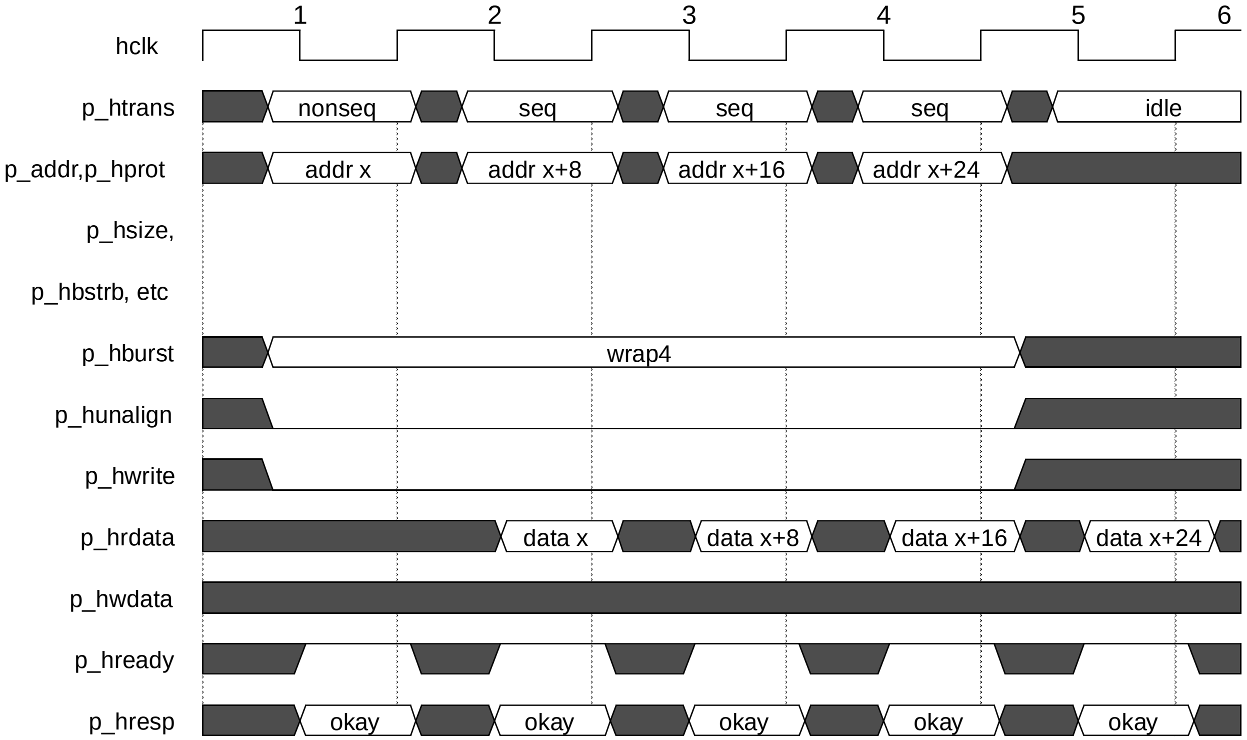


Figure 13-17. Burst Read Transfer

The p_hburst signals indicate WRAP4 for all burst transfers. The **p_hunalign** signal is negated. **p_hsize** indicate 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 13-18 illustrates functional timing for a burst read with wait-state transfer.

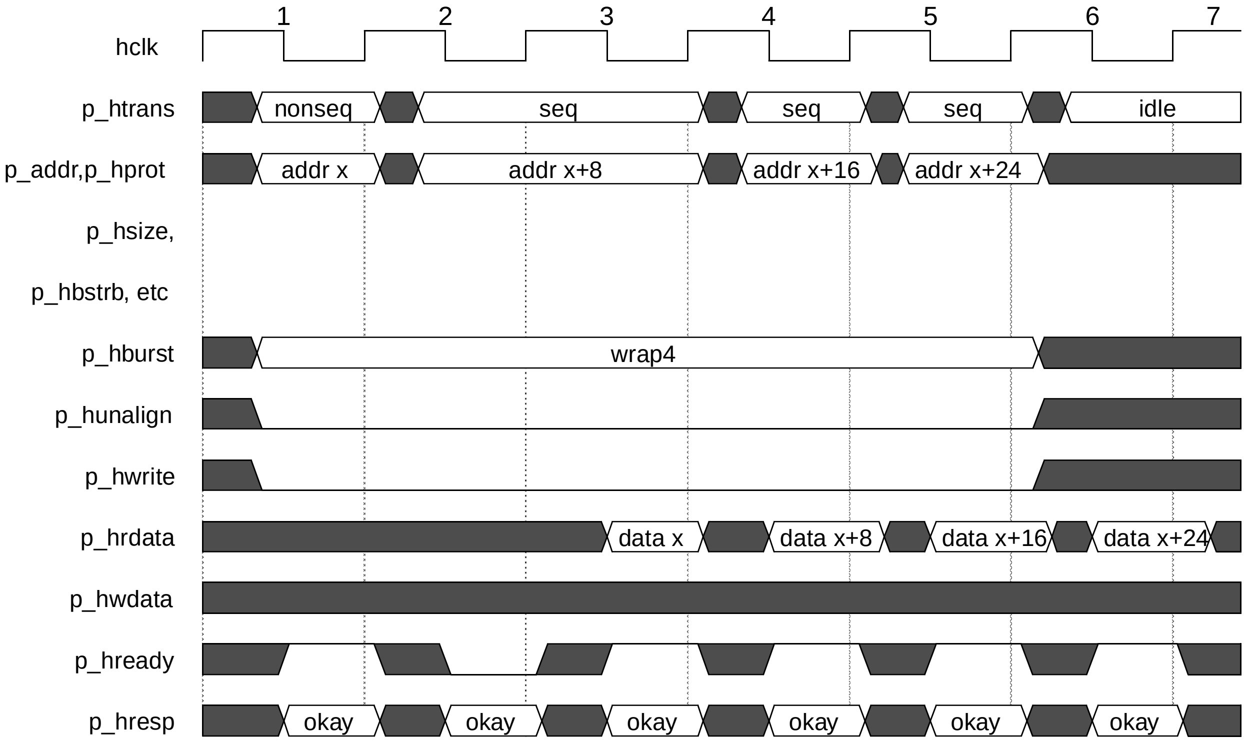


Figure 13-18. Burst Read with Wait-State Transfer

The first cycle of the burst incurs a single wait-state.

Figure 13-19 illustrates functional timing for a burst write transfer.

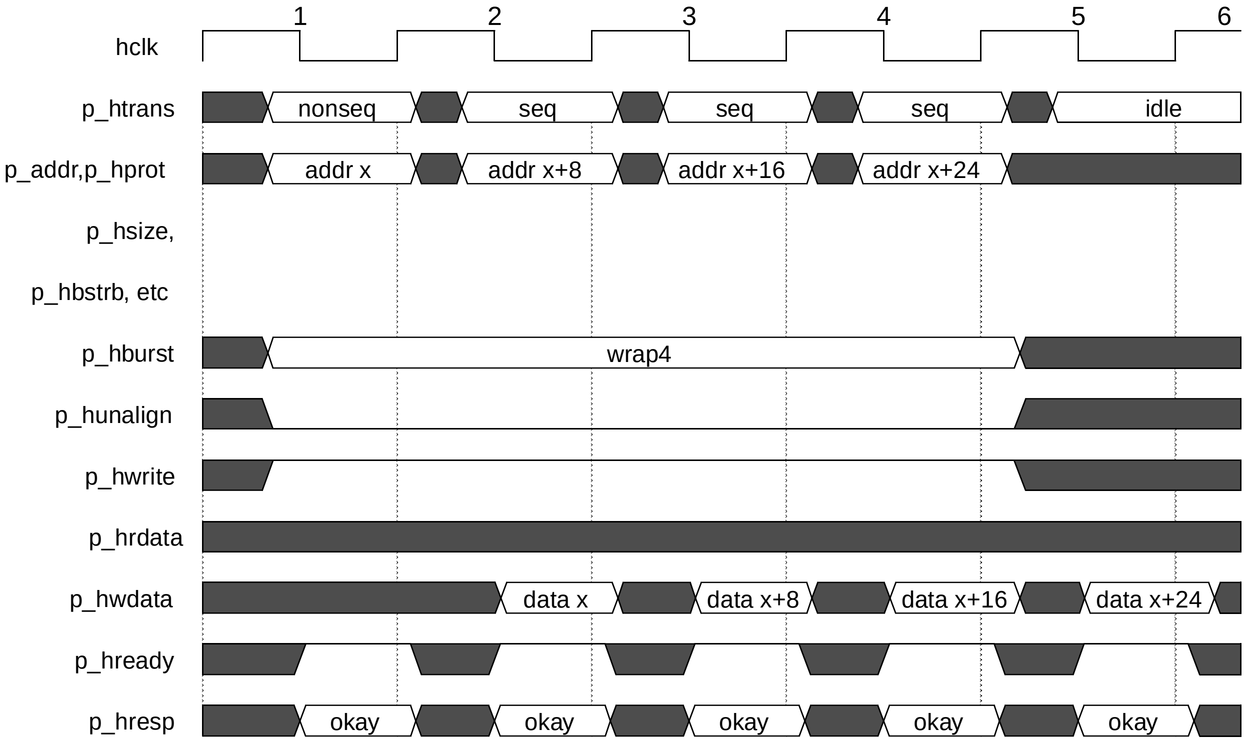


Figure 13-19. Burst Write Transfer

Figure 13-18 illustrates functional timing for a burst write with wait-state transfer.

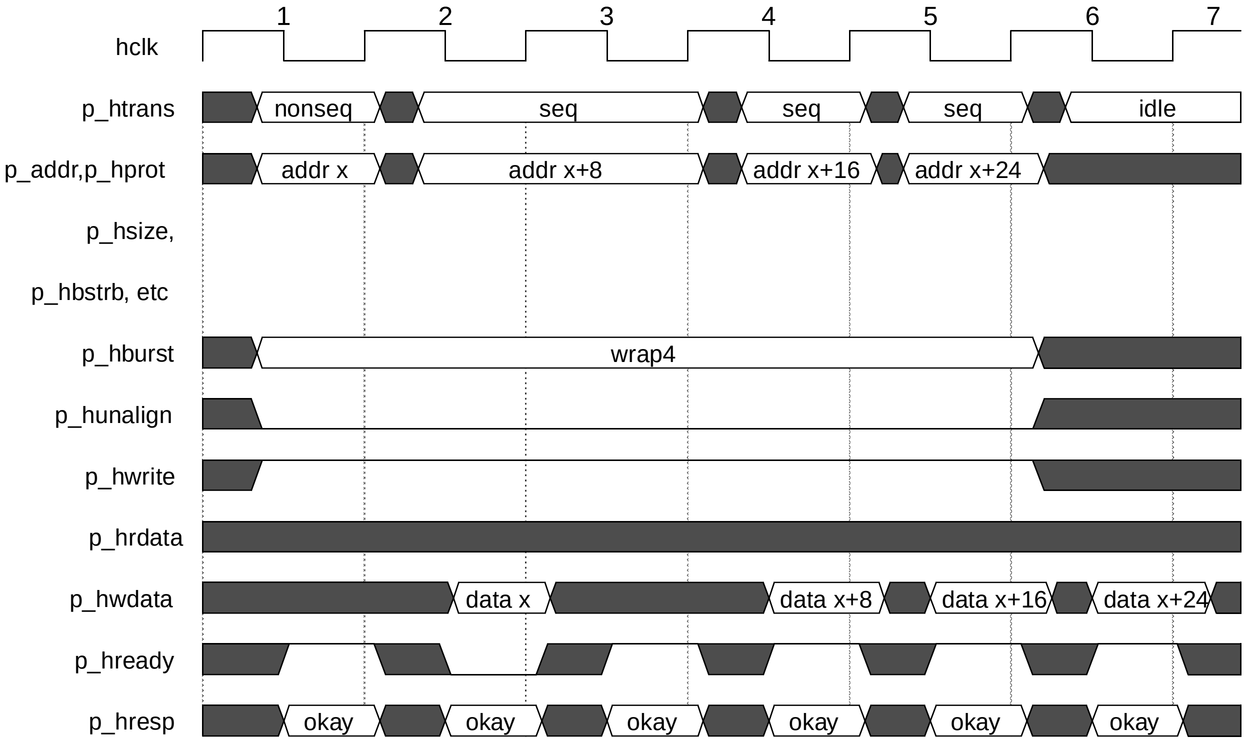


Figure 13-20. Burst Write with Wait-State Transfer

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.

13.4.2.8 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, and during the second cycle, 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 may change to any other value, and a new access unrelated to the pending access may be requested. The cycle which may have been previously pending while waiting for a response which terminates with error may be changed. It is not required to remain unchanged when an error response is received.

Figure 13-21 shows an example of error termination.

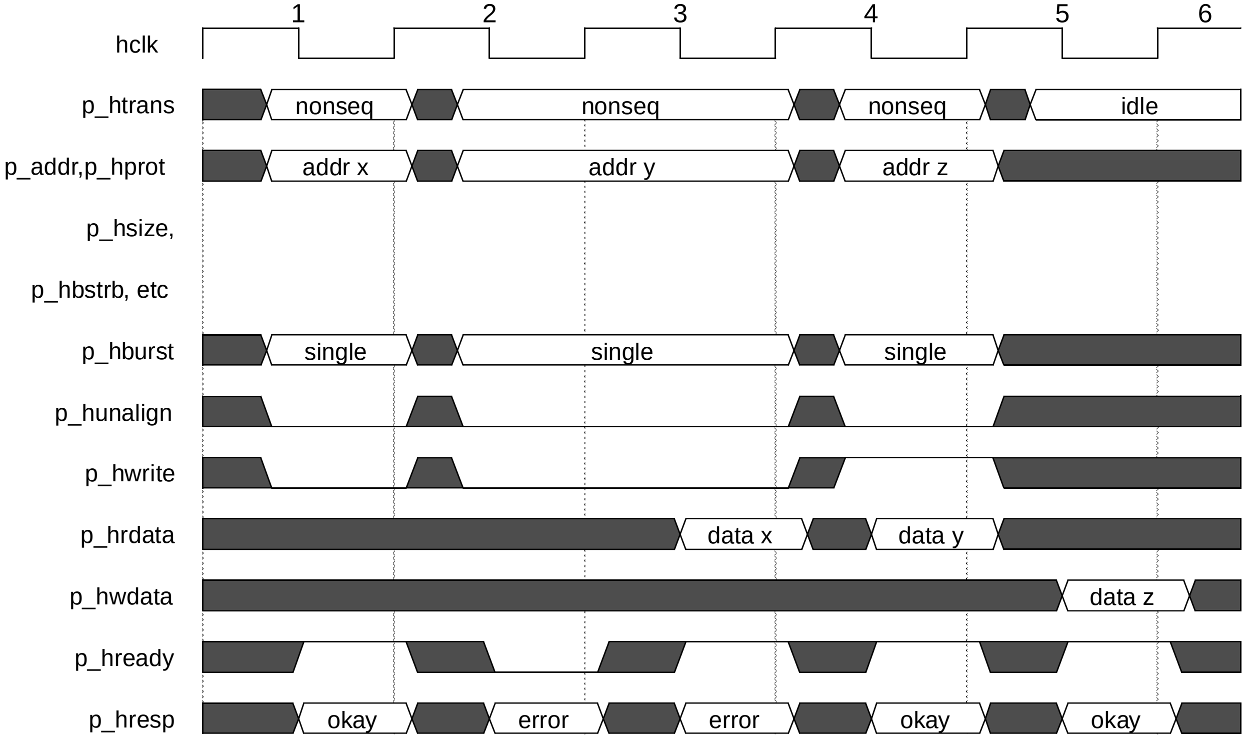


Figure 13-21. Read and Write Transfers, Instr. Read Error Termination

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.
5. During C4, read data is supplied for the addr$_y$ read, and the access is terminated normally during C4.
6. Also during C4, a request is generated for a write to addr$_z$, which is taken at the end of C4 because the second access is terminating.
7. Data for the addr$_z$ write cycle is driven in C5, the cycle after the access is taken.
8. During C5, a ready/OKAY response is signaled to complete the write cycle to addr$_z$.

9. 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 13-22 shows another example of error termination.

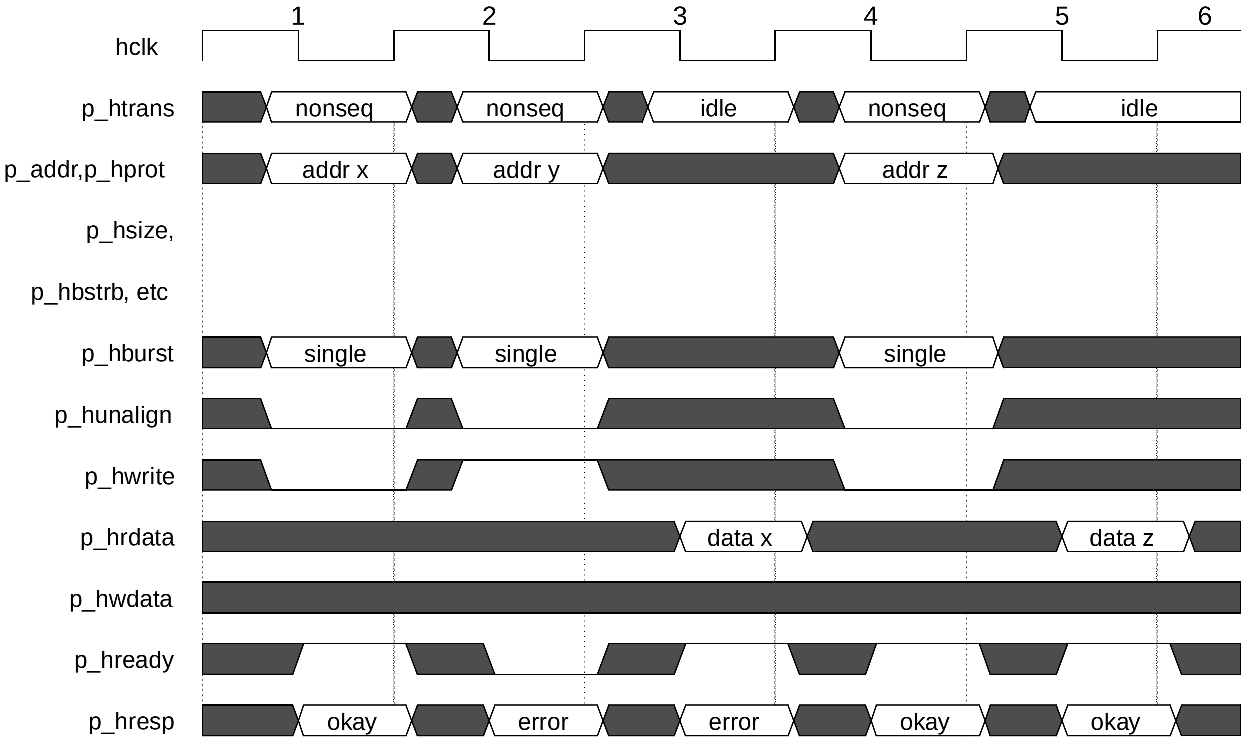


Figure 13-22. Data Read Error Termination

The sequence 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.
6. In this example of error termination, a subsequent access was aborted.

Figure 13-23 shows another example of error termination, this time on the initial portion of a misaligned write.

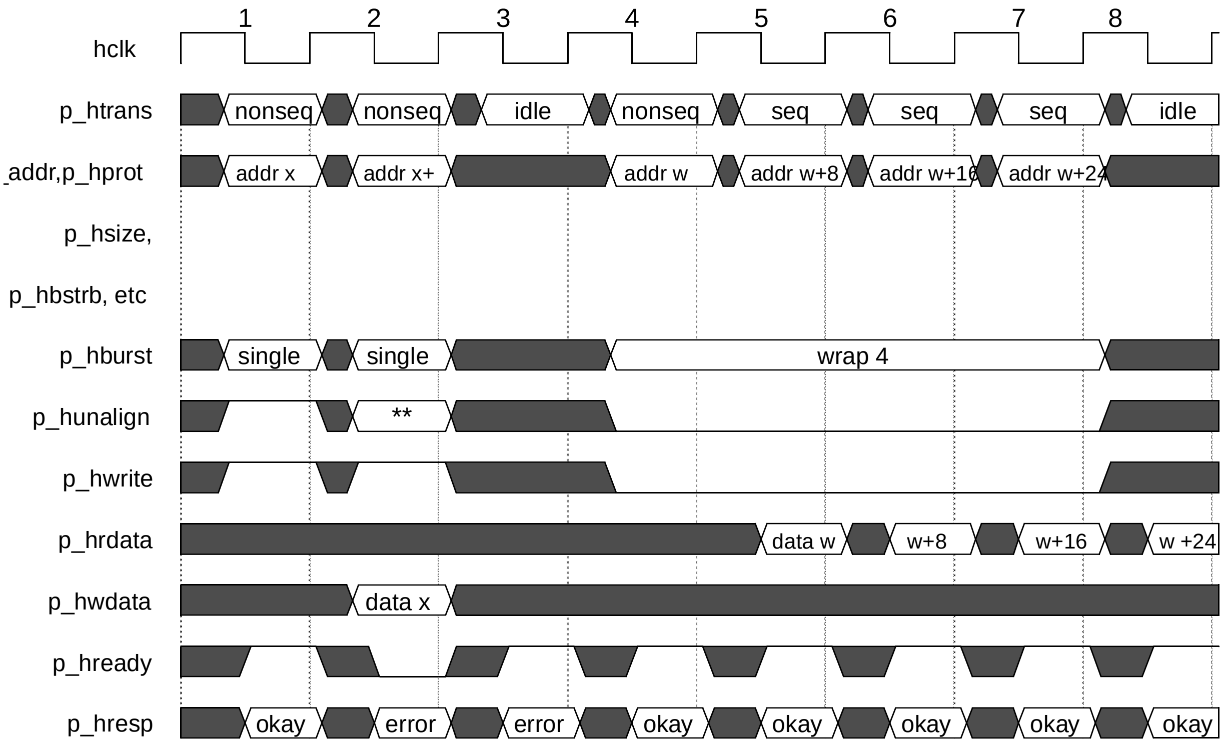


Figure 13-23. Misaligned Write Error Termination, Burst Substituted

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 13-24 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.

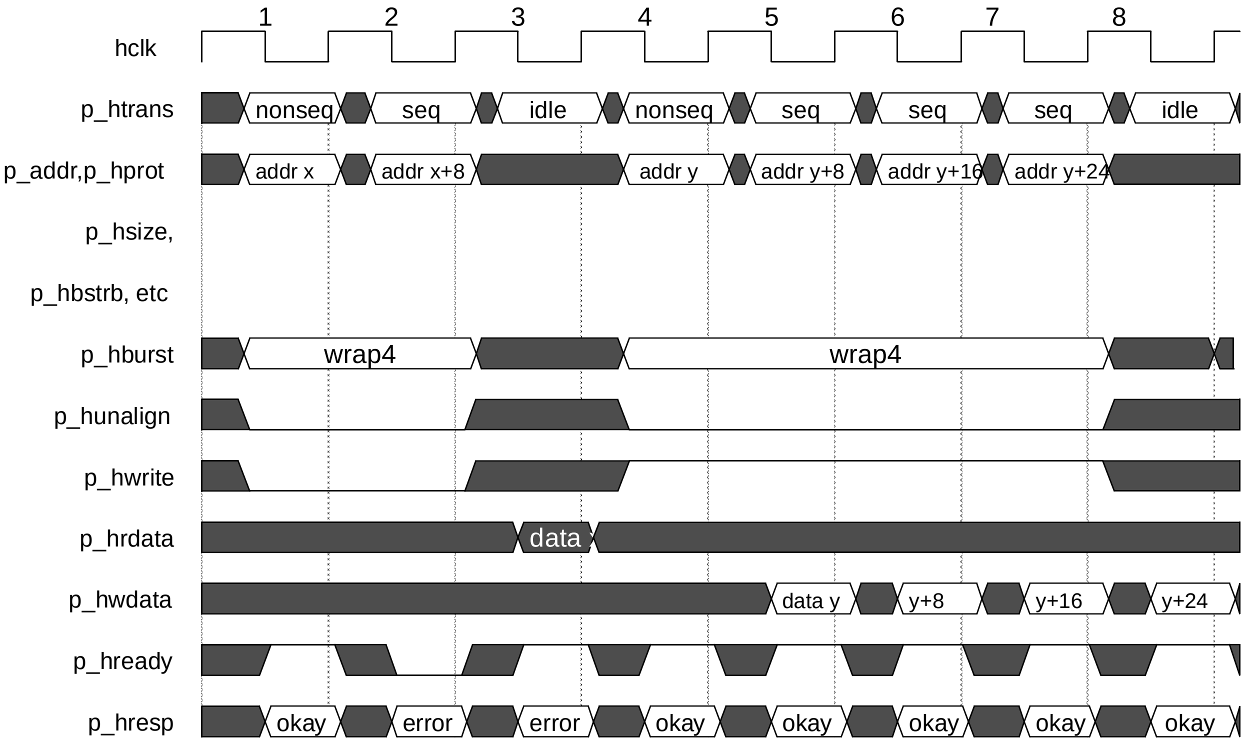


Figure 13-24. Burst Read Error Termination, Burst Write Substituted

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.

13.4.3 Cache Error Cross-Signaling Operation

The cache error cross-signaling interface is provided to allow for lockstep operation of two or more CPUs in the presence of cache parity/EDC errors. The interface provides a means for 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 error is indicated by means of the **p_cache_tagerr_out** and **p_cache_dataerr_out** outputs, and the corresponding address and cache way(s) incurring the error are indicated with the **p_cerraddr_out[0:31]** and **p_cerrway_out[0:3]** outputs in the next cycle. In a dual-CPU lockstep system, these outputs are normally tied to the corresponding **p_cache_tagerr_in**, **p_cache_dataerr_in**, **p_cerraddr_in[0:31]**, and **p_cerrway_in[0:3]** inputs of the other CPU.

Normally, due to extremely low probability, it is not expected that the two CPUs would incur an error during the same lookup cycle. If this is an issue requiring detection, then system logic may be utilized to detect simultaneous assertion of more than one CPU’s error output signal(s) and perform appropriate error recovery, such as a reset operation.

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_err_in**, **p_cache_tagerr_in**, **p_cache_dataerr_in**, **p_tagerrway_in[0:3]**, and **p_drterrway_in[0:3]** inputs are ignored. In the examples which follow, is is assumed that p_lkstep_en has been properly asserted.

13.4.3.1 Cross-Signaling with Machine Check Operation Selected

Figure 13-25 illustrates functional timing for a cross-signaling operation by a CPU encountering an internal cache error with the error action indicating that a machine check should be generated (L1CSR1[ICEA] = 00). A cache error is detected in cycle 2 and results in a machine check exception being signaled.

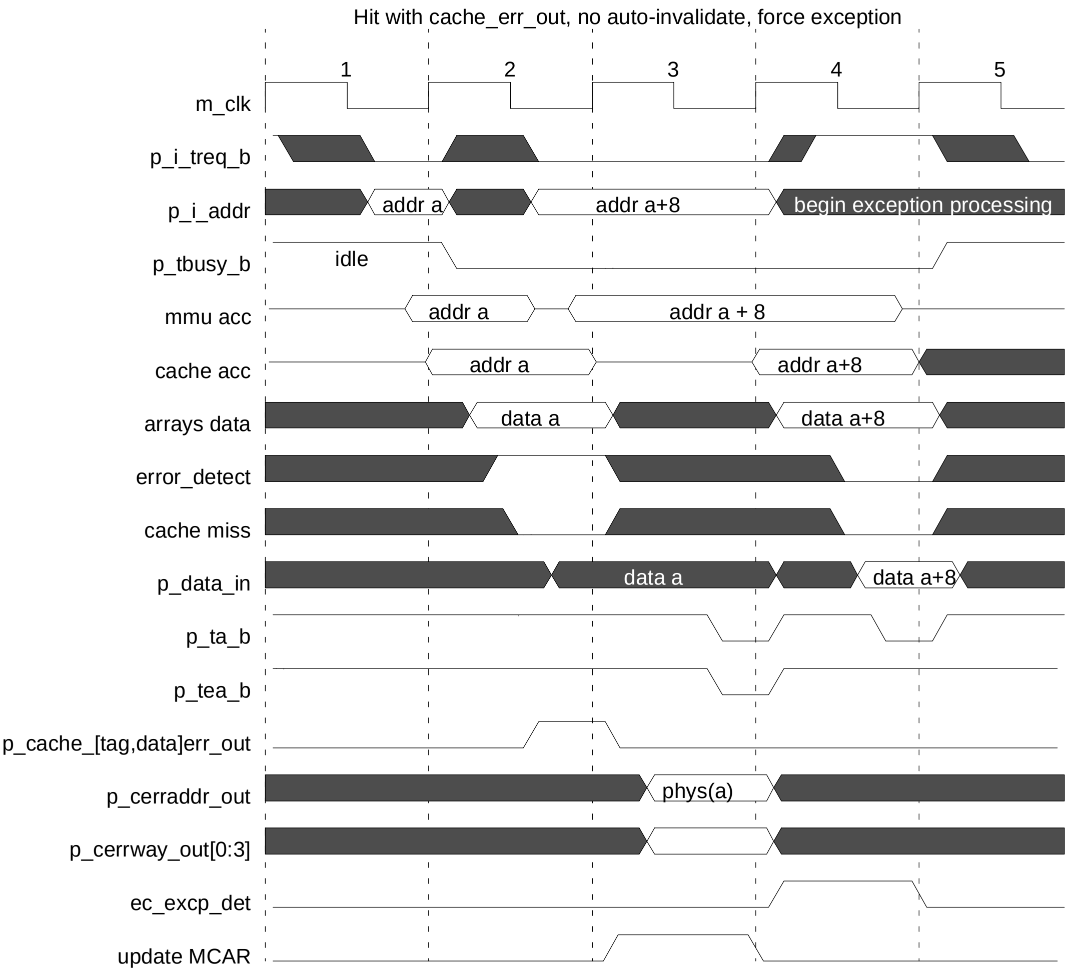


Figure 13-25. Cross-Signaling Exception Output Operation

For cross-signaling operations during **icbi** invalidate operations when machine check error action is selected (L1CSR1[ICEA] = ‘00’), the signaling of a **p_cache_tagerr_out** event indicates that a false hit to one or more unlocked line occurred and the line(s) should be invalidated in the other CPU(s), regardless of hit or miss conditions in the other CPU(s), rather than to cause a machine check condition. The ways(s) that incurred a false hit are signaled on the **p_cerrway_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 (L1CSR1[ICEA] = ‘00’).

Figure 13-26 illustrates functional timing for a cross-signaling operation by a CPU receiving a cache error cross-signaling operation with the error action indicating a machine check should be generated. A cache error is detected in cycle 2 by an external cache and results in a machine check exception being generated.

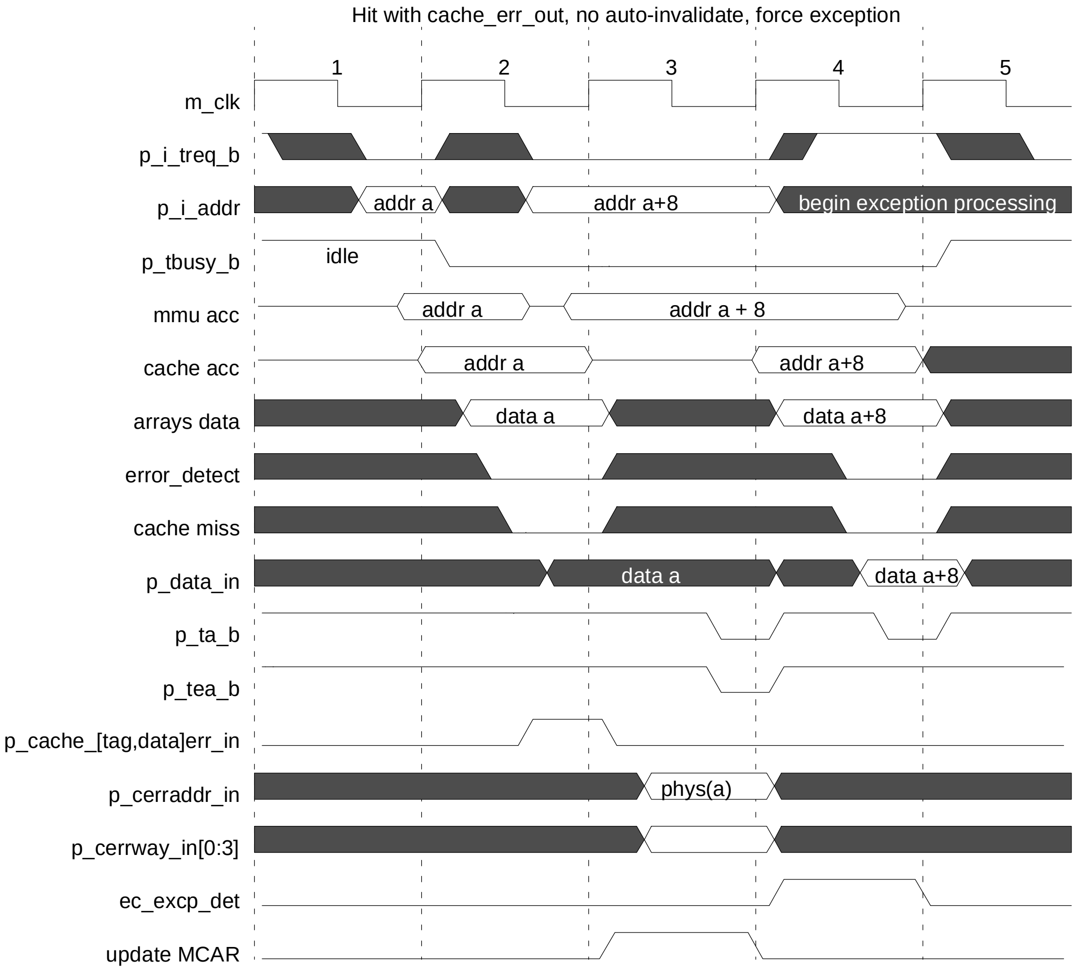


Figure 13-26. Cross-Signaling Exception Input Operation

13.4.3.2 Cross-Signaling with Auto-Invalidation Operation Selected

Figure 13-27 illustrates functional timing for a cross-signaling operation by a CPU which encounters an internal cache error in the cache data array with the error action indicating that an auto-invalidation should be generated. A cache data array error is detected in cycle 2 and results in a cache miss being forced. The error entry is refilled beginning in cycle 3.

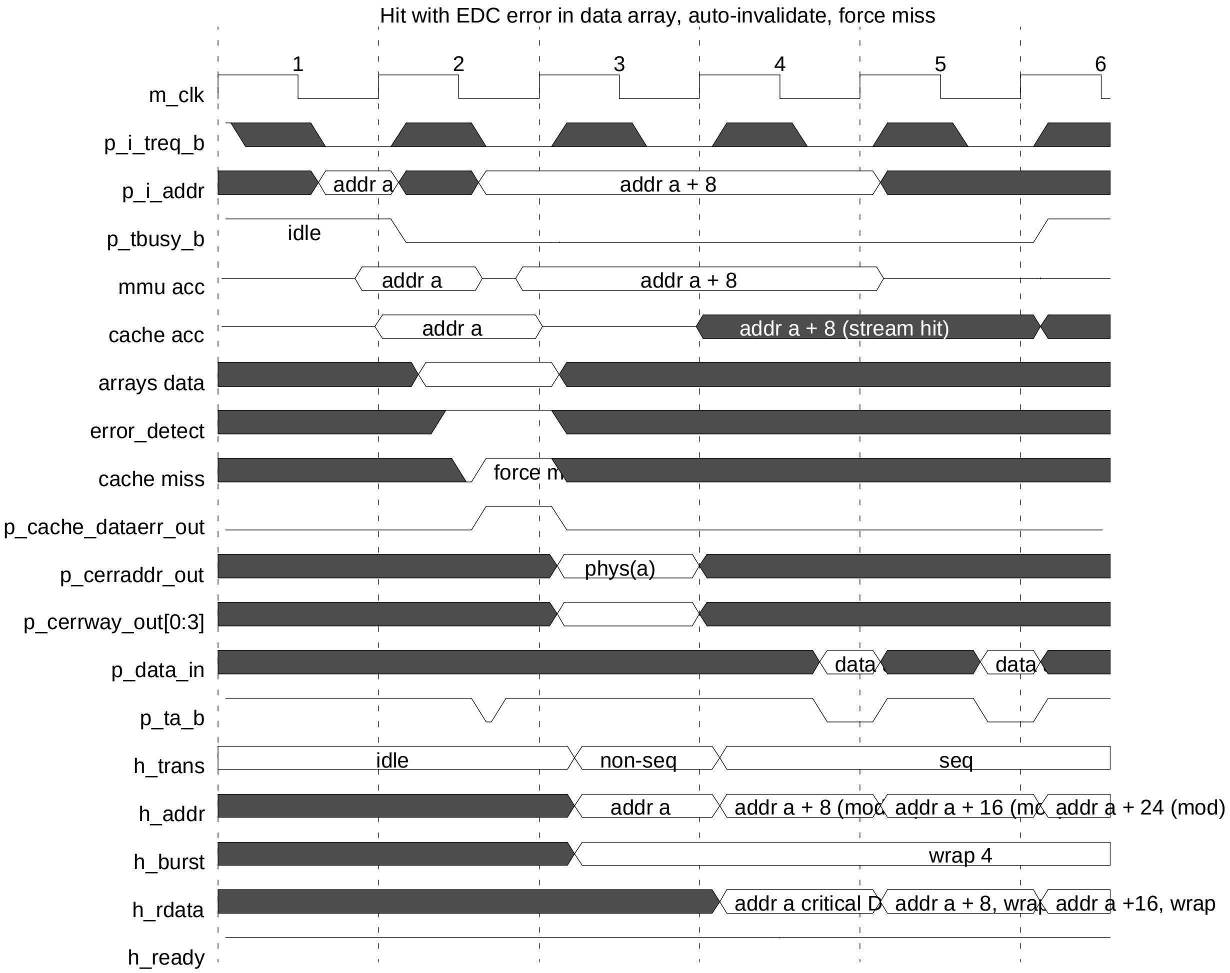


Figure 13-27. Cross-Signaling Invalidation Output Operation—Data Error

Figure 13-28 illustrates functional timing for a cross-signaling operation by a CPU encountering an internal cache error in the cache tag array with the error action indicating that an auto-invalidation should be generated. A cache tag array error is detected in cycle 2 and results in a cache correction/invalidation cycle being forced. In this example, way 0 has a correctable error and way 2 has an uncorrectable error and requires invalidation. The correction of way 0 and invalidation of way 2 are performed in cycle 4. The address ‘a’ access is recycled in cycle 5 and results in a miss. The cache is refilled beginning in cycle 6.

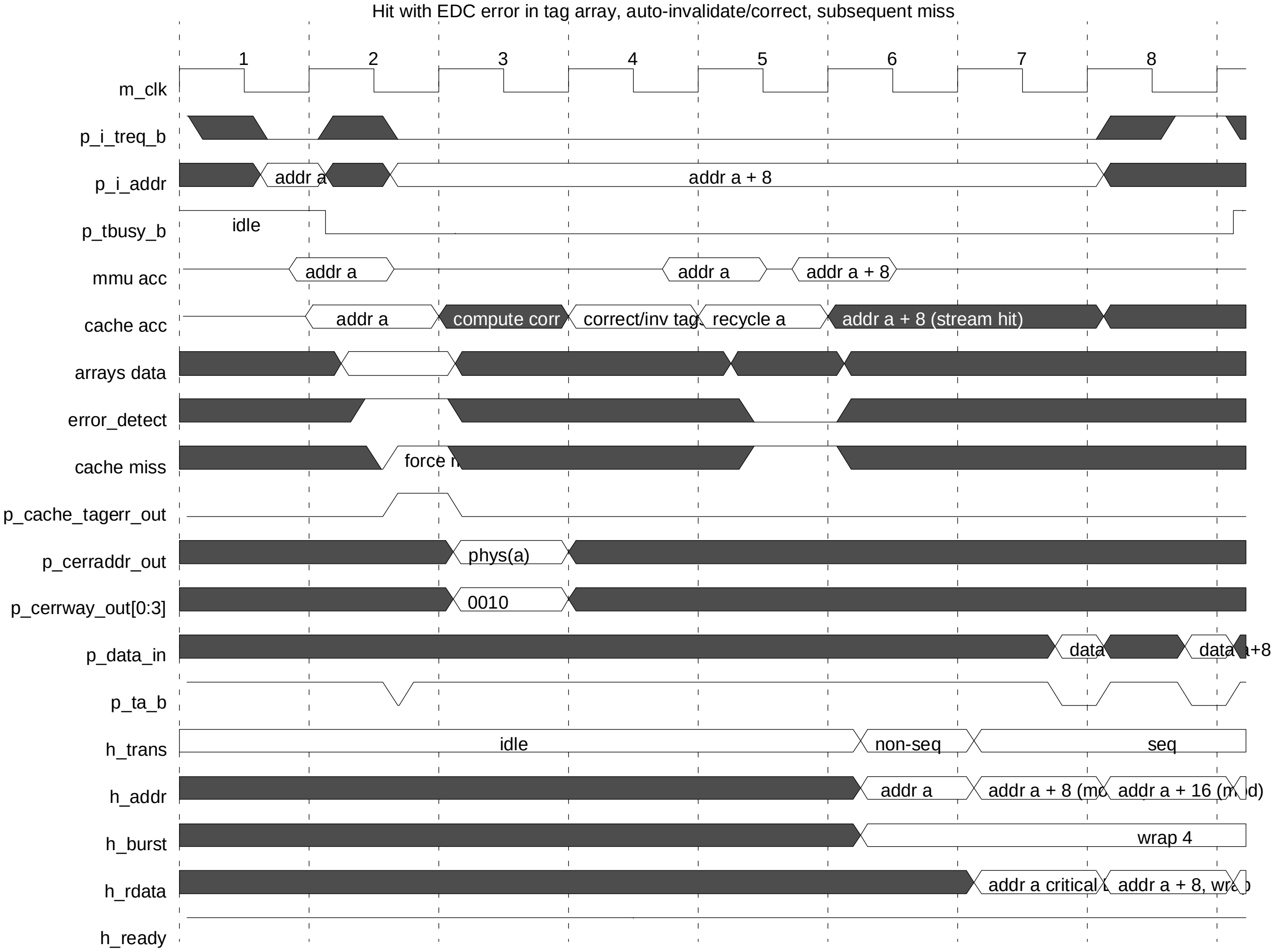


Figure 13-28. Cross-Signaling Invalidation Output Operation—Tag Error, Miss

Figure 13-29 illustrates functional timing for a cross-signaling operation by a CPU encountering an internal cache error in the cache tag array with the error action indicating that an auto-invalidation should be generated. A cache tag array error is detected in cycle 2 and results in a cache correction/invalidation cycle being forced. In this example, way 1 has a correctable error. The correction of way 1 is performed in cycle 4. The address 'a' access is recycled in cycle 5 and results in a hit.

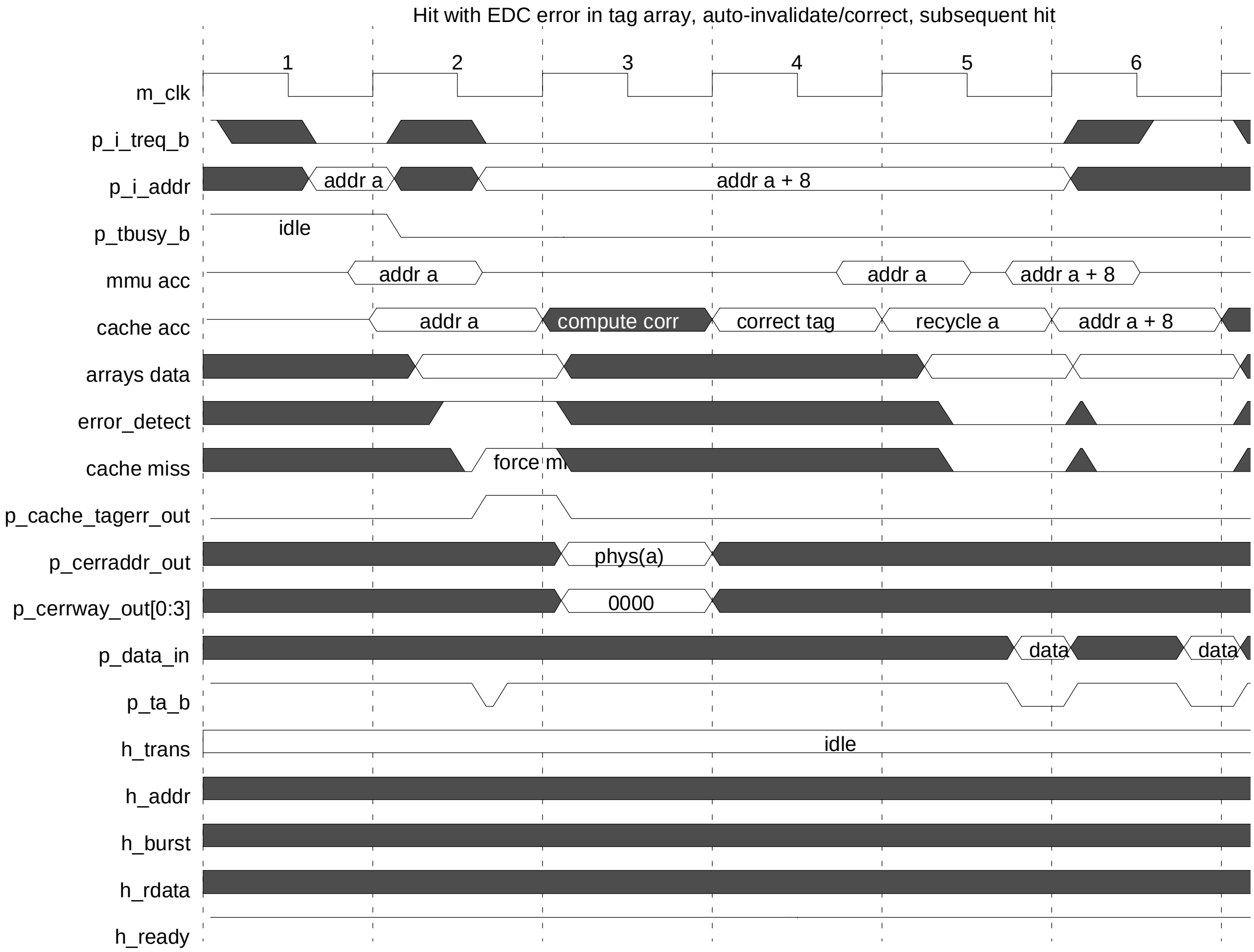


Figure 13-29. Cross-signaling Invalidation Output Operation—Tag Error, Hit

Figure 13-30 illustrates functional timing for a cross-signaling operation by a CPU which encounters an internal cache error in the cache tag array with the error action indicating that an auto-invalidation should be generated. A cache tag array error is detected in cycle 2 and 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.

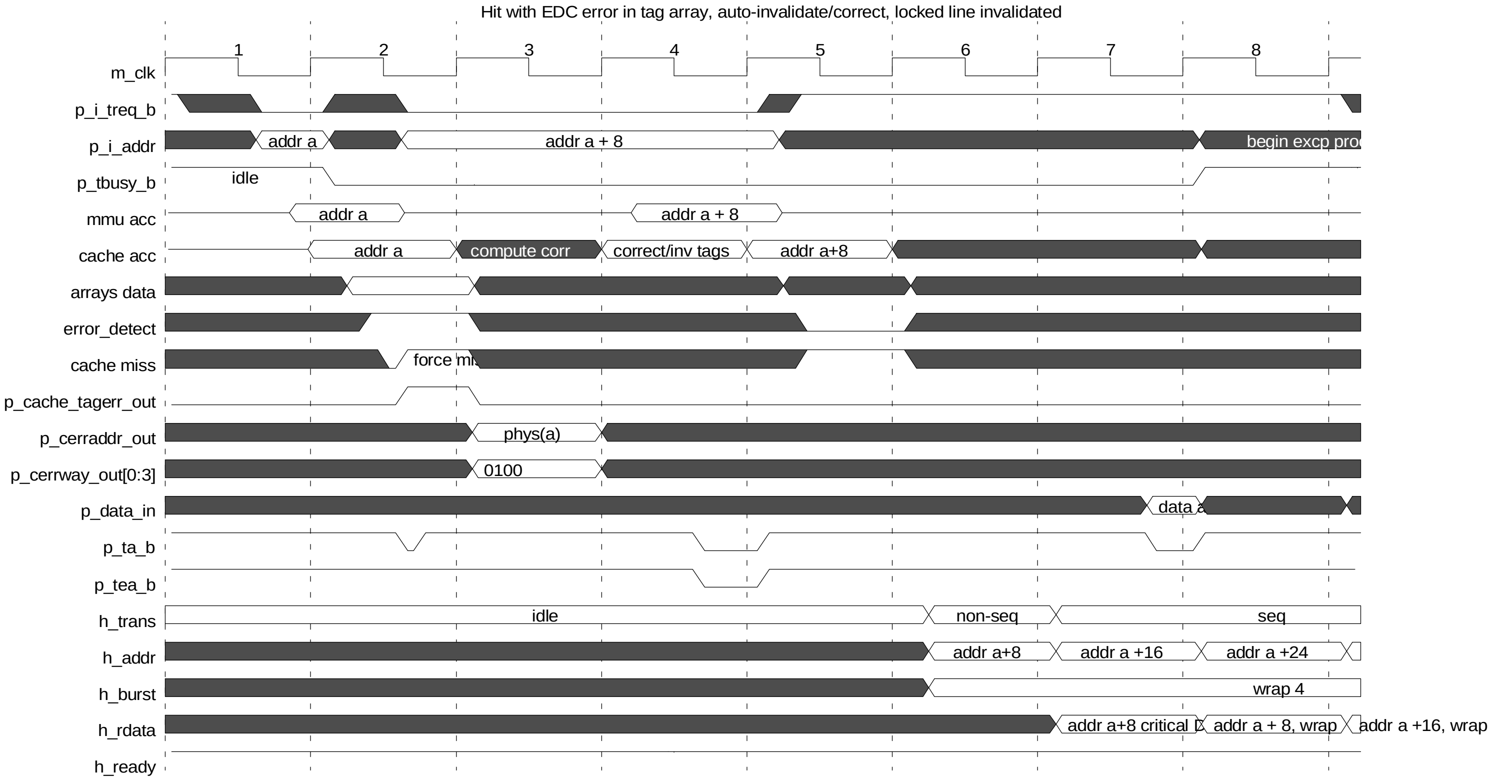

Hit with EDC error in tag array, auto-invalidate/correct, locked line invalidated

Figure 13-30. Cross-Signaling Invalidation Output Operation—Tag Error, Locked Inv

Figure 13-31 illustrates functional timing for a cross-signaling operation by a CPU receiving a cache error cross-signaling operation for the cache data array with the error action indicating that an auto-invalidation should be generated. A cache data array error is detected in cycle 2 by an external cache, and results in a cache miss being forced. The error entry is refilled beginning in cycle 3.

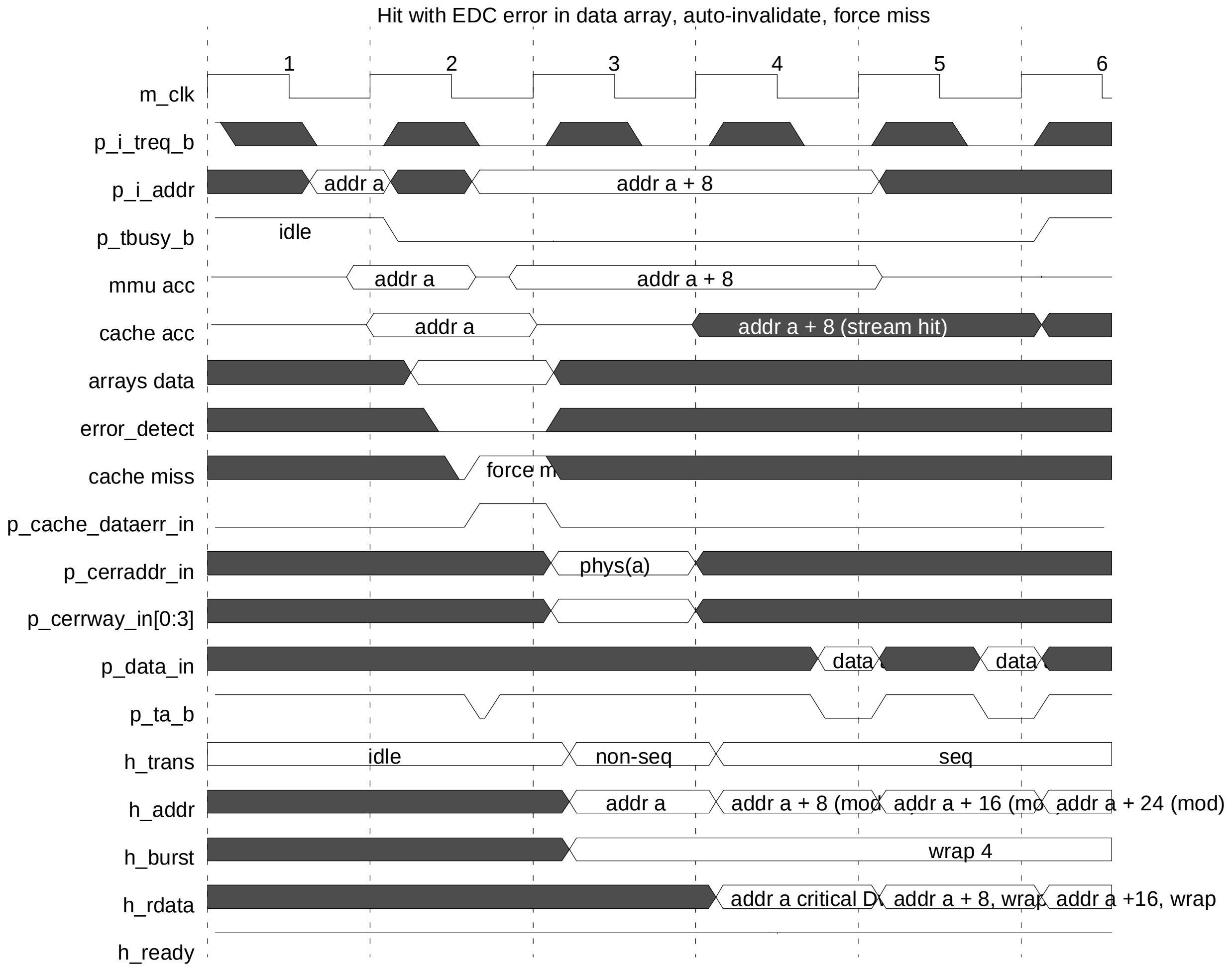


Figure 13-31. Cross-Signaling Invalidation Input Operation—Data Error

Figure 13-32 illustrates functional timing for a cross-signaling operation by a CPU receiving a cache error cross-signaling operation for the cache tag array with the error action indicating that an auto-invalidation should be generated. A cache tag array error is detected in cycle 2 by an external cache, and results in a cache correction/invalidation cycle being forced. In this example, in the external cache, way 0 has a correctable error, and way 2 has an uncorrectable error and requires invalidation. Only the invalidations are signaled on the **p_cerrway_in[0:3]** inputs. The invalidation of way 2 is performed in cycle 4. The address ‘a’ access is recycled in cycle 5 and results in a miss. The cache is refilled beginning in cycle 6.

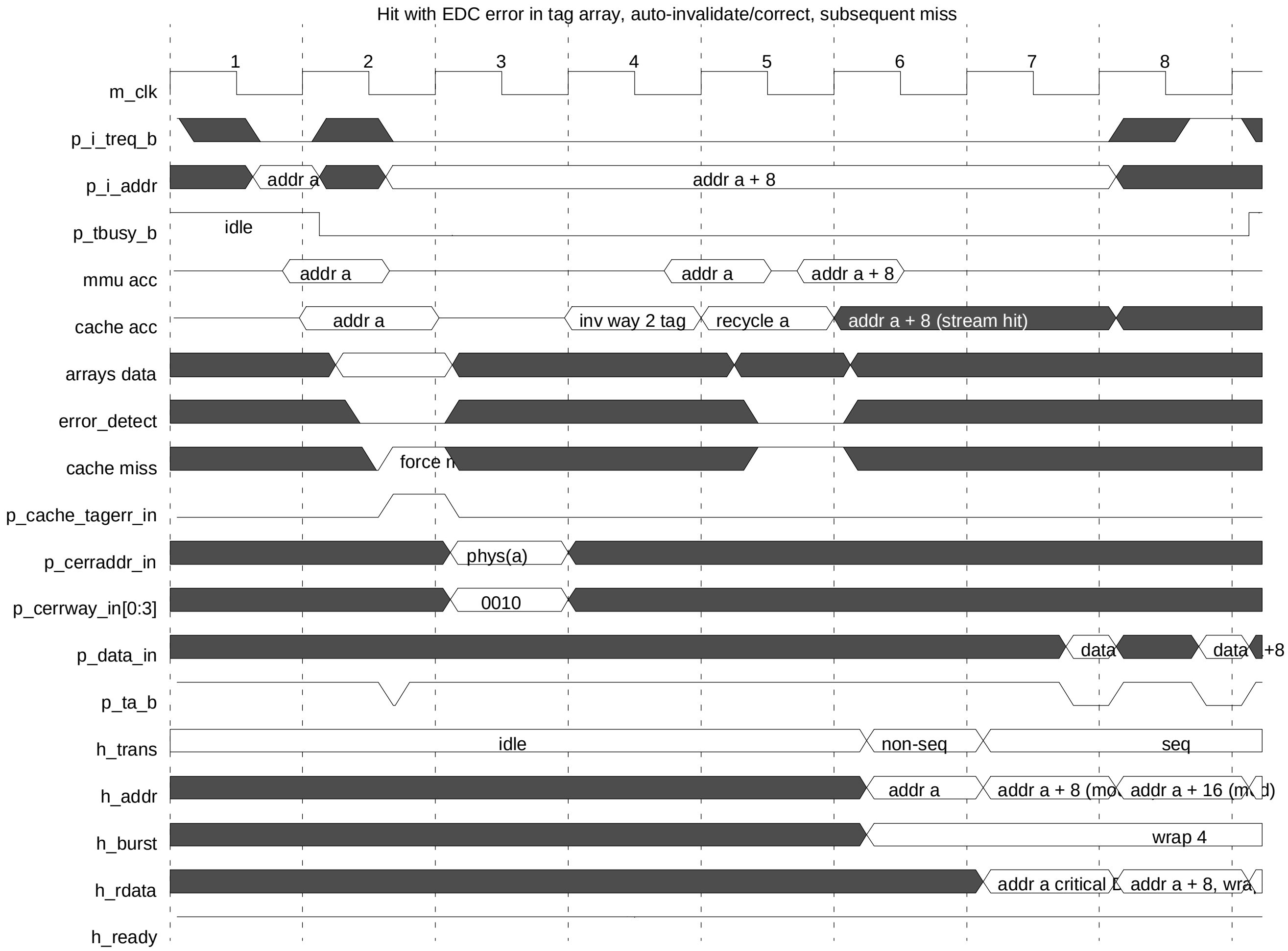


Figure 13-32. Cross-Signaling Invalidation Input Operation—Tag Error, Miss

Figure 13-33 illustrates functional timing for a cross-signaling operation by a CPU which receives a cache error cross-signaling operation for the cache tag array with the error action indicating that an auto-invalidation should be generated. A cache tag array error is detected in cycle 2 by an external cache, and results in a cache correction/invalidation cycle being forced in cycle 3. In this example, in the external cache, way 1 has a correctable error. Only the invalidations are signaled on the **p_cerrway_in[0:3]** inputs.

Since no invalidations are required, no access is performed in cycle 4. The address ‘a’ access is recycled in cycle 5 and results in a hit.

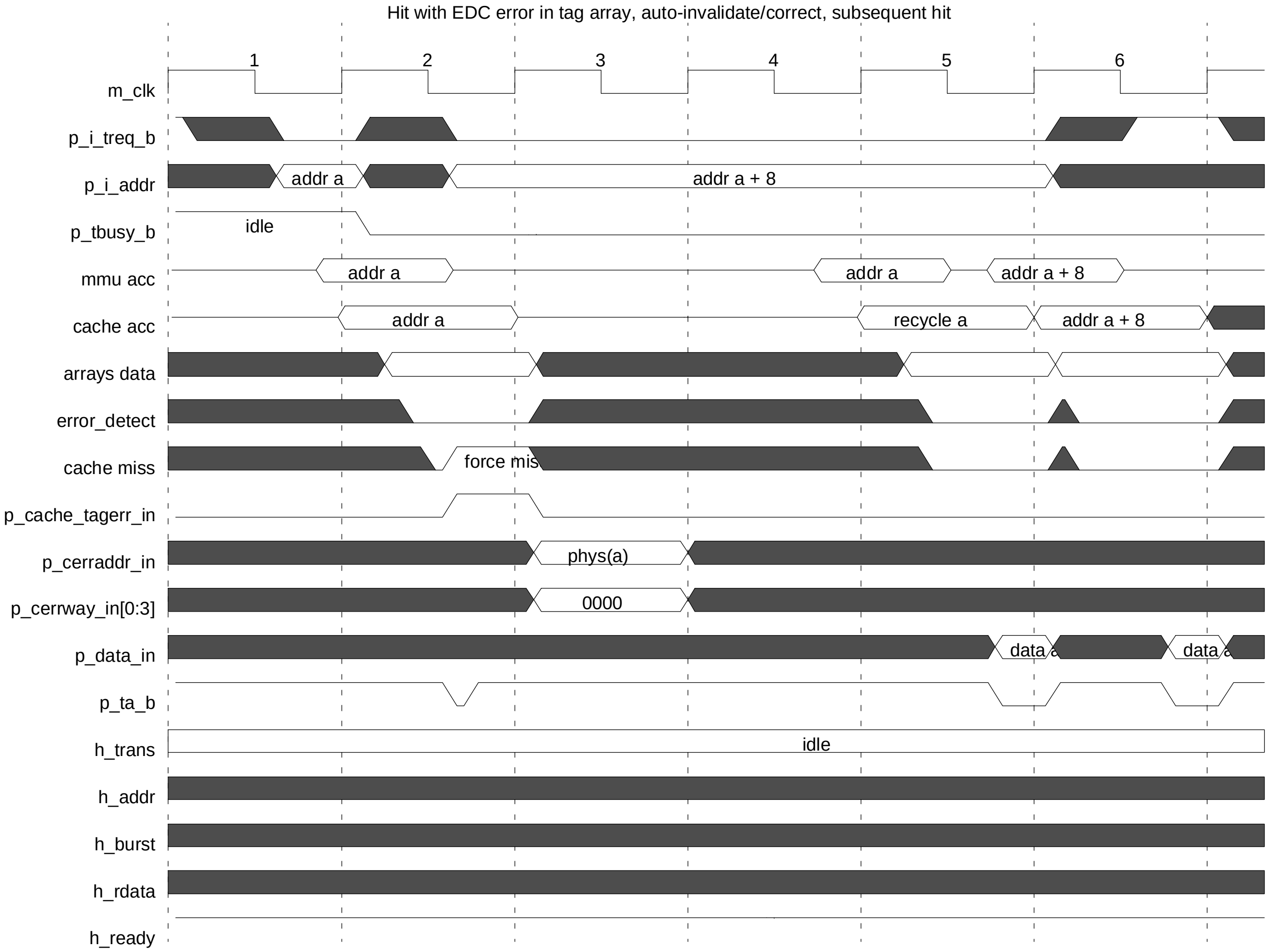


Figure 13-33. Cross-Signaling Invalidation Input Operation—Tag Error, Hit

Figure 13-34 illustrates functional timing for a cross-signaling operation by a CPU encountering an internal cache error in the cache tag array with the error action indicating that an auto-invalidation should be generated. A cache tag array error is detected in cycle 2 and 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.

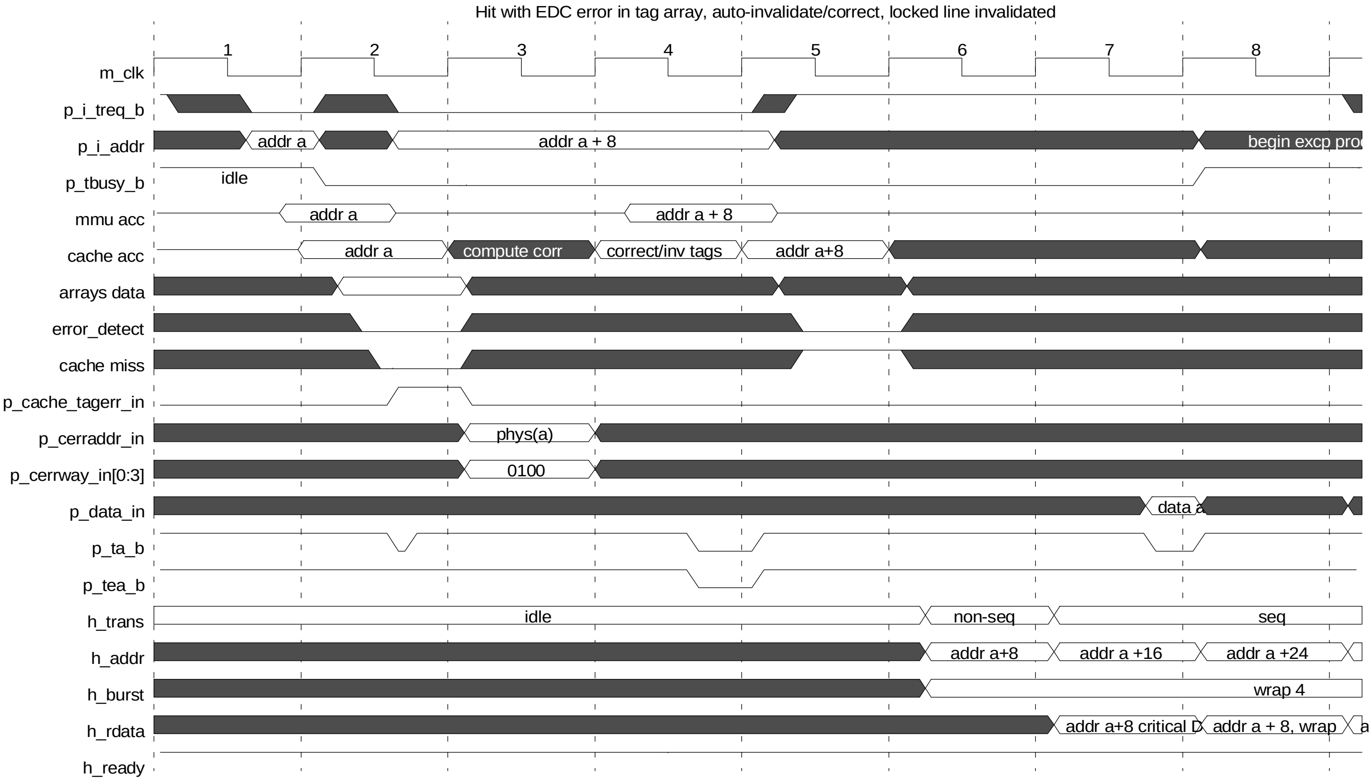


Figure 13-34. Cross-signaling Invalidation Input Operation—Tag Error, Locked Inv

13.4.4 Debug Lockstep Cross-signaling Operation

The debug lockstep cross-signaling interface allows the lockstep operation of two or more CPUs during external debug mode (EDM) operations in which the CPUs must maintain lockstep operation in the presence of asynchronous debug operations causing the CPU to enter or exit a debug halted mode. The interface provides a means for 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 by 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. Since 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 the means for ensuring lockstep updates of register resources.

13.4.4.1 Debug Entry Cross-Signaling

Figure 13-35 illustrates functional timing for debug entry cross-signaling operation with lockstep operation disabled.

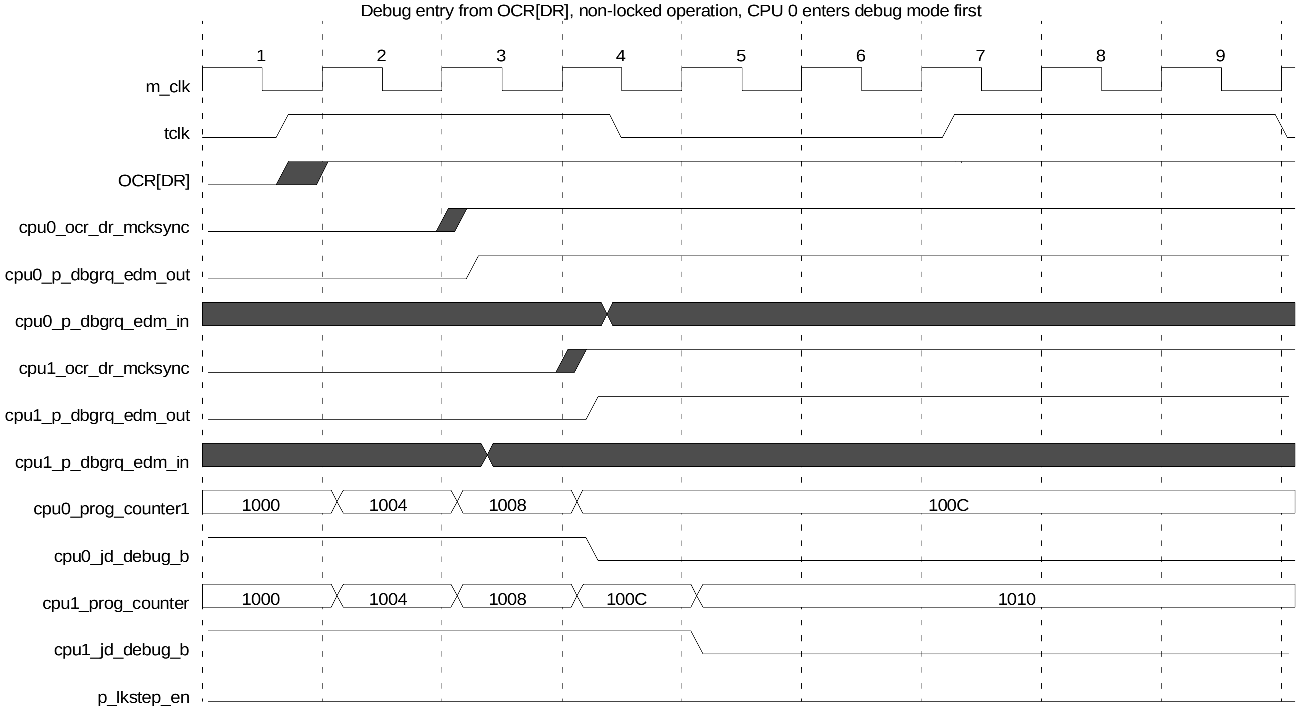


Figure 13-35. Debug Entry Cross-Signaling Interface, Non-Lockstep Mode

In this example, entry into debug mode is requested by setting the OCR_{DR} bit 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 shown 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 13-36 illustrates functional timing for debug entry cross-signaling operation with lockstep operation enabled.

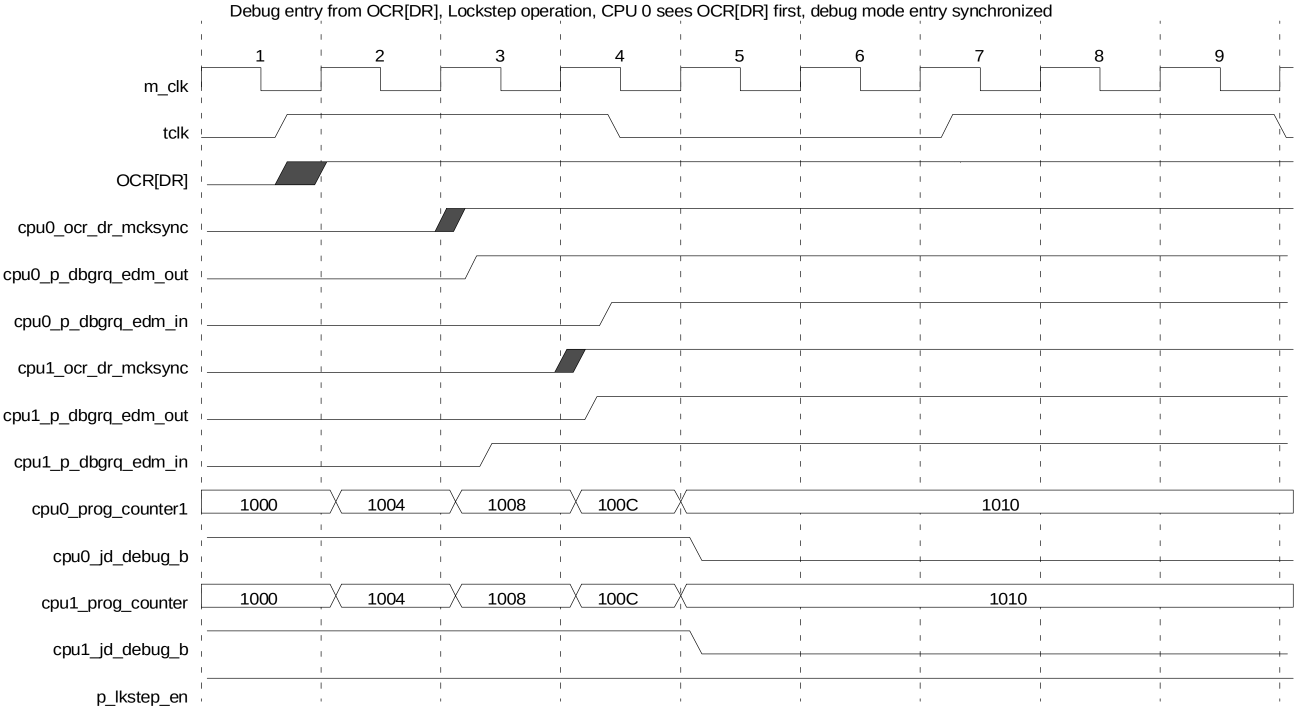


Figure 13-36. Debug Entry Cross-Signaling Interface, Lockstep Mode

In this example, entry into debug mode is requested by setting the OCR_{DR} bit 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 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. 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 13-37 illustrates functional timing for debug entry cross-signaling operation with lockstep operation enabled.

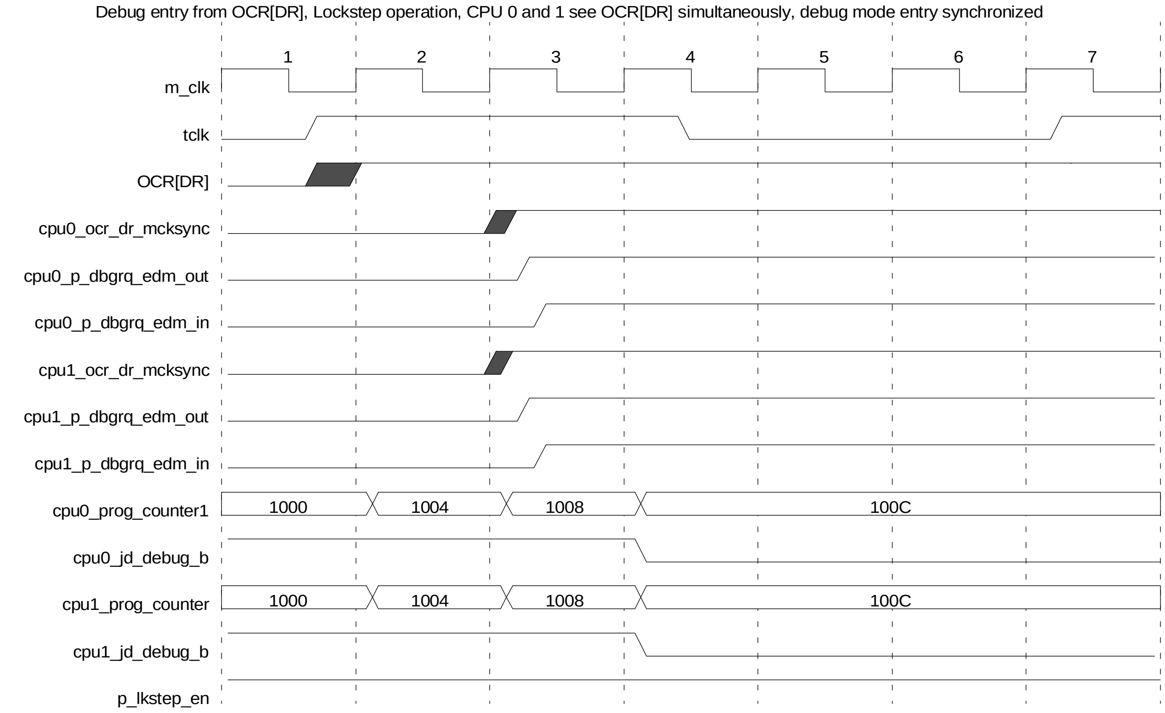


Figure 13-37. Debug Entry Cross-Signaling Interface, Lockstep Mode (2)

In this example, entry into debug mode is requested by setting the OCR$_{DR}$ bit 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.

13.4.4.2 Debug Exit Cross-Signaling

Figure 13-38 illustrates functional timing for debug exit cross-signaling operation with lockstep operation disabled.

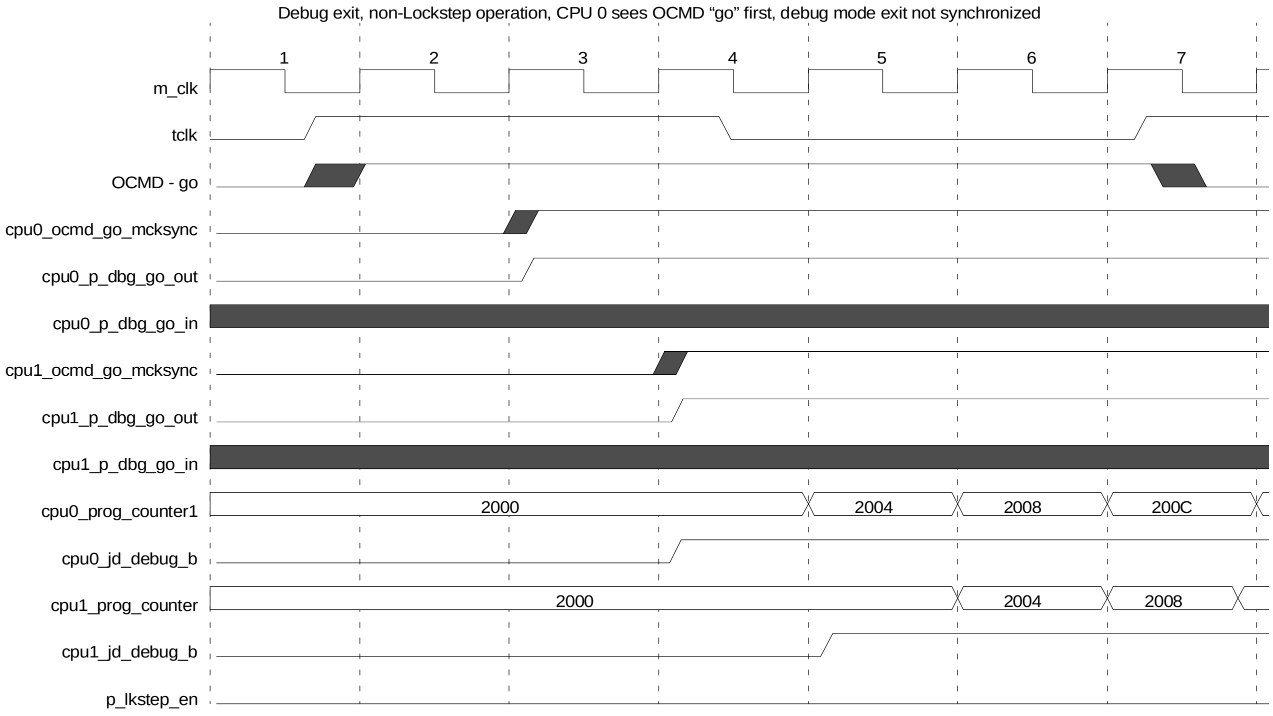


Figure 13-38. Debug Exit Cross-Signaling Interface, Non-Lockstep mode

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. Because 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 not asserted for this example, 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 13-39 illustrates functional timing for debug exit cross-signaling operation with lockstep operation enabled.

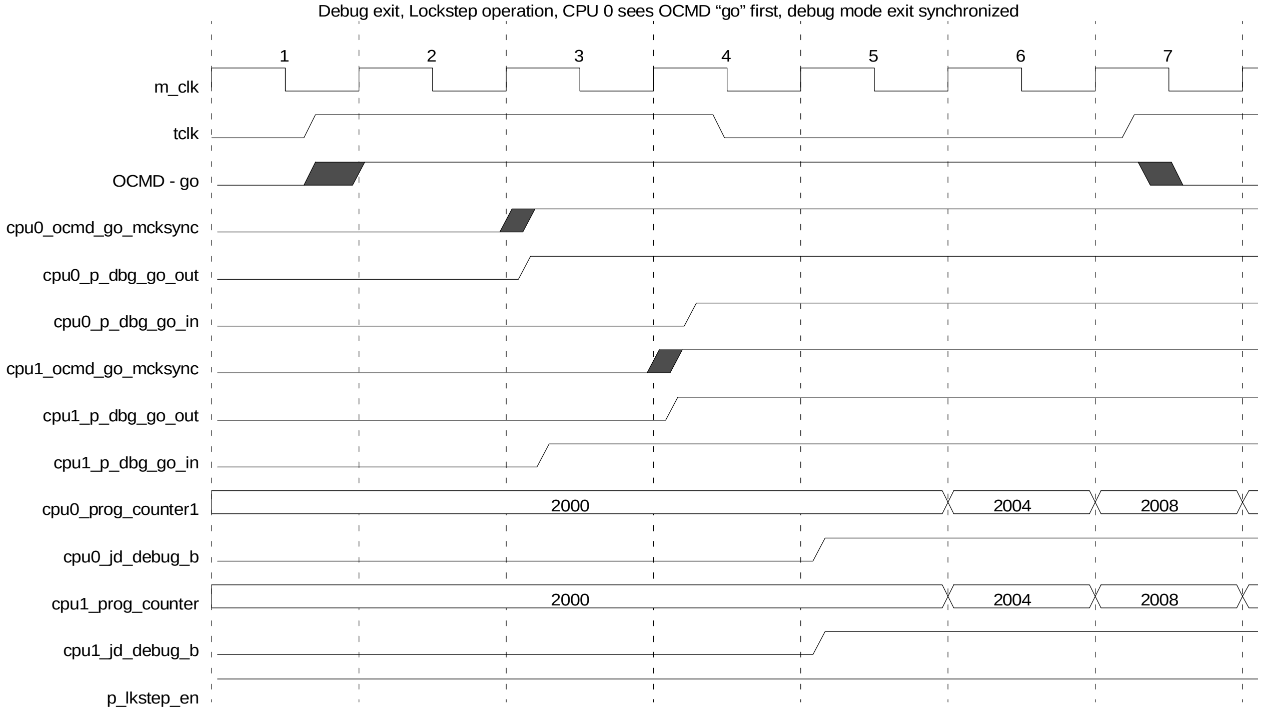


Figure 13-39. Debug Exit Cross-Signaling Interface, Lockstep Mode

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.

Since 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 13-40 illustrates functional timing for debug exit cross-signaling operation with lockstep operation enabled.

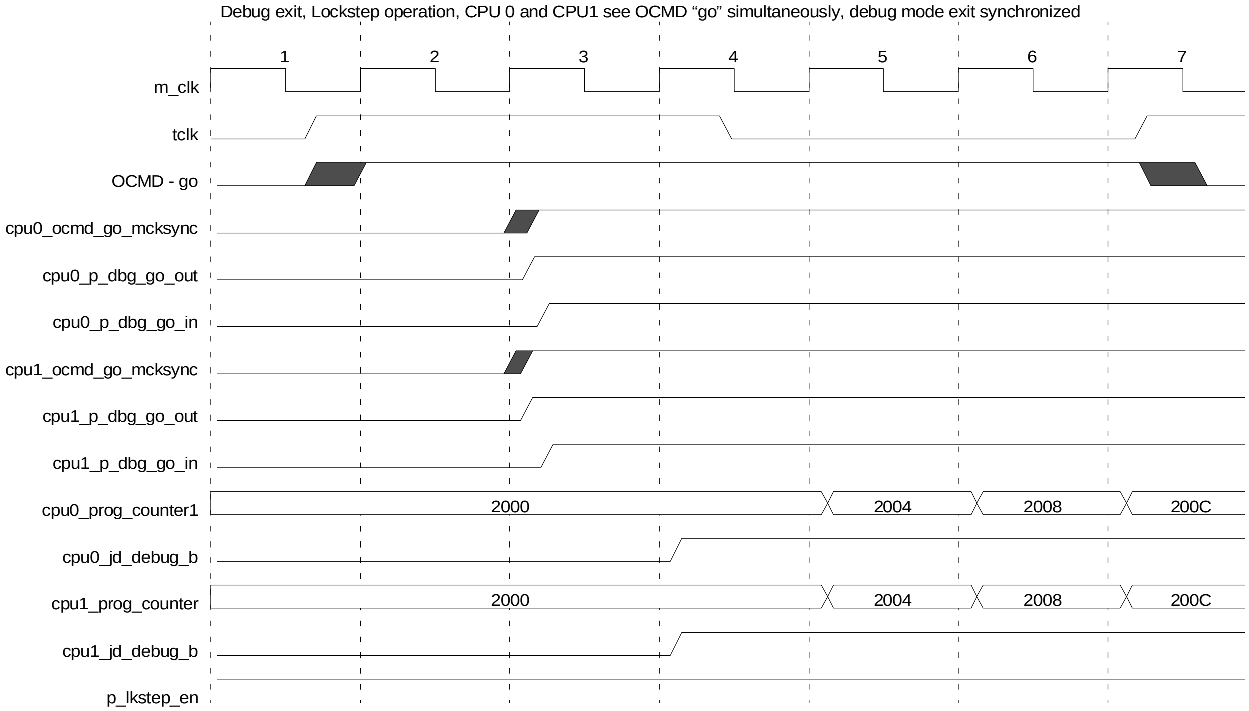


Figure 13-40. Debug Exit Cross-Signaling Interface, Lockstep mode (2)

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. 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 and 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_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.

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 4 (**jd_debug_b** negated) and begin execution. The two CPUs are thus kept in sync.

13.4.4.3 Update_DR State Cross-Signaling

Figure 13-41 illustrates functional timing for Update_DR cross-signaling operation with lockstep operation enabled.

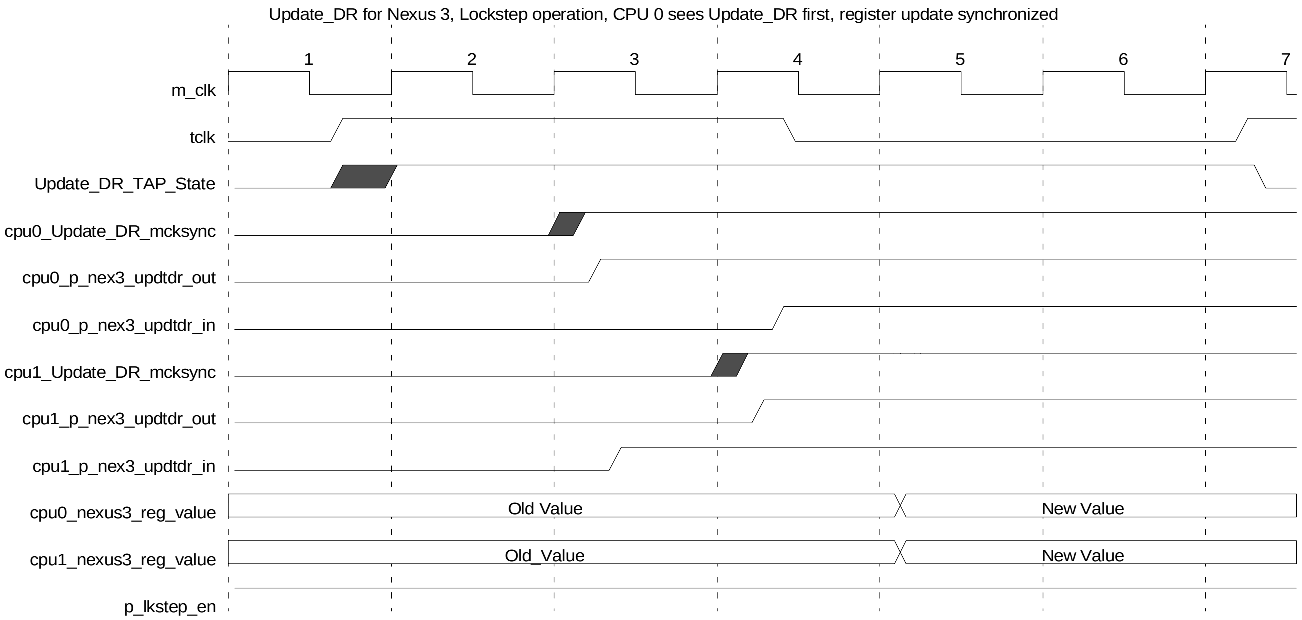


Figure 13-41. Debug Update_DR State Cross-Signaling Interface, Lockstep mode

In this example, an update to one of the Nexus 3 registers 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. In the example in the timing diagram, 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. Since the lockstep control signal **p_lkstep_en** is asserted for this example, 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 thus properly in sync.

Figure 13-42 illustrates functional timing for Update_DR cross-signaling operation with lockstep operation enabled.

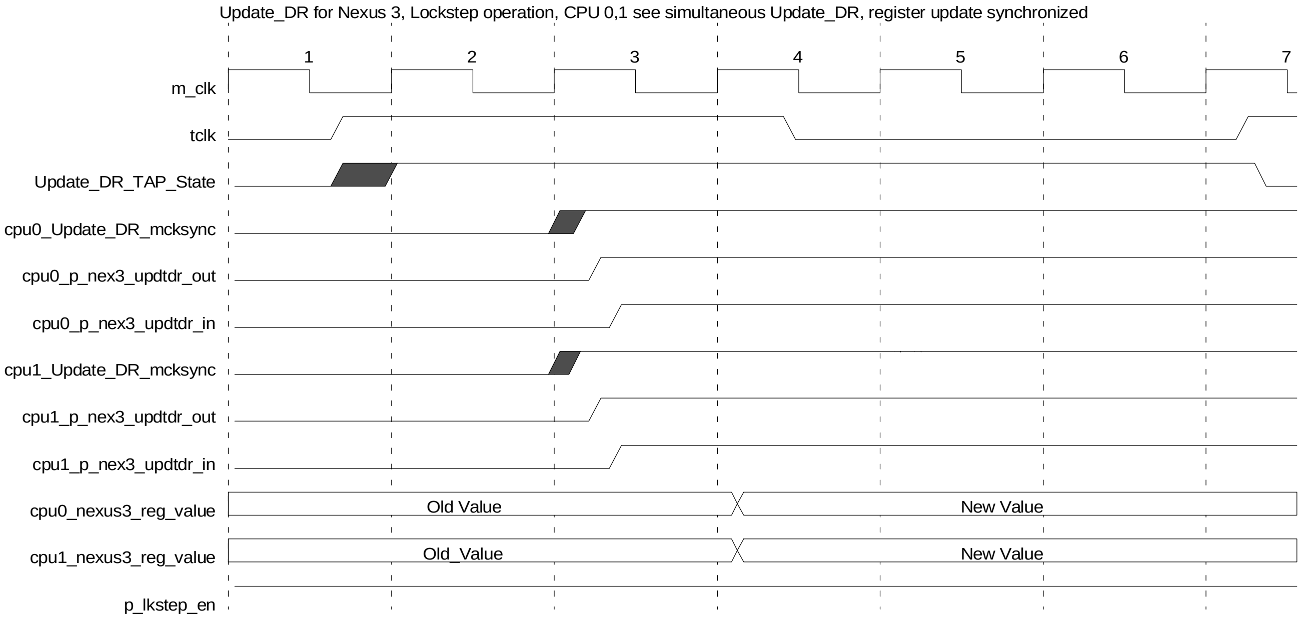


Figure 13-42. Debug Update_DR State Cross-Signaling Interface, Lockstep Mode (2)

In this example, an update to one of the Nexus 3 registers 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. In the example in the timing diagram, 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 for this example, 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.

13.4.5 Power Management

The following diagram shows the relationship of the wakeup control signal, **p_wakeup,** to the relevant input signals.

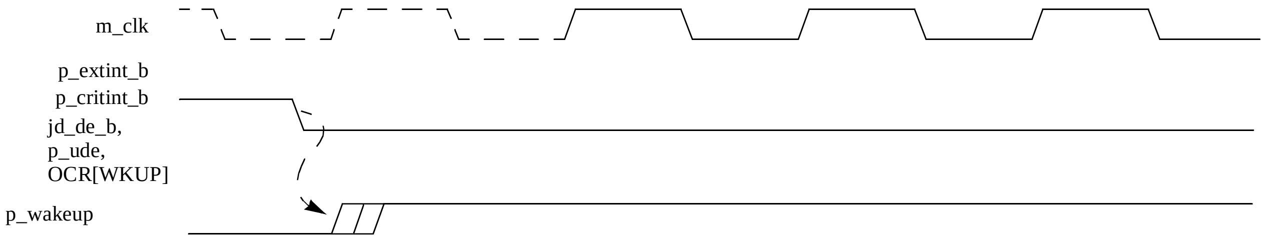


Figure 13-43. Wakeup Control Signal (p_wakeup)

13.4.6 Interrupt Interface

Figure 13-44 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.

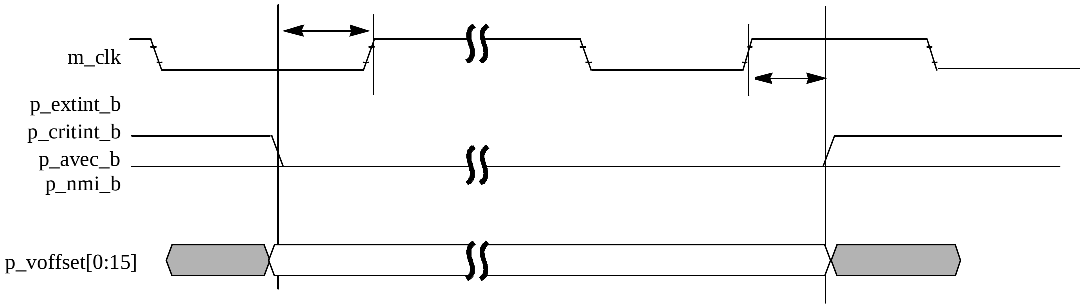


Figure 13-44. Interrupt Interface Input Signals

Figure 13-45 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.

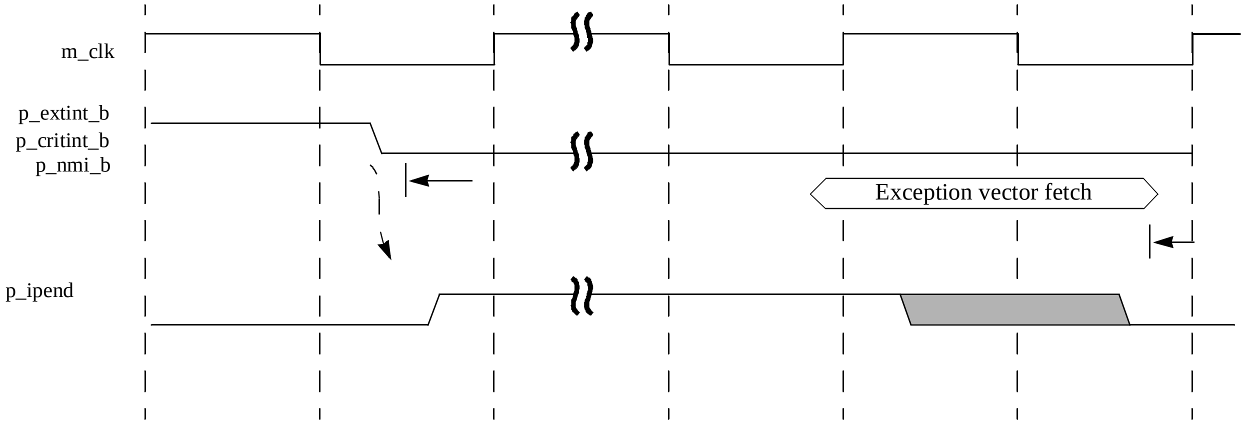


Figure 13-45. Interrupt Pending operation

Figure 13-46 shows the relationship of the interrupt acknowledge signal to the interrupt request inputs and exception vector fetching.

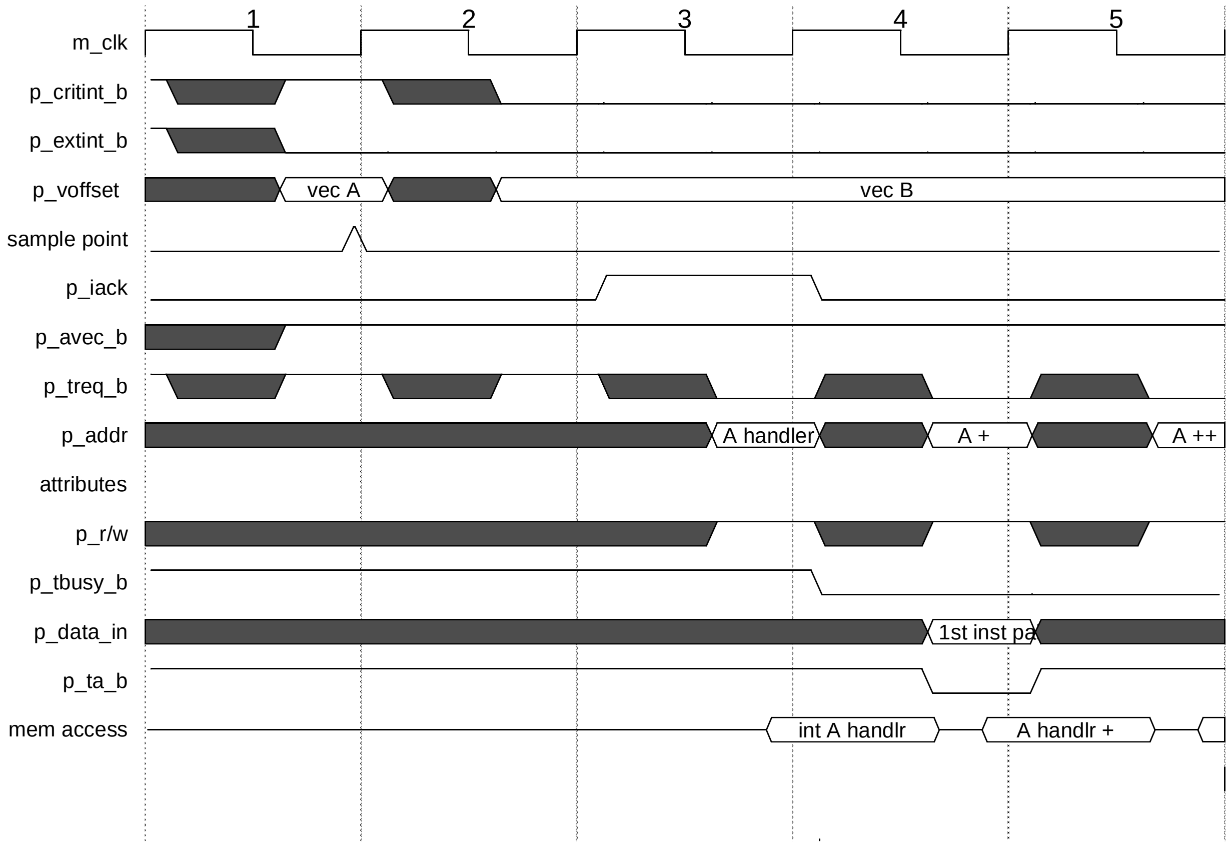


Figure 13-46. Interrupt acknowledge operation

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. For this example, 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 autovector 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/autovector 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 is indicating that the values present at the rise of cycle 2 (vector ‘A’) have been committed to. During cycle 3, 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 is 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 the cycle to which an interrupt is committed. In the following example, since the CPU was unable to acknowledge the external input interrupt during cycle 2 due to internal or external execution conditions, the critical input request was sampled. This case is shown in Figure 13-47.

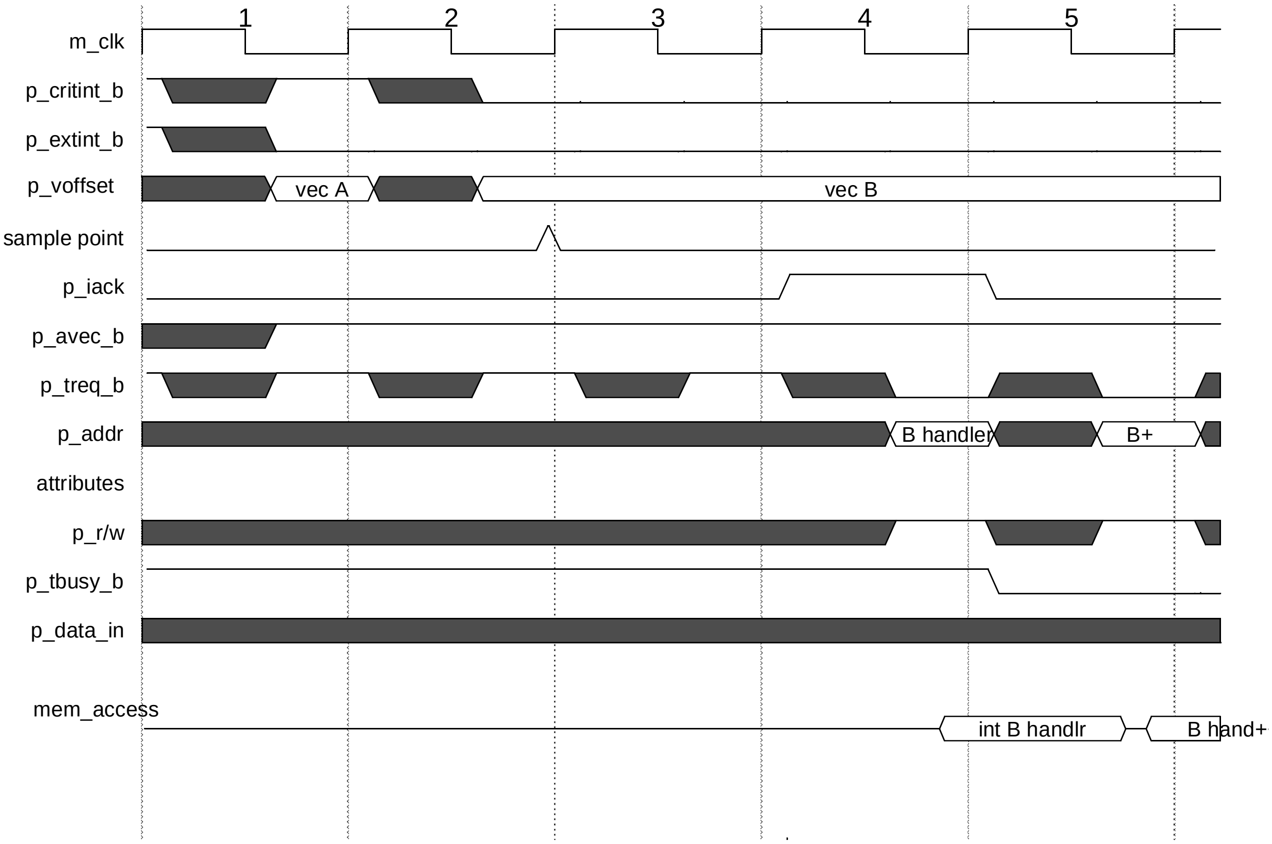


Figure 13-47. Interrupt Acknowledge Operation—2

13.4.7 Time Base Interface

The following figure shows the required relationships of the time base inputs. The electrical values associated with these timings may be found in the *e200 Integration Guide*.

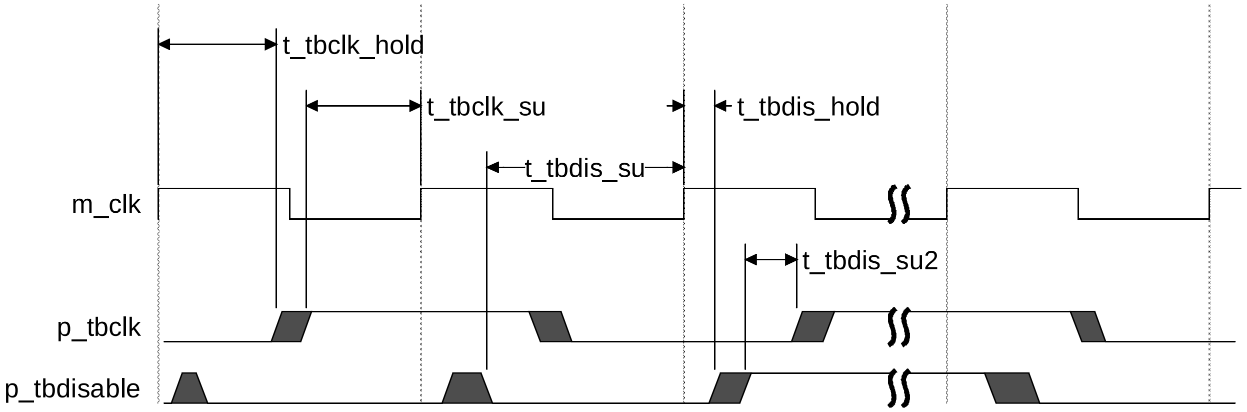


Figure 13-48. Time Base Input Timing

13.4.8 JTAG Test Interface

Figure 13-49, Figure 13-50, and Figure 13-51 show the relationships of the various JTAG related signals to the **j_tclk** input. The electrical values associated with these timings may be found in the *e200 Integration Guide*.

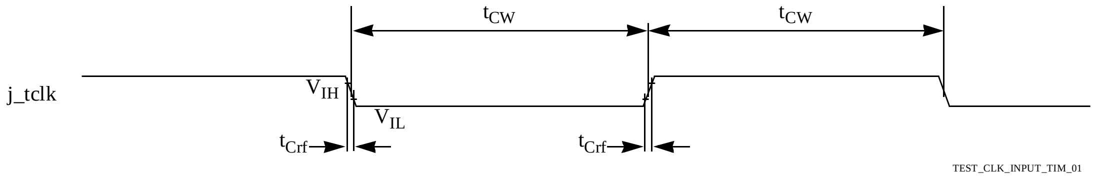


Figure 13-49. Test Clock Input Timing

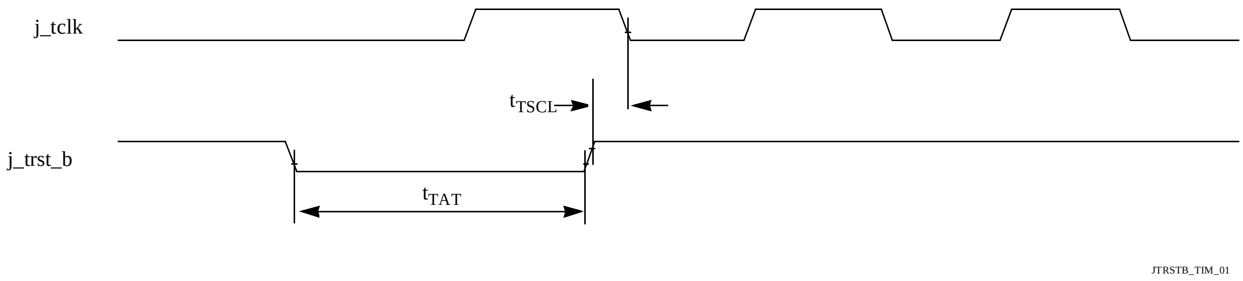


Figure 13-50. j_trst_b Timing

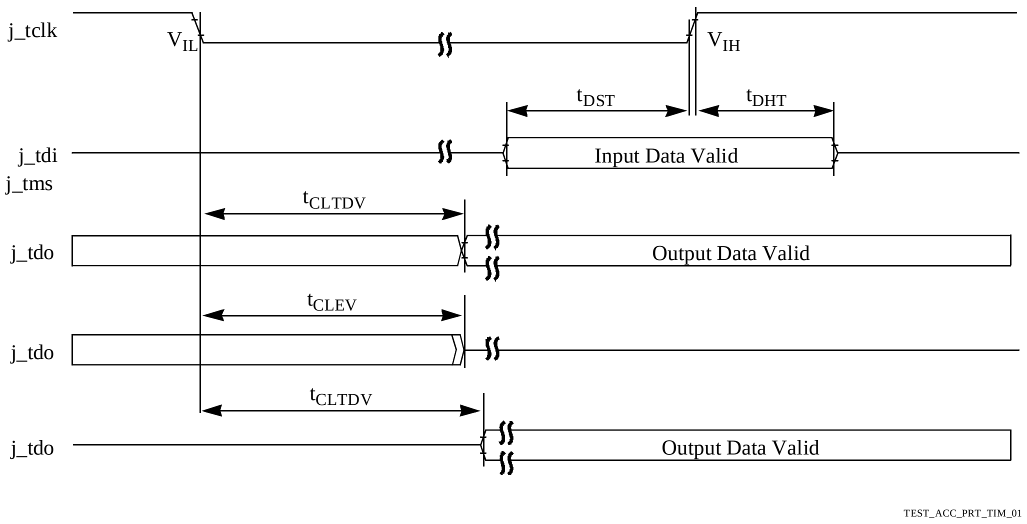


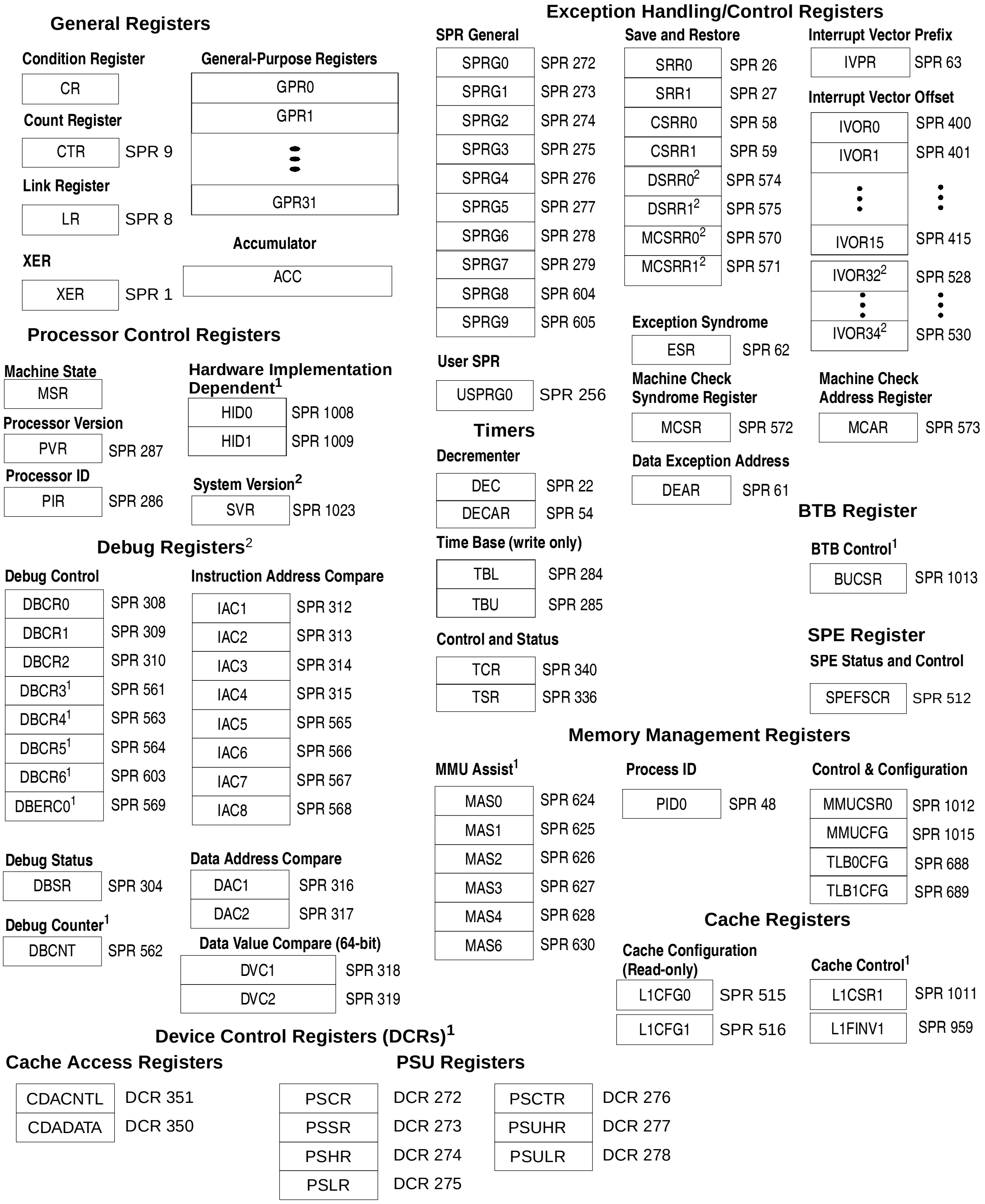
Figure 13-51. Test Access Port Timing

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 Freescale 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 Freescale EIS; in others it is left up to the processor. This chapter identifies the level at which each features is defined.

Figure A-1 and Figure A-2 show the e200 register set, grouped by whether they can be accessed by user- or supervisor-level software. Unless otherwise indicated, these registers are defined by the base or embedded category of the Power ISA architecture.



General Registers

Condition Register

CR	

Count Register

CTR	SPR 9

Link Register

LR	SPR 8

XER

XER	SPR 1

General-Purpose Registers

GPR0
GPR1
⋮
GPR31

Accumulator

ACC

Processor Control Registers

Machine State

MSR	

Processor Version

PVR	SPR 287

Processor ID

PIR	SPR 286

Hardware Implementation Dependent[1]

HID0	SPR 1008
HID1	SPR 1009

System Version[2]

SVR	SPR 1023

Debug Registers[2]

Debug Control

DBCR0	SPR 308
DBCR1	SPR 309
DBCR2	SPR 310
DBCR3[1]	SPR 561
DBCR4[1]	SPR 563
DBCR5[1]	SPR 564
DBCR6[1]	SPR 603
DBERC0[1]	SPR 569

Instruction Address Compare

IAC1	SPR 312
IAC2	SPR 313
IAC3	SPR 314
IAC4	SPR 315
IAC5	SPR 565
IAC6	SPR 566
IAC7	SPR 567
IAC8	SPR 568

Debug Status

DBSR	SPR 304

Debug Counter[1]

DBCNT	SPR 562

Data Address Compare

DAC1	SPR 316
DAC2	SPR 317

Data Value Compare (64-bit)

DVC1	SPR 318
DVC2	SPR 319

Exception Handling/Control Registers

SPR General

SPRG0	SPR 272
SPRG1	SPR 273
SPRG2	SPR 274
SPRG3	SPR 275
SPRG4	SPR 276
SPRG5	SPR 277
SPRG6	SPR 278
SPRG7	SPR 279
SPRG8	SPR 604
SPRG9	SPR 605

User SPR

USPRG0	SPR 256

Save and Restore

SRR0	SPR 26
SRR1	SPR 27
CSRR0	SPR 58
CSRR1	SPR 59
DSRR0[2]	SPR 574
DSRR1[2]	SPR 575
MCSRR0[2]	SPR 570
MCSRR1[2]	SPR 571

Interrupt Vector Prefix

IVPR	SPR 63

Interrupt Vector Offset

IVOR0	SPR 400
IVOR1	SPR 401
⋮	⋮
IVOR15	SPR 415
IVOR32[2]	SPR 528
⋮	⋮
IVOR34[2]	SPR 530

Exception Syndrome

ESR	SPR 62

Machine Check Syndrome Register

MCSR	SPR 572

Machine Check Address Register

MCAR	SPR 573

Data Exception Address

DEAR	SPR 61

Timers

Decrementer

DEC	SPR 22
DECAR	SPR 54

Time Base (write only)

TBL	SPR 284
TBU	SPR 285

Control and Status

TCR	SPR 340
TSR	SPR 336

BTB Register

BTB Control[1]

BUCSR	SPR 1013

SPE Register

SPE Status and Control

SPEFSCR	SPR 512

Memory Management Registers

MMU Assist[1]

MAS0	SPR 624
MAS1	SPR 625
MAS2	SPR 626
MAS3	SPR 627
MAS4	SPR 628
MAS6	SPR 630

Process ID

PID0	SPR 48

Control & Configuration

MMUCSR0	SPR 1012
MMUCFG	SPR 1015
TLB0CFG	SPR 688
TLB1CFG	SPR 689

Cache Registers

Cache Configuration (Read-only)

L1CFG0	SPR 515
L1CFG1	SPR 516

Cache Control[1]

L1CSR1	SPR 1011
L1FINV1	SPR 959

Device Control Registers (DCRs)[1]

Cache Access Registers

CDACNTL	DCR 351
CDADATA	DCR 350

PSU Registers

PSCR	DCR 272	PSCTR	DCR 276
PSSR	DCR 273	PSUHR	DCR 277
PSHR	DCR 274	PSULR	DCR 278
PSLR	DCR 275		

1 - These e200-specific registers may not be supported by other processors built on Power Architecture technology
2 - Optional registers defined by the Power ISA embedded architecture
3 - Read-only registers

Figure A-1. e200z446n3 Supervisor Mode Programmer's Model SPRs

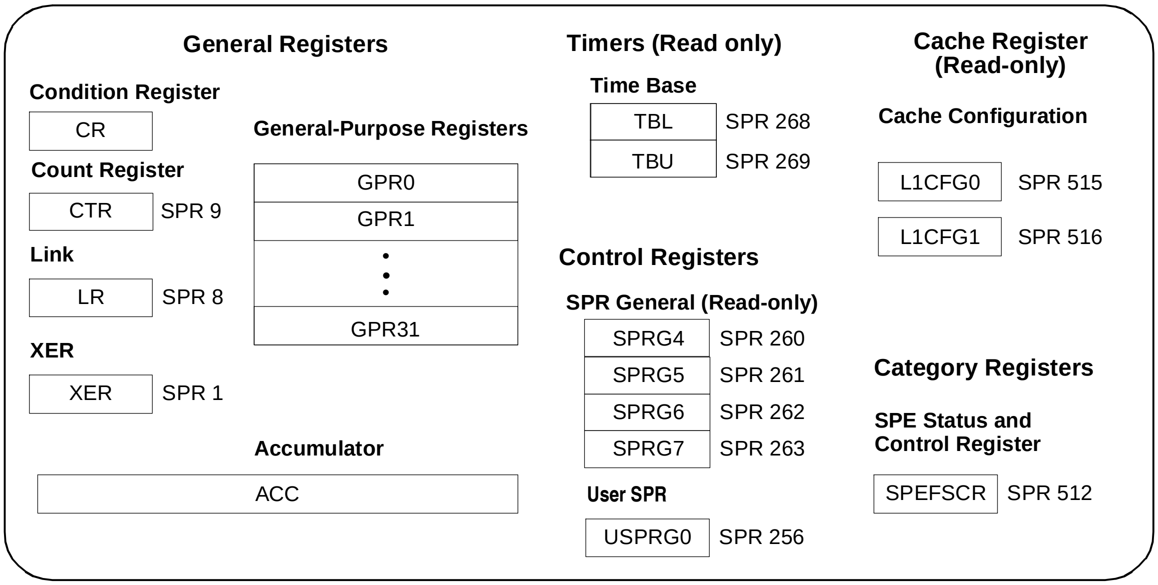


Figure A-2. e200z4463 User Mode Programmer's Model SPRs

Figure A-3–Figure A-47 show the individual registers.

0					UCLE	SPE	0						WE	CE	0	EE	PR	FP	ME	FE0	0	DE	FE1	0		IS	DS	0		RI	0
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

Figure A-3. Machine State Register (MSR)

ID																															
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

Figure A-4. Processor ID Register (PIR)

1	0	0	0	0	0	0	1	0	1	0	1	Version				MBG Reserved				Minor Rev				Major Rev				MBG ID			
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

Figure A-5. Processor Version Register (PVR)

System Version																															
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

Figure A-6. System Version Register (SVR)

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
SO	OV	CA	0																						Bytecnt						

Figure A-7. Integer Exception Register (XER)

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				PIL	PPR	PTR	FP	ST	0	DLK	ILK	AP	PUO	BO	PIE	0								SPE	0	VLEMI	0			MIF	0

Figure A-8. Exception Syndrome Register (ESR)

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
MCP	IC_DPERR	0		EXCP_ERR	IC_TPERR	0	IC_LKERR	0			NMI	MAV	MEA	0	IF	LD	ST	G	0								BUS_IRERR	BUS_DRERR	BUS_WRERR	0	

Figure A-9. Machine Check Syndrome Register (MCSR)

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
WP		WRC		WIE	DIE	FP		FIE	ARE	0	WPEXT				FPEXT				0												

Figure A-10. Timer Control Register (TCR)

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
ENW	WIS	WRS		DIS	FIS	0																									

Figure A-11. Timer Status Register (TSR)

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
EMCP	0							DOZE	NAP	SLEEP	0			ICR	NHR	0	TBEN	SELTBCLK	DCLREE	DCLRCE	CICLRDE	MCCLRDE	DAPUEN	0							NOPTI

Figure A-12. Hardware Implementation Dependent Register 0 (HID0)

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																SYSCTL								ATS	0						

Figure A-13. Hardware Implementation Dependent Register 1 (HID1)

0–21	22	23–25	26–27	28	29–30	31
0	BBFI	0	BALLOC	0	BPRED	BPEN

Figure A-14. Branch Unit Control and Status Register (BUCSR)

0–15	16–27	28–31
0	Vector Offset	0

Figure A-15. e200 Interrupt Vector Offset Register (IVOR)

0–16	17–31
CNT1	CNT2

Figure A-16. DBCNT Register

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	27–30	31
EDM	IDM	RST	ICMP	BRT	IRPT	TRAP	IAC1	IAC2	IAC3	IAC4	DAC1	DAC2	RET	IAC5	IAC6	IAC7	IAC8	DEVT1	DEVT2	DCNT1	DCNT2	CIRPT	CRET	0	FT

Figure A-17. DBCR0 Register

0–1	2–3	4–5	6–7	8–9	10–15	16–17	18–19	20–21	22–23	24–25	26–31
IAC1US	IAC1ER	IAC2US	IAC2ER	IAC12M	0	IAC3US	IAC3ER	IAC4US	IAC4ER	IAC34M	0

Figure A-18. DBCR1 Register

0–1	2–3	4–5	6–7	8–9	10	11	12–13	14–15	16–23	24–31
DAC1US	DAC1ER	DAC2US	DAC2ER	DAC12M	DAC1LNK	DAC2LNK	DVC1M	DVC2M	DVC1BE	DVC2BE

Figure A-19. DBCR2 Register

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
DEVT1C1	DEVT2C1	ICMPC1	IAC1C1	IAC2C1	IAC3C1	IAC4C1	DAC1RC1	DAC1WC1	DAC2RC1	DAC2WC1	IRPTC1	RETC1	DEVT1C2	DEVT2C2	ICMPC2	IAC1C2	IAC2C2	IAC3C2	IAC4C2	DAC1RC2	DAC1WC2	DAC2RC2	DAC2WC2	DEVT1T1	DEVT2T1	IAC1T1	IAC3T1	DAC1RT1	DAC1WT1	CNT2T1	CONFIG

Figure A-20. DBCR3 Register

0	DVC1C	0	DVC2C	0												DAC1XM				DAC2XM				0							
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

Figure A-21. DBCR4 Register

IAC5US		IAC5ER		IAC6US		IAC6ER		IAC56M		0						IAC7US		IAC7ER		IAC8US		IAC8ER		IAC78M		0					
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

Figure A-22. DBCR5 Register

IAC1XM				IAC2XM				IAC3XM				IAC4XM				IAC5XM				IAC6XM				IAC7XM				IAC8XM			
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

Figure A-23. DBCR6 Register

IDE	UDE	MRR		ICMP	BRT	IRPT	TRAP	IAC1	IAC2	IAC3	IAC4-8	DAC1R	DAC1W	DAC2R	DAC2W	RET	0				DEVT1	DEVT2	DCNT1	DCNT2	CIRPT	CRET	VLES	DAC_OFST			CNT1TRG
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

Figure A-24. DBSR Register

0	IDM	RST	UDE	ICMP	BRT	IRPT	TRAP	IAC1	IAC2	IAC3	IAC4	DAC1	0	DAC2	0	RET	IAC5	IAC6	IAC7	IAC8	DEVT1	DEVT2	DCNT1	DCNT2	CIRPT	CRET	BKPT	DQM	0		FT
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

Figure A-25. DBERC0 Register

MCLK	ERR	0	RESET	HALT	STOP	DEBUG	WAIT	0	1
0	1	2	3	4	5	6	7	8	9

Figure A-26. OnCE Status Register

0	1	2	3	4	5	6	7	8	9
R/W	GO	EX	RS[0:6]						

Figure A-27. OnCE Command Register

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								I_DMDIS	0		I_DVLE	I_DI	I_DM	0	I_DE	D_DMDIS	0		D_DW	D_DI	D_DM	D_DG	D_DE	0					WKUP	FDB	DR

Figure A-28. OnCE Control Register

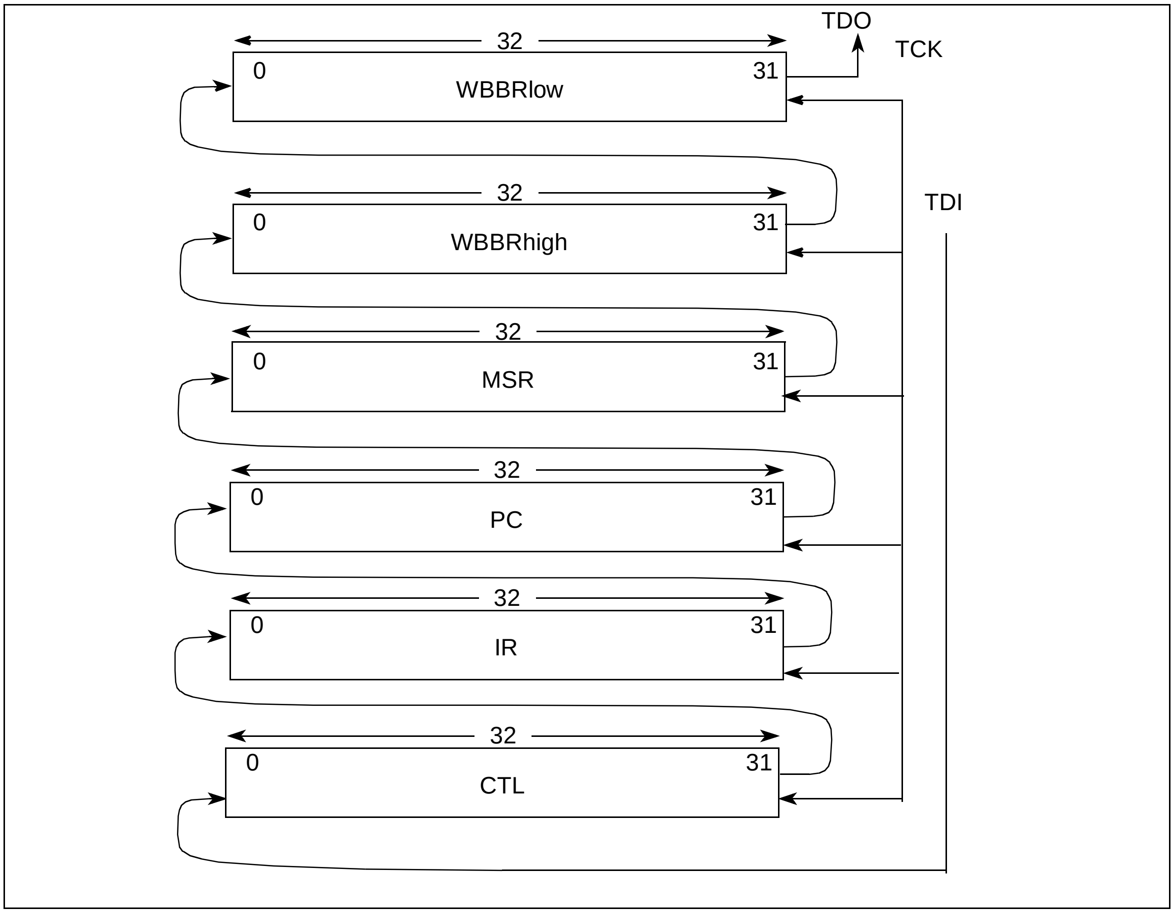


Figure A-29. CPU Scan Chain Register (CPUSCR)

*											IRSTAT13	IRSTAT12	IRSTAT11	IRSTAT10	WAITING	PCOFST				PCINV	FFRA	IRSTAT0	IRSTAT1	IRSTAT2	IRSTAT3	IRSTAT4	IRSTAT5	IRSTAT6	IRSTAT7	IRSTAT8	IRSTAT9
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

Figure A-30. Control State Register (CTL)

SOVH	OVH	FGH	FXH	FINVH	FDBZH	FUNFH	FOVFH	0		FINXS	FINVS	FDBZS	FUNFS	FOVFS	MODE	SOV	OV	FG	FX	FINV	FDBZ	FUNF	FOVF	0	FINXE	FINVE	FDBZE	FUNFE	FOVFE	FRMC	
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

Figure A-31. SPE Status and Control Register (SPEFSCR)

WID				0																											
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

Figure A-32. L1 Cache Control and Status Register 0 (L1CSR0)

0															ICECE	ICEI	0	ICEDT		0	ICUL	ICLO	ICLFC	ICLOA	ICEA		ICORG	0	ICABT	ICINV	ICE
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

Figure A-33. L1 Cache Control and Status Register 1 (L1CSR1)

CARCH		CWPA	CFAHA	DCFISWA	0		DCBSIZE		DCREPL		DCLA	DCPA	DCNWAY								DCSIZE										
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

Figure A-34. L1 Cache Configuration Register 0 (L1CFG0)

0				ICFISWA	0		ICBSIZE		ICREPL		ICLA	ICPA	ICNWAY								ICSIZE										
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

Figure A-35. L1 Cache Configuration Register 1 (L1CFG1)

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						CWAY		0													CSET						0			CCMD	

Figure A-36. L1 Flush/Invalidate Register (L1FINV1)

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								RASIZE							0		NPIDS				PIDSIZE					0		NTLBS		MAVN	

Figure A-37. MMU Configuration Register (MMUCFG)

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
ASSOC								MINSIZE				MAXSIZE				IPROT	AVAIL	P2PSA	0	NENTRY											

Figure A-38. TLB Configuration Register (TLB0CFG, TLB1CFG)

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																														TLB1_FI	0

Figure A-39. MMU Control and Status Register 0 (MMUCSR0)

	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
MAS0	0		TLBSEL (01)		0								ESEL				0												NV			
MAS1	VALID	IPROT	0						TID								0			TS	TSIZ					0						
MAS2	EPN																						0				VLE	W	I	M	G	E
MAS3	RPN																						U0	U1	U2	U3	UX	SX	UW	SW	UR	SR
MAS4	0		TLBSELD		0										TIDSELD		0				TSIZED					0	VLED	WD	ID	MD	GD	ED
MAS6	0								SPID								0															SAS
	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

Figure A-40. MMU Assist Registers Summary

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																										CNTEN	0		RDEN	WREN	INIT

DCR - 272; Read/Write; Reset - 0x0

Figure A-41. Parallel Signature Control Register (PSCR)

0																															TERR
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

DCR - 273; Read/Write; Reset -Unaffected

Figure A-42. Parallel Signature Status Register (PSSR)

High Signature																															
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

DCR - 274; Read/Write; Reset -Unaffected

Figure A-43. Parallel Signature High Register (PSHR)

Low Signature																															
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

DCR - 275; Read/Write; Reset -Unaffected

Figure A-44. Parallel Signature Low Register (PSLR)

Counter																															
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

DCR - 276; Read/Write; Reset -Unaffected

Figure A-45. Parallel Signature Counter Register (PSCTR)

High Signature Update Data																															
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

DCR - 277; Write-only; Reset -Unaffected

Figure A-46. Parallel Signature Update High Register (PSUHR)

Low Signature Update Data																															
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

DCR - 278; Write-only; Reset -Unaffected

Figure A-47. Parallel Signature Update Low Register (PSULR)

Appendix B
Revision History

This appendix provides a list of the major differences between revisions of the *e200z4 Power Architecture™ Core Reference Manual*. This is the initial version of the manual, so there are currently no differences.