e200z6RM
6/2004
Rev. 0

e200z6 PowerPC™ Core Reference Manual

freescale™
semiconductor

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

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

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

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

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

freescale™
semiconductor

Contents

Contents

Contents

Contents

Contents

Chapter 5
Interrupts and Exceptions

Contents

Contents

Chapter 7
Instruction Pipeline and Execution Timing

Contents

Chapter 8
External Core Complex Interfaces

Contents

Contents

Contents

Contents

Figures

Figures

Figures

Figures

Figures

Figures

Figure Number	Title	Page Number

Tables

Tables

Tables

Tables

Tables

Tables

Table Number	Title	Page Number

Chapter 1
e200z6 Overview

This chapter provides an overview of the PowerPC e200z6 microprocessor core. It includes the following:

- An overview of the Book E version of the PowerPC architecture features as implemented in this core
- A summary of the core feature set
- An overview of the programming model
- An overview of interrupts and exception handling
- A summary of instruction pipeline and flow
- A description of the memory-management architecture
- High-level details of the e200z6 core memory and coherency model
- A summary of the Book E architecture compatibility and migration from the original version of the PowerPC architecture as it is defined by Apple, IBM, and Motorola (referred to as the AIM version of the PowerPC architecture)
- Information regarding e200z6 features that are defined by the Motorola Book E implementation standards (EIS)

1.1 Overview of the e200z6

The e200z6 processor family is a set of CPU cores that implement low-cost versions of the PowerPC Book E architecture. e200z6 processors are designed for deeply embedded control applications that require low-cost solutions rather than maximum performance.

Figure 1-1 is a block diagram of the e200z6 core.

The e200z6 is a single-issue, 32-bit Book E–compliant design with 64-bit general-purpose registers (GPRs).

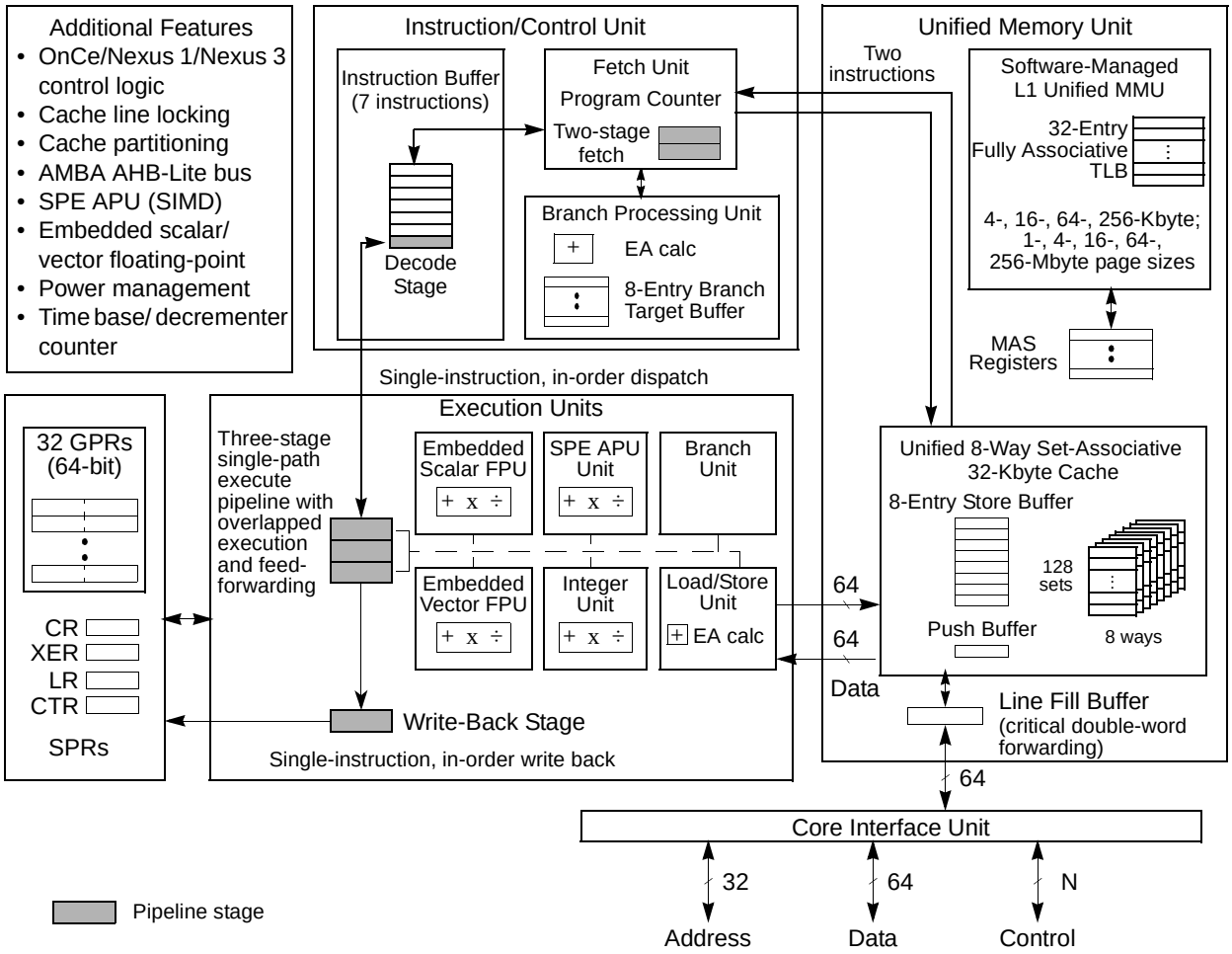


Figure 1-1. e500z6 Block Diagram

A signal processing extension (SPE) APU and embedded vector and scalar floating-point APUs are provided to support real-time integer and single-precision, embedded numeric operations using the GPRs. The e200z6 does not support Book E floating-point instructions in hardware, but traps them so they can be emulated by software.

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

The e200z6 contains a32-Kbyte unified cache and memory management unit (MMU). A Nexus Class 3+ module is also integrated.

The e200z6 platform is specified in such a way that functional units can be added or removed. The e200z6 can be configured with a powerful vectored interrupt controller and one or more IP slave interfaces, as well as support for configured memory units.

1.1.1 Features

The following lists key features of the e200z6:

- Single-issue, 32-bit Book E–compliant core
- In-order execution and retirement
- Precise exception handling
- Branch processing unit (BPU)
  - — Dedicated branch address calculation adder
  - — Branch target prefetching using an eight-entry branch target buffer (BTB)
- Load/store unit (LSU)
  - — 3-cycle load latency
  - — Fully pipelined
  - — Big- and little-endian support on a per-page basis
  - — Misaligned access support
- 64-bit GPR file
- AMBA™ (advanced microcontroller bus architecture) AHB (advanced high-performance bus)-Lite 64-bit system bus
- MMU with 32-entry fully associative TLB and multiple page-size support
- 32-Kbyte, 8-way set-associative unified cache
- Signal processing extension (SPE) APU supporting integer operations using both halves of the 64-bit GPRs
- Single-precision embedded scalar floating-point APU
- Single-precision embedded vector floating-point APU that uses both halves of the 64-bit GPRs
- Nexus Class 3+ real-time development unit
- Power management
  - — Low-power design–extensive clock gating
  - — Power-saving modes: doze, nap, sleep
  - — Dynamic power management of execution units, caches, and MMUs
- e200z6-specific debug interrupt. The e200z6 implements the debug interrupt as defined in Book E with the following changes:
  - — When the debug APU is enabled (MSR[DE] = 1), 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**) is implemented to support the debug APU save/restore registers (DSRR0 and DSRR1).

  - 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** instructions.
- Testability
  - Synthesizeable, full MuxD scan design
  - ABIST/MBIST for arrays

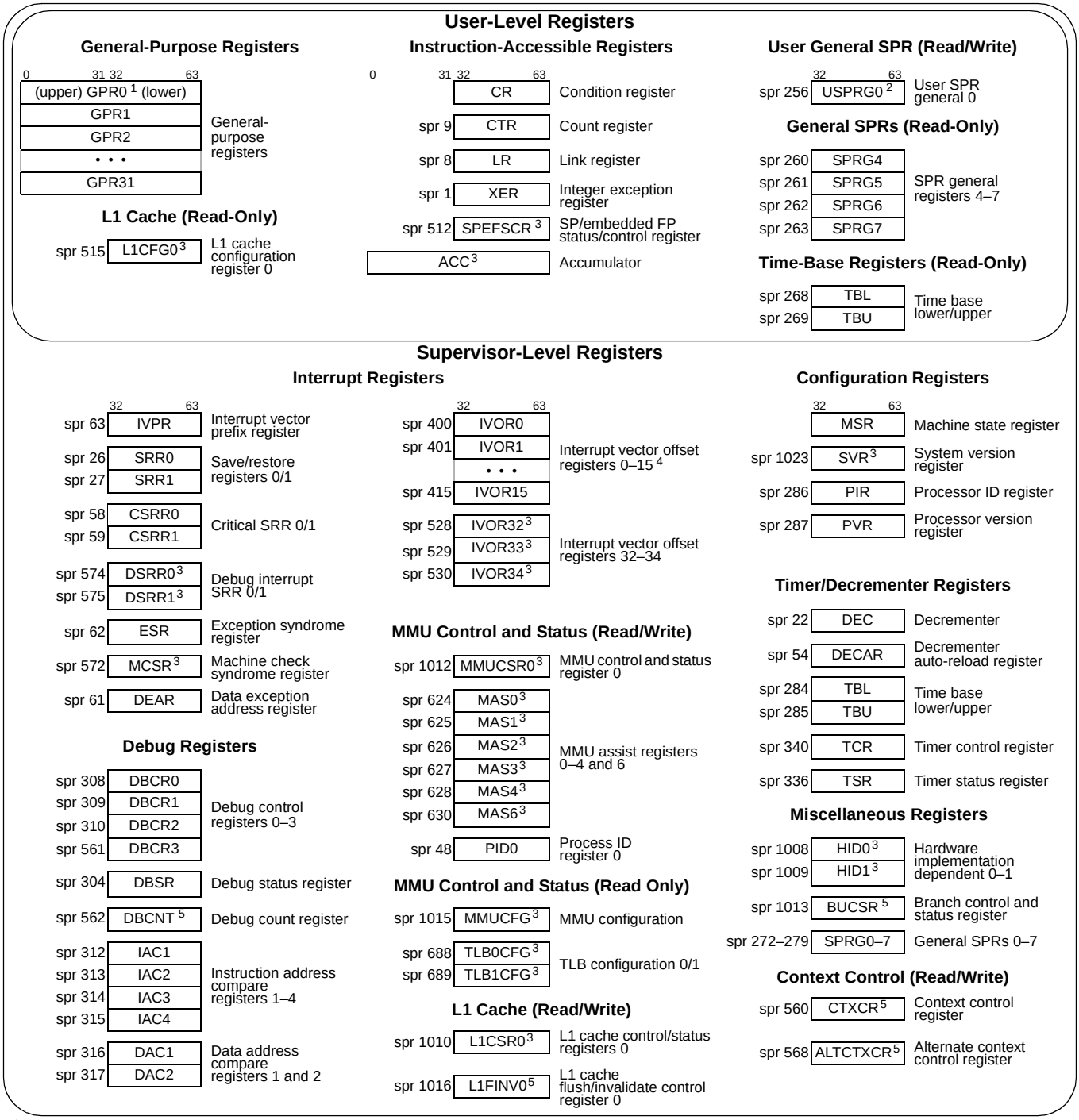
1.2 Programming Model

This section describes the register model, instruction model, and the interrupt model as they are defined by Book E, Motorola EIS, and the e200z6 implementation.

1.2.1 Register Set

Figure 1-2 shows the e200z6 register set, indicating which registers are accessible in supervisor mode and which are accessible in user mode. The number to the left 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.)

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



1 The 64-bit GPR registers are accessed by the SPE as separate 32-bit registers by SPE instructions. Only SPE vector instructions can access the upper word.
2 USPRG0 is a separate physical register from SPRG0.
3 EIS–specific registers; not part of the Book E architecture.
4 IVOR9 (handles auxiliary processor unavailable interrupt) is defined by the EIS but not supported by the e200z6.
5 e200z6-specific registers

Figure 1-2. e200z6 Programmer's Model

1.3 Instruction Set

The e200z6 implements the following instructions:

- The Book E instruction set for 32-bit implementations. This is composed primarily of the user-level instructions defined by the PowerPC user instruction set architecture (UISA). The e200z6 does not include the Book E floating-point, load string, or store string instructions.
- The e200z6 supports the following implementation-specific instructions:
  - Integer select APU. This APU consists of the Integer Select instruction (**isel**), which functions as an if-then-else statement that selects between two source registers by comparison to a CR bit. This instruction eliminates conditional branches, takes fewer clock cycles than the equivalent coding, and reduces the code footprint.
  - Cache line lock and unlock APU. The cache block lock and unlock APU consists of the instructions described in Table 1-1.

Table 1-1. Cache Block Lock and Unlock APU Instructions

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

  - Debug APU. This APU defines the Return from Debug Interrupt instruction (**rfdi**).
  - SPE APU vector instructions. New vector instructions are defined that view the 64-bit GPRs as being composed of a vector of two 32-bit elements. (Some of the instructions also read or write 16-bit elements.) Some scalar instructions are defined for DSP that produce a 64-bit scalar result.
  - The embedded floating-point APUs provide single-precision scalar and vector floating-point instructions. Scalar floating-point instructions use only the lower 32 bits of the GPRs for single-precision floating-point calculations. Table 1-2 lists embedded floating-point instructions.

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

Instruction	Mnemonic		Syntax
	Scalar	Vector	
Convert Floating-Point from Signed Fraction	**efscfsf**	**evfscfsf**	**r**D,**r**B
Convert Floating-Point from Signed Integer	**efscfsi**	**evfscfsi**	**r**D,**r**B
Convert Floating-Point from Unsigned Fraction	**efscfuf**	**evfscfuf**	**r**D,**r**B
Convert Floating-Point from Unsigned Integer	**efscfui**	**evfscfui**	**r**D,**r**B
Convert Floating-Point to Signed Fraction	**efsctsf**	**evfsctsf**	**r**D,**r**B
Convert Floating-Point to Signed Integer	**efsctsi**	**evfsctsi**	**r**D,**r**B
Convert Floating-Point to Signed Integer with Round toward Zero	**efsctsiz**	**evfsctsiz**	**r**D,**r**B
Convert Floating-Point to Unsigned Fraction	**efsctuf**	**evfsctuf**	**r**D,**r**B
Convert Floating-Point to Unsigned Integer	**efsctui**	**evfsctui**	**r**D,**r**B
Convert Floating-Point to Unsigned Integer with Round toward Zero	**efsctuiz**	**evfsctuiz**	**r**D,**r**B
Floating-Point Absolute Value	**efsabs**	**evfsabs**	**r**D,**r**A
Floating-Point Add	**efsadd**	**evfsadd**	**r**D,**r**A,**r**B
Floating-Point Compare Equal	**efscmpeq**	**evfscmpeq**	**cr**D,**r**A,**r**B
Floating-Point Compare Greater Than	**efscmpgt**	**evfscmpgt**	**cr**D,**r**A,**r**B
Floating-Point Compare Less Than	**efscmplt**	**evfscmplt**	**cr**D,**r**A,**r**B
Floating-Point Divide	**efsdiv**	**evfsdiv**	**r**D,**r**A,**r**B
Floating-Point Multiply	**efsmul**	**evfsmul**	**r**D,**r**A,**r**B
Floating-Point Negate	**efsneg**	**evfsneg**	**r**D,**r**A
Floating-Point Negative Absolute Value	**efsnabs**	**evfsnabs**	**r**D,**r**A
Floating-Point Subtract	**efssub**	**evfssub**	**r**D,**r**A,**r**B
Floating-Point Test Equal	**efststeq**	**evfststeq**	**cr**D,**r**A,**r**B
Floating-Point Test Greater Than	**efststgt**	**evfststgt**	**cr**D,**r**A,**r**B
Floating-Point Test Less Than	**efststlt**	**evfststlt**	**cr**D,**r**A,**r**B

1.4 Interrupts and Exception Handling

The core supports an extended exception handling model, with nested interrupt capability and extensive interrupt vector programmability. The following sections define the exception model, including an overview of exception handling as implemented on the e200z6 core, a brief description of the exception classes, and an overview of the registers involved in the processes.

1.4.1 Exception Handling

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

Once the handler is executing, the implementation may need to check one or more bits in the exception syndrome register (ESR), the machine check syndrome register (MCSR), or the signal processing and embedded floating-point status and control register (SPEFSCR), depending on the exception type, to verify the specific cause of the exception and take appropriate action.

The core complex supports the interrupts described in Section 1.4.4, “Interrupt Registers.”

1.4.2 Interrupt Classes

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

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

1.4.3 Interrupt Types

The e200z6 core processes all interrupts as either debug, critical, or noncritical types. Separate control and status register sets are provided for each type of interrupt. The core handles interrupts from these three categories in the following priority order:

1. Debug interrupt—The e200z6 core defines a separate set of resources for the debug interrupt. They use the debug save and restore registers (DSRR0/DSRR1) to save state when they are taken, and they use the **rfdi** instruction to restore state. These interrupts can be masked by the debug enable bit, MSR[DE]. If MSR[DE] = 0, the debug interrupt is disabled, and debug interrupts are handled as critical interrupts.
2. Noncritical interrupts—First-level interrupts that allow the processor to change program flow to handle conditions generated by external signals, errors, or unusual conditions arising from program execution or from programmable timer-related events. These interrupts are largely identical to those previously defined by the OEA portion of the PowerPC architecture. They use the save and restore registers (SRR0/SRR1) to save state when they are taken, and they use the **rfi** instruction to restore state. Asynchronous noncritical interrupts can be masked by the external interrupt enable bit, MSR[EE].
3. Critical interrupts—Critical interrupts can be taken during a noncritical interrupt or during regular program flow. They use the critical save and restore registers (CSRR0/CSRR1) to save state when they are taken, and they use the **rfci** instruction to restore state. These interrupts can be masked by the critical enable bit, MSR[CE]. Book E defines the critical input, watchdog timer, and machine check interrupts as critical interrupts, but the e200z6 core defines a third set of resources for the debug interrupt, as described in Table 1-3.

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

1.4.4 Interrupt Registers

The registers associated with interrupt and exception handling are described in Table 1-3.

Table 1-3. Interrupt Registers

Register	Description
Noncritical Interrupt Registers	
SRR0	Save/restore register 0 —Stores the address of the instruction causing the exception or the address of the instruction that will execute after the **rfi** instruction.
SRR1	Save/restore register 1—Saves machine state on noncritical interrupts and restores machine state after an **rfi** instruction is executed.

Table 1-3. Interrupt Registers (continued)

Register	Description
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 will execute 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—Used to store the address of the instruction that will execute after an **rfdi** instruction is executed.
DSRR1	Debug save/restore register 1—Stores machine state on debug interrupts and restores machine state after an **rfdi** 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 between the different kinds of exceptions that generate the same interrupt type. Upon generation of a specific exception type, the associated bit(s) is set and all other bits are cleared.
SPE APU Interrupt Registers	
SPEFSCR	Signal processing and embedded floating-point status and control register—Provides interrupt control and status as well as various condition bits associated with the operations performed by the SPE APU.
Other Interrupt Registers	
DEAR	Data exception address register—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] || IVORn [48–59] || 0b0000 define the address of an interrupt-processing routine. See Table 1-4 and Chapter 5, “Interrupts and Exceptions,” for more information.

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 1-3 lists IVOR registers implemented on the e200z6 core and the associated interrupts.

Table 1-4. Exceptions and Conditions

Interrupt Type	IVORn	Causal Conditions	Section/Page
System reset (not an interrupt)	None [1],	• Reset by assertion of *p_reset_b* • Watchdog timer reset control • Debug reset control	—
Critical input	IVOR 0 [2]	*p_critint_b* is asserted and MSR[CE]=1.	5.6.1/5-9

Table 1-4. Exceptions and Conditions (continued)

Interrupt Type	IVORn	Causal Conditions	Section/Page
Machine check	IVOR 1	• *p_mcp_b* is asserted and MSR[ME] =1. • ISI, ITLB error on first instruction fetch for an exception handler and current MSR[ME] = 1 • Parity error signaled on cache access and current MSR[ME]=1 • Write bus error on buffered store or cache line push	5.6.2/5-10
Data storage	IVOR 2	• Access control • Byte ordering due to misaligned access across page boundary to pages with mismatched E bits • Cache locking exception • Precise external termination error (*p_tea_b* assertion and precise recognition)	5.6.3/5-12
Instruction storage	IVOR 3	• Access control • Precise external termination error (*p_tea_b* assertion and precise recognition)	5.6.4/5-13
External input	IVOR 4 [2]	*p_extint_b* is asserted and MSR[EE]=1.	5.6.5/5-14
Alignment	IVOR 5	• **lmw**, **stmw** not word aligned • **lwarx** or **stwcx.** not word aligned • **dcbz** with disabled cache or no cache present, or to W or I storage • Misaligned SPE load and store instructions	5.6.6/5-14
Program	IVOR 6	Illegal, privileged, trap, floating-point enabled, APU enabled, unimplemented operation.	5.6.7/5-15
Floating-point unavailable	IVOR 7	MSR[FP] = 0 and attempt to execute a Book E floating-point operation	5.6.8/5-16
System call	IVOR 8	Execution of the System Call (**sc**) instruction	5.6.9/5-17
APU unavailable	IVOR 9	Unused by the e200z6	5.6.10/5-17
Decrementer	IVOR 10	As specified in Book E	5.6.11/5-17
Fixed-interval timer	IVOR 11	As specified in Book E	5.6.12/5-18
Watchdog timer	IVOR 12	As specified in Book E	5.6.13/5-19
Data TLB error	IVOR 13	Data translation lookup did not match a valid entry in the TLB.	5.6.14/5-20
Instruction TLB error	IVOR 14	Instruction translation lookup did not match a valid entry in the TLB	5.6.15/5-20
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	5.6.16/5-21
Reserved	IVOR 16–31	—	—
SPE unavailable exception	IVOR 32	See Section 5.6.18, “SPE APU Unavailable Interrupt (IVOR32).”	5.6.18/5-25
SPE data exception	IVOR 33	See Section 5.6.19, “SPE Floating-Point Data Interrupt (IVOR33).”	5.6.19/5-25
SPE round exception	IVOR 34	See Section 5.6.20, “SPE Floating-Point Round Interrupt (IVOR34).”	5.6.20/5-26

[1] Vector to [p_rstbase[0:19]] || 0xFFC
[2] Autovectored external and critical input interrupts use this IVOR. Vectored interrupts supply an interrupt vector offset directly.

1.5 Microarchitecture Summary

The e200z6 processor has a seven-stage pipeline for instruction execution.

1. Instruction fetch 0
2. Instruction fetch 1
3. Instruction decode/register file read
4. Execute 0
5. Execute 1/memory access 0
6. Execute 2/memory access 1
7. Register writeback

These stages are pipelined, allowing single-clock instruction throughput for most instructions.

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

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

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

Branch target addresses are calculated in parallel with branch instruction decode, resulting in execution time of 3 clocks. Conditional branches which are not taken execute in a single clock. Branches with successful BTB target prefetching have an effective latency of 1 clock.

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.

The condition register unit supports the condition register (CR) and condition register operations defined by the PowerPC architecture. The CR 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, and provide 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 APU supports vector instructions operating on 16- and 32-bit integer and fractional data types. The vector and scalar floating-point APUs operate on 32-bit IEEE-754 single-precision floating-point formats, and support single-precision floating-point operations in a pipelined fashion.

The 64-bit GPRs are used for source and destination operands for all vector instructions, and there is a unified storage model for single-precision floating-point data types of 32 bits and the normal integer type. Low-latency integer and floating-point add, subtract, multiply, divide, compare, and conversion operations are provided, and most operations can be pipelined.

1.5.1 Instruction Unit Features

The features of the e200z6 instruction unit are as follows:

- 64-bit path to cache supports fetching of two 32-bit instructions per clock
- Instruction buffer holds up to seven sequential instructions
- Dedicated PC (program counter) incrementer supporting instruction fetches
- Branch processing unit with dedicated branch address adder and branch target buffer (BTB) supporting single-cycle execution of successfully predicted branches
- Target instruction buffer holds up to two prefetched branch target instructions

1.5.2 Integer Unit Features

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

- 32-bit AU for arithmetic and comparison operations
- 32-bit LU for logical operations
- 32-bit priority encoder for count-leading-zeros function
- 32-bit single-cycle barrel shifter for static shifts and rotates
- 32-bit mask unit for data masking and insertion
- Divider logic for signed and unsigned divide in 15 clocks with minimized execution timing

- Pipelined 32×32 hardware multiplier array supports $32 \times 32 \rightarrow 32$ multiply with 3-clock latency, 1-clock throughput

1.5.3 Load/Store Unit (LSU) Features

The e200z6 LSU supports load, store, and the load multiple/store multiple instructions:

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

1.5.4 L1 Cache Features

The features of the cache are as follows:

- 32-Kbyte, 8-way set-associative unified cache
- Partitionable cache
- Copy-back and write-through support
- Eight-entry store buffer
- Push buffer
- Unified line-fill buffer with critical double-word forwarding for both data loads and instruction fetches
- 32-bit address bus plus attributes and control
- Separate unidirectional 64-bit read and 64-bit write data buses
- Cache line locking supported by the Motorola Book E cache line locking APU
  - — Data cache locking control instructions—Data Cache Block Touch and Lock Set (**dcbtls**), Data Cache Block Touch for Store and Lock Set (**dcbtstls**), and Data Cache Block Lock Clear (**dcblc**)
  - — Instruction cache locking control instructions—Instruction Cache Block Touch and Lock Set (**icbtls**) and Instruction Cache Block Lock Clear (**icblc**)
- Way allocation
- Tag and data parity
- e200z6-specific L1 cache flush and invalidate register (L1FINV0) supports software-based flush and invalidation control on a set and way basis.

1.5.5 MMU Features

The e200z6 memory management unit (MMU) is a 32-bit Book E–compliant PowerPC implementation, with the following feature set:

- Motorola Book E MMU architecture compliant
- Translates from 32-bit effective to 32-bit real addresses
- 8-bit process identifier (PID)
- 32-entry fully associative TLB
- Support for multiple 4-, 16-, 64-, 256-Kbyte; 1-, 4-, 16-, 64-, 256-Mbyte page sizes
- Hardware assist for TLB miss exceptions
- Software managed by **tlbre**, **tlbwe**, **tlbsx**, **tlbsync**, and **tlbivax** instructions
- Entry flush protection
- Byte ordering (endianness) configurable on a per-page basis

1.5.6 e200z6 System Bus (Core Complex Interface) Features

The features of the e200z6 core complex interface are as follows:

- AMBA AHB-Lite protocol
- 32-bit address bus plus attributes and control
- Separate unidirectional 64-bit read data bus and 64-bit write data bus
- Pipelined, in-order accesses

1.5.7 Nexus3 Module Features

The Nexus3 module provides real-time development capabilities for e200z6 processors in compliance with the IEEE-ISTO Nexus 5001-2003 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/Nexus1 unit. The IEEE-ISTO 5001-2003 standard defines an extensible auxiliary port which is used in conjunction with the JTAG port in e200z6 processors.

1.6 Legacy Support of PowerPC Architecture

This section provides an overview of the architectural differences and compatibilities of the e200z6 core compared with the AIM PowerPC architecture. The two levels of the e200z6 core programming environment are as follows:

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

In general, the e200z6 core supports the user-level architecture from the existing AIM architecture. The following subsections are intended to highlight the main differences. For specific implementation details refer to the relevant chapter.

1.6.1 Instruction Set Compatibility

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

1.6.1.1 User Instruction Set

The e200z6 core executes legacy user-mode binaries and object files except for the following:

- The e200z6 core supports vector and scalar single-precision floating-point operations as APUs. These instructions have different encoding than the AIM definition of the PowerPC architecture. Additionally, the e200z6 core uses GPRs for floating-point operations, rather than the FPRs defined by the UISA. Most porting of floating-point operations can be handled by recompiling.
- String instructions are not implemented on the e200z6 core; therefore, trap emulation must be provided to ensure backward compatibility.

1.6.1.2 Supervisor Instruction Set

The supervisor-mode instruction set defined by the AIM version of the PowerPC architecture is compatible with the e200z6 core with the following exceptions:

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

1.6.2 Memory Subsystem

Both Book E and the AIM version of the PowerPC architecture provide separate instruction and data memory resources. The e200z6 core provides additional cache control features, including cache locking.

1.6.3 Exception Handling

Exception handling is generally the same as that defined in the AIM version of the PowerPC architecture for the e200z6 core, with the following differences:

- Book E defines a new critical interrupt, providing an extra level of interrupt nesting. The critical interrupt includes external critical and watchdog timer time-out inputs.

- The debug interrupt differs from the Book E and from the AIM definition. It defines the Return from Debug Interrupt instruction, **rfdi**, and two debug save/restore registers, DSRR0 and DSRR1.
- Book E processors can use IVPR and the IVORs to set exception vectors individually, but they can be set to the address offsets defined in the OEA to provide compatibility.
- Unlike the AIM version of the PowerPC architecture, Book E does not define a reset vector; execution begins at a fixed virtual address, 0xFFFF_FFFC. The e200z6 allows this to be hard-wired to any page.
- Some Book E and e200z6 core-specific SPRs are different from those defined in the AIM version of the PowerPC architecture, particularly those related to the MMU functions. Much of this information has been moved to a new exception syndrome register (ESR).
- Timer services are generally compatible, although Book E defines a new decrementer auto reload feature, the fixed-interval timer critical interrupt, and the watchdog timer interrupt, which are implemented in the e200z6 core.

An overview of the interrupt and exception handling capabilities of the e200z6 core can be found in Section 1.4, “Interrupts and Exception Handling.”

1.6.4 Memory Management

The e200z6 core implements a straightforward virtual address space that complies with the Book E MMU definition, which eliminates segment registers and block address translation resources. Book E defines resources for multiple, variable page sizes that can be configured in a single implementation. TLB management is provided with new instructions and SPRs.

1.6.5 Reset

Book E–compliant cores do not share a common reset vector with the AIM version of the PowerPC architecture. Instead, at reset, fetching begins at address 0xFFFF_FFFC. In addition to the Book E reset definition, the EIS and the e200z6 core define specific aspects of the MMU page translation and protection mechanisms. Unlike the AIM version of the PowerPC core, as soon as instruction fetching begins, the e200z6 core is in virtual mode with a hardware-initialized TLB entry.

1.6.6 Little-Endian Mode

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

Chapter 2
Register Model

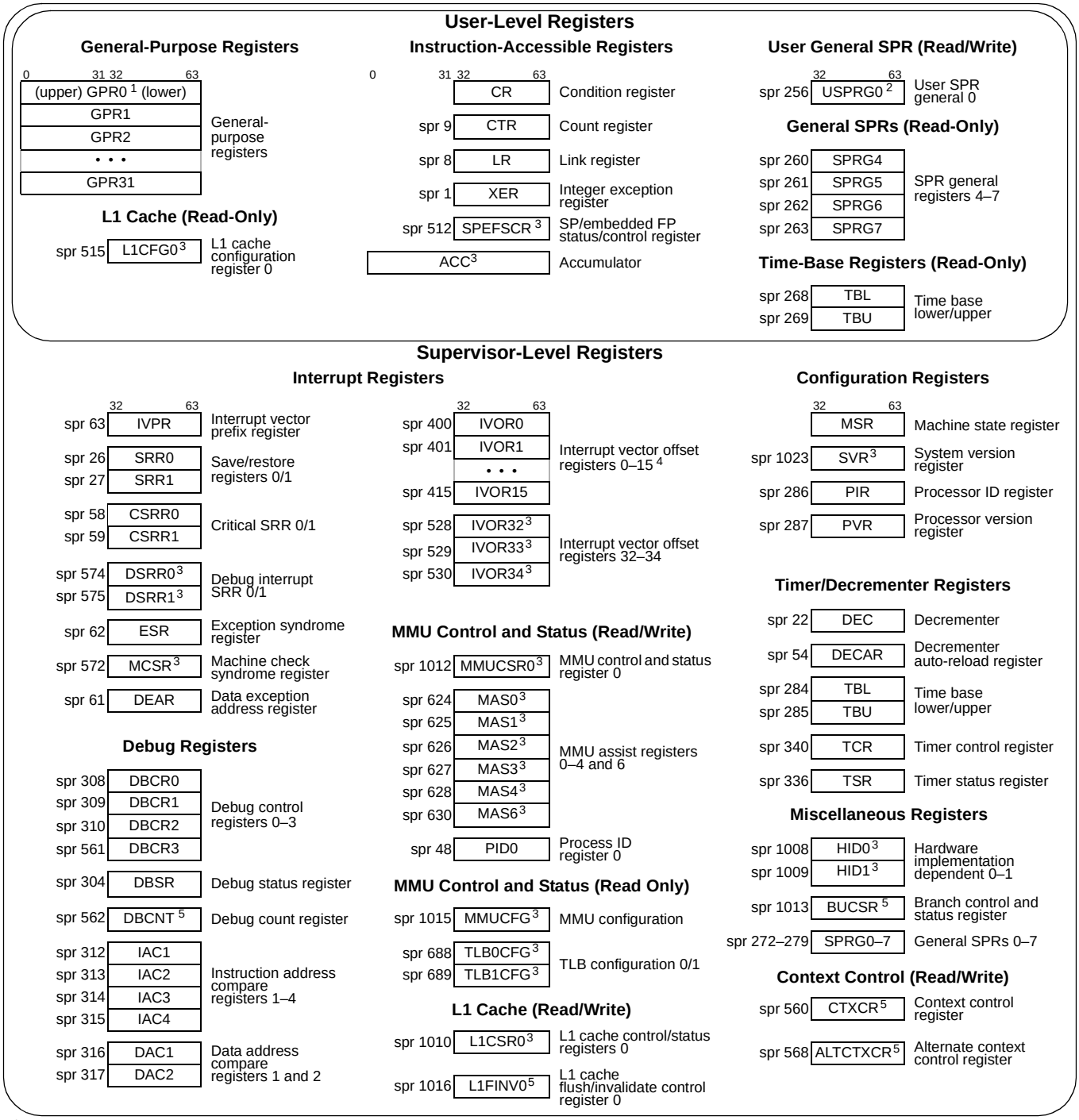
This chapter describes the registers implemented in the e200z6 core. It includes an overview of registers defined by the Book E architecture, highlighting differences in how these registers are implemented in the e200z6 core, and provides a detailed description of the e200z6-specific registers. Full descriptions of the architecture-defined register set are provided in the EREF.

The Book E architecture defines register-to-register operations for all computational instructions. Source data for these instructions is accessed from the on-chip registers or is 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 e200z6 extends the general-purpose registers (GPRs) to 64 bits for supporting SPE APU operations. PowerPC Book E 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 APU defines load and store instructions for transferring 64-bit values to/from memory.

Figure 2-1 shows the complete e200z6 register set, indicating which registers are accessible in supervisor mode and which are accessible in user mode. The number to the left 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.

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**)) or implicit as part of the execution of an instruction. Some registers are accessed both explicitly and implicitly.



1 The 64-bit GPR registers are accessed by the SPE as separate 32-bit registers by SPE instructions. Only SPE vector instructions can access the upper word.

2 USPRG0 is a separate physical register from SPRG0.

3 EIS–specific registers; not part of the Book E architecture.

4 IVOR9 (handles auxiliary processor unavailable interrupt) is defined by the EIS but not supported by the e200z6.

5 e200z6-specific registers

Figure 2-1. e200z6 Programmer's Model

2.1 PowerPC Book E Registers

The e200z6 supports most of the registers defined by Book E architecture. Notable exceptions are the floating-point registers FPR0–FPR31 and the FPSCR. The e200z6 does not support the Book E floating-point architecture in hardware. The GPRs have been extended to 64 bits. The Book E registers implemented by the e200z6 are described as follows (e200z6-specific registers are described in the next section):

- User-level registers—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. PowerPC Book E instructions affect only the lower 32 bits of the GPRs. SPE APU instructions operate on the entire 64-bit register.
  - — Condition register (CR). The 32-bit CR consists of eight 4-bit fields, CR0–CR7, that reflect results of certain arithmetic operations and provide a mechanism for testing and branching.

  The remaining user-level registers are SPRs. Note that the PowerPC architecture provides the **mtspr** and **mfspr** instructions for accessing SPRs.
  - — Integer exception register (XER). The XER indicates overflow and carries for integer operations.
  - — Link register (LR). The LR provides the branch target address for the branch conditional to link register (**bclr**, **bclrl**) instructions and is used to hold the address of the instruction that follows a branch and link instruction, typically used for linking to subroutines.
  - — Count register (CTR). CTR holds a loop count that can be decremented during execution of appropriately coded branch instructions. CTR also provides the branch target address for the Branch Conditional to Count Register (**bcctr**, **bcctrl**) instructions.
  - — The time base facility (TB) consists of two 32-bit registers—time base upper (TBU) and time base lower (TBL). These two registers may be read (but not written) by user-level software.
  - — SPRG4–SPRG7. Book E defines software-use special purpose registers (SPRGs). SPRG4–SPRG7 are read only by user-level software. The e200z6 does not allow user-mode access to SPRG3. (Such access is defined as implementation dependent by Book E.)
  - — USPRG0. Book E defines user-software-use SPR USPRG0, which is read-write accessible by user-level software.
- Supervisor-level registers—In addition to the registers accessible in user mode, supervisor-level software has access to additional control and status registers an

operating system might use for configuration, exception handling, and other operating system functions. Book E defines the following supervisor-level registers:

— Processor control registers
  – Machine state register (MSR). The MSR defines the state of the processor. The MSR can be modified by the Move to Machine State Register (**mtmsr**), System Call (**sc**), and return from interrupt (**rfi**, **rfci**, **rfdi)** instructions. It can be read by the Move from Machine State Register (**mfmsr)** instruction. When an interrupt occurs, the contents of the MSR are saved to one of the machine state save/restore registers (SRR1, CSRR1, DSRR1).
  – Processor version register (PVR). This is a read-only register that identifies the version (model) and revision level of the PowerPC processor.
  – Processor identification register (PIR). This read-only register is provided to distinguish the processor from other processors in the system.

— Process ID register (PID0, also referred to as PID). Provided to indicate the current process or task identifier. It is used by the MMU as an extension to the effective address, and by external Nexus 2/3/4 modules for ownership trace message generation. PowerPC Book E allows for multiple PIDs; the e200z6 implements only one.

— SPRG0–SPRG7, USPRG0. SPRG0–SPRG7 and USPRG0 are provided for software use. See Section 2.8, “Software-Use SPRs (SPRG0–SPRG7 and USPRG0),” for more information on these registers.

  Note that the e200z6 does not allow user-mode access to the SPRG3 register. (Access to SPRG3 is defined as implementation dependent by Book E.)

— Interrupt registers
  – Data exception address register (DEAR). After most data storage interrupts (DSIs), or on an alignment interrupt or data TLB interrupt, DEAR is set to the effective address (EA) generated by the faulting instruction.
  – Exception syndrome register (ESR). ESR provides a syndrome to differentiate between the different kinds of exceptions that can generate the same interrupt.
  – Interrupt vector prefix register (IVPR) and the interrupt-specific interrupt vector offset registers (IVORs). These registers together provide the address of the interrupt handler for different classes of interrupts.
  – Save/restore register 0 (SRR0). SRR0 is used to save machine state on a non-critical interrupt, and contains the address of the instruction at which execution resumes when an **rfi** instruction is executed at the end of a non-critical-class interrupt handler routine.
  – Save/restore register 1 (SRR1). SRR1 is used to save machine state from the MSR on non-critical interrupts, and to restore machine state when **rfi** executes.

  - Critical save/restore register 0 (CSRR0). CSRR0 is used to save machine state on a critical interrupt, and contains the address of the instruction at which execution resumes when an **rfci** instruction is executed at the end of a critical-class interrupt handler routine.
  - Critical save/restore register 1 (CSRR1). CSRR1 is used to save machine state from the MSR on critical interrupts and to restore machine state when **rfci** executes.

— Debug facility registers

  - Debug control registers (DBCR0–DBCR2). These registers provide control for enabling and configuring debug events.
  - Debug status register (DBSR). This register contains debug event status.
  - Instruction address compare registers (IAC1–IAC4). These registers contain addresses and/or masks which are used to specify instruction address compare debug events.
  - Data address compare registers (DAC1–DAC2). These registers contain addresses and/or masks used to specify data address compare debug events.

    Note that the e200z6 does not implement the data value compare registers (DVC1 and DVC2).

— Timer registers

  - Time base (TB). The 64-bit time base is provided for maintaining the time of day and operating interval timers. The TB consists of two 32-bit registers, time base upper (TBU) and time base lower (TBL). The time base registers can be written to only by supervisor-level software, but can be read by both user and supervisor-level software.
  - Decrementer register (DEC). This 32-bit decrementing counter 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). TCR controls decrementer, fixed-interval timer, and watchdog timer options.
  - Timer status register (TSR). TSR contains status on timer events and the most recent watchdog-timer-initiated processor reset.

2.2 e200z6-Specific Registers

Book E allows implementation-specific registers. Those incorporated in the e200z6 core are as follows:

- User-level registers —The user-level registers can be accessed by all software with either user or supervisor privileges. They include the following:

— Signal processing/embedded floating-point status and control register (SPEFSCR). The SPEFSCR contains all integer and floating-point exception signal bits, exception summary bits, exception enable bits, and rounding control bits needed for compliance with the IEEE 754 standard.

— L1 cache configuration register (L1CFG0). This read-only register allows software to query the configuration of the L1 unified cache.

- Supervisor-level registers—The following supervisor-level registers are defined in the e200z6 in addition to the Book E registers described above:
  — Configuration registers—Hardware implementation-dependent registers 0 and 1 (HID0 and HID1). These registers control various processor and system functions.
  — Exception handling and control registers
    – Machine check syndrome register (MCSR). This register provides a syndrome to differentiate between the different kinds of conditions that can generate a machine check.
    – Debug save/restore register 0 (DSRR0). When the debug APU is enabled, DSRR0 is used to save the address of the instruction at which execution continues when **rfdi** executes at the end of a debug interrupt handler routine.
    – Debug save/restore register 1 (DSRR1). When the debug APU is enabled, (MSR[DE] = 1, DSRR1 used to save machine state from the MSR on debug interrupts and to restore machine state when **rfdi** executes.
  — Debug facility registers
    – Debug control register 3 (DBCR3). This register provides control for debug functions not described in Book E
    – Debug counter register (DBCNT). This register provides counter capability for debug functions
  — Context control registers
    – Context control register (CTXCR). This register provides control for register context selection.
    – Alternate context control register (ALTCTXCR). This virtual register provides access to the context control register of each register context when multiple register contexts exist.
  — Branch unit control and status register (BUCSR). This register controls operation of the branch target buffer (BTB).
  — Cache registers
    – L1 cache configuration register (L1CFG0). This read-only register allows software to query the configuration of the L1 cache.

  - L1 cache control and status register (L1CSR0). This register controls operation of the L1 unified cache, providing such facilities as cache enabling, cache invalidation, and cache locking.
  - L1 cache flush and invalidate register (L1FINV0). This register controls software flushing and invalidation of the L1 unified cache.
- Memory management unit (MMU) 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 e200z6 core from the MMU.
  - MMU control and status register (MMUCSR0). This register controls MMU invalidation.
  - TLB configuration registers (TLB0CFG and TLB1CFG). These are read-only registers that allow software to query the configuration of the TLBs.
- System version register (SVR). SVR is a read-only register that identifies the version (model) and revision level of the system that includes an e200z6 processor.
- The EIS-defined accumulator, which is part of the SPE APU. See Section 2.6.2, “Accumulator (ACC).”

Note that although other processors may implement similar or identical registers, it is not guaranteed that the implementation of e200z6-core-specific registers is consistent among PowerPC processors.

All e200z6 SPR definitions comply with the Motorola Book E definitions.

2.3 Processor Control Registers

2.3.1 Machine State Register (MSR)

The MSR, shown in Figure 2-2, defines the state of the processor. Chapter 5, “Interrupts and Exceptions,” describes how the MSR is affected when interrupts occur.

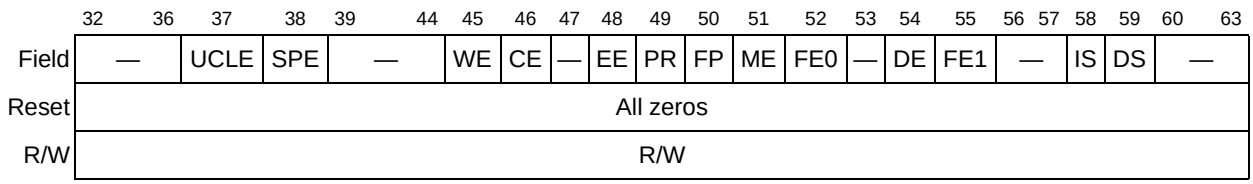


Figure 2-2. Machine State Register (MSR)

MSR fields are described in Table 2-1.

Table 2-1. MSR Field Descriptions

Bits	Name	Description
32–36	—	Reserved, should be cleared.
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 is set in the ESR. 1 Execution of the cache lock instructions in user mode enabled
38	SPE	SPE Available 0 Execution of SPE APU vector instructions is disabled; SPE unavailable exception taken instead, and ESR[SPE] is set. 1 Execution of SPE APU vector instructions is enabled.
39–44	—	Reserved, should be cleared.
45	WE	Wait state (power management) enable. Defined as optional by Book E and implemented in the e200z6. 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 HID0[DOZE,NAP,SLEEP], described in Section 2.11.1, “Hardware Implementation-Dependent Register 0 (HID0).”
46	CE	Critical interrupt enable 0 Critical input and watchdog timer interrupts are disabled. 1 Critical input and watchdog timer interrupts are enabled.
47	—	Preserved
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.
49	PR	Problem state 0 The processor is in supervisor mode, can execute any instruction, and can access any resource (for example, GPRs, SPRs, MSR, etc.). 1 The processor is in user mode, cannot execute any privileged instruction, and cannot access any privileged resource.
50	FP	Floating-point available 0 Floating-point unit is unavailable. The processor cannot execute floating-point instructions, including floating-point loads, stores, and moves. (An FP unavailable interrupt is generated on attempted execution of floating-point instructions). 1 Floating-point unit is available. The processor can execute floating-point instructions. (Note that for the e200z6, the floating-point unit is not supported; an unimplemented operation exception is generated for attempted execution of floating-point instructions when FP is set).
51	ME	Machine check enable 0 Machine check interrupts are disabled. Checkstop mode is entered when the *p_mcp_b* input is recognized asserted or an ISI or ITLB exception occurs on a fetch of the first instruction of an exception handler. 1 Machine check interrupts are enabled.
52	FE0	Floating-point exception mode 0 (not used by the e200z6)
53	—	Reserved, should be cleared.

Table 2-1. MSR Field Descriptions (continued)

Bits	Name	Description
54	DE	Debug interrupt enable 0 Debug interrupt APU is disabled and the Book E defined critical-type debug interrupt is invoked if a debug interrupt occurs. 1 Debug interrupt APU is enabled and the e200z6-defined debug APU interrupt is invoked if a debug interrupt occurs.
55	FE1	Floating-point exception mode 1 (not used by the e200z6)
56–57	—	Reserved, should be cleared.
58	IS	Instruction address space 0 The processor directs all instruction fetches to address space 0 (TS = 0 in the relevant TLB entry). 1 The processor directs all instruction fetches to address space 1 (TS = 1 in the relevant TLB entry).
59	DS	Data address space 0 The core directs all data storage accesses to address space 0 (TS = 0 in the relevant TLB entry). 1 The core directs all data storage accesses to address space 1 (TS = 1 in the relevant TLB entry).
60–63	—	Reserved, should be cleared.

2.3.2 Processor ID Register (PIR)

The processor ID for the CPU core is contained in the processor ID register (PIR), shown in Figure 2-3. The contents of PIR reflect the hardware input signals to the e200z6 core.

	32 … 55	56 … 63
Field	—	PID
Reset	0000_0000_0000_0000_0000_0000	*p_cpuid[0:7]*
R/W	Read only	
SPR	SPR 286	

Figure 2-3. Processor ID Register (PIR)

PIR fields are described in Table 2-2.

Table 2-2. PIR Field Descriptions

Bits	Name	Description
32–55	—	These bits always read 0.
56–63	PID	These bit are a reflection of the values provided on the *p_cpuid[0:7]* input signals.

2.3.3 Processor Version Register (PVR)

The processor version register (PVR), shown in Figure 2-4, contains the processor version number for the CPU core.

	32–35	36–37	38–43	44–47	48–55	56–59	60–63
Field	Manufacturer ID	—	Type	Version	MBG Use	Major Rev	MBG ID
Reset	1000	00	01_0001	0010	*p_pvrin[16:31]*		
R/W	Read only						
SPR	SPR 287						

Figure 2-4. Processor Version Register (PVR)

The PVR contains fields to specify a particular implementation of an e200z6 family member. Interface signals *p_pvrin[16:31]* provide the contents of bits 48–63 of this register.

Table 2-3. PVR Field Descriptions

Bits	Name	Description
32–35	Manufacturer ID	Manufacturer ID. Motorola is 0b1000.
36–37	—	Reserved, should be cleared.
38–43	Type	Identifies the processor type. For the e200z6, this field is 0b01_0001.
44–47	Version	Identifies the version of the processor and inclusion of optional elements. For e200z6, this field is b0010.
48–55	MBG Use	Allocated to distinguish different system variants; provided by the *p_pvrin[16:23]* inputs
56–59	Major Rev	Distinguish between implementations of the version; provided by the *p_pvrin[24:27]* inputs
60–63	MBG ID	These bits are provided by the *p_pvrin[28:31]* input signals.

2.3.4 System Version Register (SVR)

The system version register (SVR) contains system version information for an e200z6-based SoC.

	32–63
Field	Version
Reset	SoC-dependent value (determined by *p_sysvers[0:31]* on the e500 core)
R/W	Read only
SPR	SPR 1023

Figure 2-5. System Version Register (SVR)

SVR is used to specify a particular implementation of an e200z6-based system.

Table 2-4. SVR Field Description

Bits	Name	Description
32–63	Version	This field distinguishes different system variants, and is provided by the *p_sysvers[0:31]* input signals

2.4 Registers for Integer Operations

This section describes the registers used for integer operations.

2.4.1 General-Purpose Registers (GPRs)

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

2.4.2 Integer Exception Register (XER)

The XER, shown in Figure 2-6, tracks exception conditions for integer operations.

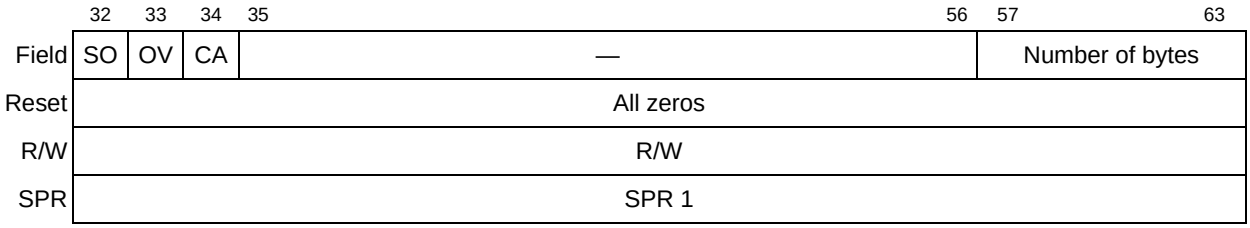


Figure 2-6. Integer Exception Register (XER)

XER fields are described in Table 2-5.

Table 2-5. XER Field Descriptions

Bits	Name	Description
32	SO	Summary overflow. Set when an instruction (except **mtspr**) sets the overflow bit (OV). Once set, SO remains set until it is cleared by **mtspr[XER]** or **mcrxr**. SO is not altered by compare instructions or by other instructions (except **mtspr[XER]** and **mcrxr**) that cannot overflow. Executing **mtspr[XER]**, supplying the values 0 for SO and 1 for OV, causes SO to be cleared and OV to be set.
33	OV	Overflow. X-form add, subtract from, and negate instructions having OE=1 set OV if the carry out of bit 32 is not equal to the carry out of bit 33, and clear OV otherwise to indicate a signed overflow. X-form multiply low word and divide word instructions having OE=1 set OV if the result cannot be represented in 32 bits (**mullwo**, **divwo**, and **divwuo**) and clear OV otherwise. OV is not altered by compare instructions or by other instructions (except **mtspr[XER]** and **mcrxr**) that cannot overflow.

Table 2-5. XER Field Descriptions (continued)

Bits	Name	Description
34	CA	Carry. Add carrying, subtract from carrying, add extended, and subtract from extended instructions set CA if there is a carry out of bit 32 and clear it otherwise. CA can be used to indicate unsigned overflow for add and subtract operations that set CA. Shift right algebraic word instructions set CA if any 1 bits are shifted out of a negative operand and clear CA otherwise. Compare instructions and instructions that cannot carry (except Shift Right Algebraic Word, **mtspr[XER]**, and **mcrxr**) do not affect CA.
35–56	—	Reserved, should be cleared.
57–63	Number of bytes	Supports emulation of load and store string instructions. Specifies the number of bytes to be transferred by a load string indexed or store string indexed instruction.

2.5 Registers for Branch Operations

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

2.5.1 Condition Register (CR)

The 32-bit condition register (CR) reflects the result of certain operations and provides a mechanism for testing and branching.

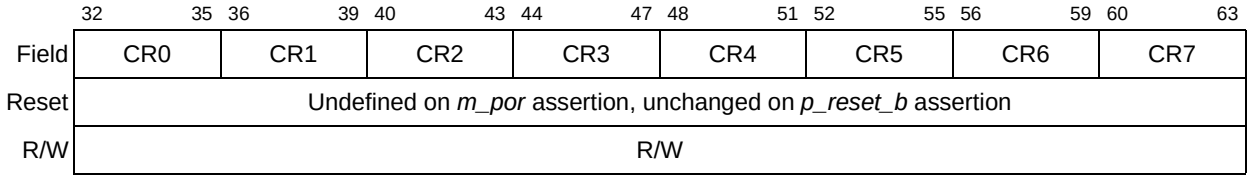


Figure 2-7. Condition Register (CR)

CR bits are grouped into eight 4-bit fields, CR0–CR7, which are set as follows:

- Specified CR fields can be set by a move to the CR from a GPR (**mtcrf**).
- A specified CR field can be set by a move to the CR from another CR field (**mcrf**), or from the XER (**mcrxr**).
- CR0 can be set as the implicit result of an integer instruction.
- A specified CR field can be set as the result of either an integer or a floating-point compare instruction (including SPE and SPFP compare instructions).

Instructions are provided to perform logical operations on individual CR bits and to test individual CR bits.

Note that Book E instructions that access CR bits, such as Branch Conditional (**bc**), CR logicals, and Move to Condition Register Field (**mtcrf**), determine the bit position by adding 32 to the value of the operand. For example, the BI operand in conditional branch instructions accesses the bit BI + 32, as shown in Table 2-6.

Table 2-6. BI Operand Settings for CR Fields

CR*n* Bits	CR Bits	BI	Description
CR0[0]	32	00000	Negative (LT)—Set when the result is negative. For SPE compare and test instructions: Set if the high-order element of **r**A is equal to the high-order element of **r**B; cleared otherwise.
CR0[1]	33	00001	Positive (GT)—Set when the result is positive (and not zero). For SPE compare and test instructions: Set if the low-order element of **r**A is equal to the low-order element of **r**B; cleared otherwise.
CR0[2]	34	00010	Zero (EQ)—Set when the result is zero. For SPE compare and test instructions: Set to the OR of the result of the compare of the high and low elements.
CR0[3]	35	00011	Summary overflow (SO). Copy of XER[SO] at the instruction's completion. For SPE compare and test instructions: Set to the AND of the result of the compare of the high and low elements.
CR1[0]	36	00100	Negative (LT) For SPE and SPFP compare and test instructions: Set if the high-order element of **r**A is equal to the high-order element of **r**B; cleared otherwise.
CR1[1]	37	00101	Positive (GT) For SPE and SPFP compare and test instructions: Set if the low-order element of **r**A is equal to the low-order element of **r**B; cleared otherwise.
CR1[2]	38	00110	Zero (EQ) For SPE and SPFP compare and test instructions: Set to the OR of the result of the compare of the high and low elements.
CR1[3]	39	00111	Summary overflow (SO) For SPE and SPFP compare and test instructions: Set to the AND of the result of the compare of the high and low elements.
CR*n*[0]	40 44 48 52 56 60	01000 01100 10000 10100 11000 11100	Less than (LT). For integer compare instructions: **r**A < SIMM or **r**B (signed comparison) or **r**A < UIMM or **r**B (unsigned comparison). For SPE and SPFP compare and test instructions: Set if the high-order element of **r**A is equal to the high-order element of **r**B; cleared otherwise.
CR*n*[1]	41 45 49 53 57 61	01001 01101 10001 10101 11001 11101	Greater than (GT). For integer compare instructions: **r**A > SIMM or **r**B (signed comparison) or **r**A > UIMM or **r**B (unsigned comparison). For SPE and SPFP compare and test instructions: Set if the low-order element of **r**A is equal to the low-order element of **r**B; cleared otherwise.
CR*n*[2]	42 46 50 54 58 62	01010 01110 10010 10110 11010 11110	Equal (EQ). For integer compare instructions: **r**A = SIMM, UIMM, or **r**B. For SPE and SPFP compare and test instructions: Set to the OR of the result of the compare of the high and low elements.
CR*n*[3]	43 47 51 55 59 63	01011 01111 10011 10111 11011 11111	Summary overflow (SO). For integer compare instructions, this is a copy of XER[SO] at the completion of the instruction. For SPE and SPFP vector compare and test instructions: Set to the AND of the result of the compare of the high and low elements.

2.5.1.1 CR Setting for Integer Instructions

For all integer word instructions in which the Rc bit is defined and set, and for **addic.**, **andi.**, and **andis.**, CR0[32–34] are set by signed comparison of bits 32–63 of the result to zero; CR[35] is copied from the final state of XER[SO]. The Rc bit is not defined for double-word integer operations.

```
if      (target_register)32–63 < 0 then c ← 0b100
else if (target_register)32–63 > 0 then c ← 0b010
else                                    c ← 0b001
CR0 ← c || XERSO
```

The value of any undefined portion of the result is undefined, and the value placed into the first three bits of CR0 is undefined. CR0 bits are interpreted as described in Table 2-7.

Table 2-7. CR0 Field Descriptions

CR Bit	Name	Description
32	Negative (LT)	Bit 32 of the result is equal to 1.
33	Positive (GT)	Bit 32 of the result is equal to 0 and at least one of bits 33–63 of the result is non-zero.
34	Zero (EQ)	Bits 32–63 of the result are equal to 0.
35	Summary overflow (SO)	This is a copy of the final state of XER[SO] at the completion of the instruction.

Note that CR0 may not reflect the true (infinitely precise) result if overflow occurs.

2.5.1.2 CR Setting for Store Conditional Instructions

CR0 is also set by the integer store conditional instruction, **stwcx.**. See instruction descriptions in Chapter 3, “Instruction Model,” for detailed descriptions of how CR0 is set.

2.5.1.3 CR Setting for Compare Instructions

For compare instructions, a CR field specified by the BI field in the instruction is set to reflect the result of the comparison, as shown in Table 2-8.

Table 2-8. CR Setting for Compare Instructions

CR*n* Bit	Bit Expression	CR Bits		BI		Description
		AIM (BI Operand)	Book E	0–2	3–4	
CR*n*[0]	**4 * cr0 + lt** (or **lt**)	0	32	000	00	Less than (LT). For integer compare instructions: **r**A < SIMM or **r**B (signed comparison) or **r**A < UIMM or **r**B (unsigned comparison).
	4 * cr1 + lt	4	36	001		
	4 * cr2 + lt	8	40	010		
	4 * cr3 + lt	12	44	011		
	4 * cr4 + lt	16	48	100		
	4 * cr5 + lt	20	52	101		
	4 * cr6 + lt	24	56	110		
	4 * cr7 + lt	28	60	111		

Table 2-8. CR Setting for Compare Instructions (continued)

CR*n* Bit	Bit Expression	CR Bits		BI		Description
		AIM (BI Operand)	Book E	0–2	3–4	
CR*n*[1]	**4 * cr0 + gt** (or **gt**) **4 * cr1 + gt** **4 * cr2 + gt** **4 * cr3 + gt** **4 * cr4 + gt** **4 * cr5 + gt** **4 * cr6 + gt** **4 * cr7 + gt**	1 5 9 13 17 21 25 29	33 37 41 45 49 53 57 61	000 001 010 011 100 101 110 111	01	Greater than (GT). For integer compare instructions: rA > SIMM or rB (signed comparison) or **r**A > UIMM or **r**B (unsigned comparison).
CR*n*[2]	**4 * cr0 + eq** (or **eq**) **4 * cr1 + eq** **4 * cr2 + eq** **4 * cr3 + eq** **4 * cr4 + eq** **4 * cr5 + eq** **4 * cr6 + eq** **4 * cr7 + eq**	2 6 10 14 18 22 26 30	34 38 42 46 50 54 58 62	000 001 010 011 100 101 110 111	10	Equal (EQ). For integer compare instructions: **r**A = SIMM, UIMM, or **r**B.
CR*n*[3]	**4 * cr0 + so** (or **so**) **4 * cr1 + so** **4 * cr2 + so** **4 * cr3 + so** **4 * cr4 + so** **4 * cr5 + so** **4 * cr6 + so** **4 * cr7 + so**	3 7 11 15 19 23 27 31	35 39 43 47 51 55 59 63	000 001 010 011 100 101 110 111	11	Summary overflow (SO). For integer compare instructions, this is a copy of XER[SO] at instruction completion.

2.5.2 Link Register (LR)

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

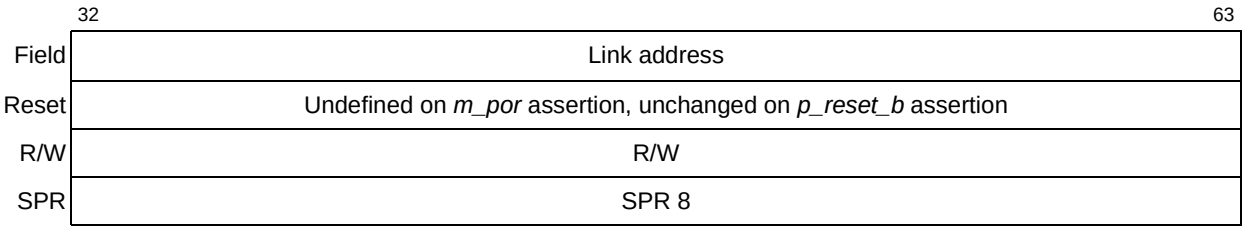


Figure 2-8. Link Register (LR)

LR contents are read into a GPR using **mfspr**. The contents of a GPR can be written to LR using **mtspr**. LR[62–63] are ignored by **bclr** instructions.

2.5.3 Count Register (CTR)

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

	32	63
Field	Count value	
Reset	Undefined on *m_por* assertion, unchanged on *p_reset_b* assertion	
R/W	R/W	
SPR	SPR 9	

Figure 2-9. Count Register (CTR)

2.6 SPE and SPFP APU Registers

The SPE and SPFP include the signal processing and embedded floating-point status and control register (SPEFSCR), described in Section 2.6.1, “Signal Processing/Embedded Floating-Point Status and Control Register (SPEFSCR).” The SPE implements a 64-bit accumulator, described in Section 2.6.2, “Accumulator (ACC).”

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

The SPEFSCR, shown in Figure 2-10, is used for status and control of SPE and SPFP instructions.

	High-Word Error Bits									Status Bits					
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
Field	SOVH	OVH	FGH	FXH	FINVH	FDBZH	FUNFH	FOVFH	—		FINXS	FINVS	FDBZS	FUNFS	FOVFS	MODE
Reset	0000_0000_0000_0000															
R/W	R/W															

	Enable Bits															
	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
Field	SOV	OV	FG	FX	FINV	FDBZ	FUNF	FOVF	—	FINXE	FINVE	FDBZE	FUNFE	FOVFE	FRMC	
Reset	0000_0000_0000_0000															
R/W	R/W															
SPR	SPR 512															

Figure 2-10. Signal Processing and Embedded Floating-Point Status and Control Register (SPEFSCR)

Table 2-9 describes SPEFSCR fields.

Table 2-9. SPEFSCR Field Descriptions

Bits	Name	Description
32	SOVH	Summary integer overflow high. Set whenever an instruction sets OVH and remains set until it is cleared by an **mtspr** specifying the SPEFSCR.
33	OVH	Integer overflow high. Set whenever an integer or fractional SPE instruction signals an overflow in the upper half of the result.
34	FGH	Embedded floating-point guard bit high. Supplied for use by the floating-point round exception handler. Zeroed if a floating-point data exception occurred for the high elements. FGH corresponds to the high element result. FGH is cleared by a scalar floating point instruction.
35	FXH	Embedded floating-point sticky bit high. Supplied for use by the floating-point round exception handler. Zeroed if a floating-point data exception occurred for the high elements. FXH corresponds to the high element result. FXH is cleared by a scalar floating point instruction.
36	FINVH	Embedded floating-point invalid operation/input error high. In mode 0, set 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, FINVH is set on an IEEE754 invalid operation (IEEE754-1985 sec7.1) in the high element. Cleared by a scalar floating point instruction.
37	FDBZH	Embedded floating-point divide by zero high. 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. Cleared by a scalar floating point instruction.
38	FUNFH	Embedded floating-point underflow high. Set 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.
39	FOVFH	Embedded floating-point overflow high. Set when the execution of a floating-point instruction results in an overflow in the high element. Cleared by a scalar floating point instruction.
40–41	—	Reserved, should be cleared.
42	FINXS	Embedded floating-point inexact sticky flag. Set 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. FINXS remains set until it is cleared by an **mtspr** specifying SPEFSCR.
43	FINVS	Embedded floating-point invalid operation sticky flag. Set when a floating-point instruction sets FINVH or FINV. FINVS remains set until it is cleared by an **mtspr** instruction specifying SPEFSCR.
44	FDBZS	Embedded floating-point divide by zero sticky flag. Set when a floating-point divide instruction sets FDBZH or FDBZ. FDBZS remains set until it is cleared by an **mtspr** specifying SPEFSCR.
45	FUNFS	Embedded floating-point underflow sticky flag. Set when a floating-point instruction sets FUNFH or FUNF. FUNFS remains set until it is cleared by an **mtspr** specifying SPEFSCR.
46	FOVFS	Embedded floating-point overflow sticky flag. Set when a floating-point instruction sets FOVFH or FOVF. FOVFS remains set until it is cleared by an **mtspr** specifying SPEFSCR.

Table 2-9. SPEFSCR Field Descriptions (continued)

Bits	Name	Description
47	MODE	Embedded floating-point operating mode. 0 Default hardware results operating mode. The e200z6 supports only mode 0. 1 IEEE754 hardware results operating mode (not supported by the e200z6) Controls the operating mode of the embedded floating-point APU. 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 indicates the hardware supported mode.
48	SOV	Summary integer overflow. Set whenever an instruction sets OV. SOV remains set until it is cleared by an **mtspr** specifying SPEFSCR.
49	OV	Integer overflow. Set whenever an integer or fractional SPE instruction signals an overflow in the low element result.
50	FG	Embedded floating-point guard bit. Used by the floating-point round exception handler. Zeroed if a floating-point data exception occurs for the low elements. Corresponds to the low element result.
51	FX	Embedded floating-point sticky bit. Supplied for use by the floating-point round exception handler.FX is zeroed if a floating-point data exception occurs for the low elements. FX corresponds to the low element result.
52	FINV	Embedded floating-point invalid operation/input error. In mode 0, FINV is set 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, FINV is set on an IEEE754 invalid operation (IEEE754-1985 sec7.1) in the low element.
53	FDBZ	Embedded floating-point divide by zero. Set 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.
54	FUNF	Embedded floating-point underflow. Set when the execution of a floating-point instruction results in an underflow in the low element.
55	FOVF	Embedded floating-point overflow. Set to 1 when the execution of a floating-point instruction results in an overflow in the low element.
56	—	Reserved, should be cleared.
57	FINXE	Embedded floating-point inexact exception enable. 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. 0 Exception disabled 1 Exception enabled
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 FINV or FINVH is set by a floating-point instruction.
59	FDBZE	Embedded floating-point divide by zero exception enable 0 Exception disabled 1 Exception enabled. If the exception is enabled, a floating-point data exception is taken if FDBZ or FDBZH is set by a floating-point instruction.

Table 2-9. SPEFSCR Field Descriptions (continued)

Bits	Name	Description
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 FUNF or FUNFH is set by a floating-point instruction.
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 FOVF or FOVFH is set by a floating-point instruction.
62–63	FRMC	Embedded floating-point rounding mode control 00 Round to nearest 01 Round toward zero 10 Round toward +infinity 11 Round toward -infinity

2.6.2 Accumulator (ACC)

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

The Initialize Accumulator instruction (**evmra**) is provided to initialize the accumulator. This instruction is described in the EREF.

2.7 Interrupt Registers

Section 2.7.1, “Interrupt Registers Defined by Book E,” and Section 2.7.2, “e200z6-Specific Interrupt Registers,” describe registers used for interrupt handling.

2.7.1 Interrupt Registers Defined by Book E

This section describes the following registers and their fields:

- Section 2.7.1.1, “Save/Restore Register 0 (SRR0)”
- Section 2.7.1.2, “Save/Restore Register 1 (SRR1)”
- Section 2.7.1.3, “Critical Save/Restore Register 0 (CSRR0)”
- Section 2.7.1.4, “Critical Save/Restore Register 1 (CSRR1)”
- Section 2.7.1.5, “Data Exception Address Register (DEAR)”

- Section 2.7.1.6, “Interrupt Vector Prefix Register (IVPR)
- Section 2.7.1.7, “Interrupt Vector Offset Registers (IVORs)
- Section 2.7.1.8, “Exception Syndrome Register (ESR)

2.7.1.1 Save/Restore Register 0 (SRR0)

On a non-critical interrupt, SRR0, shown in Figure 2-11, holds the address of the instruction where the interrupted process should resume. The instruction is interrupt specific, although for instruction-caused exceptions, it is typically the address of the instruction that caused the interrupt. When **rfi** executes, instruction execution continues at the address in SRR0. SRR0 and SRR1 are not affected by **rfci** or **rfdi**.

	32	63
Field	Next instruction address	
Reset	Undefined on *m_por* assertion, unchanged on *p_reset_b* assertion	
R/W	R/W	
SPR	SPR 26	

Figure 2-11. Save/Restore Register 0 (SRR0)

2.7.1.2 Save/Restore Register 1 (SRR1)

SRR1, shown in Figure 2-12, is provided to save and restore machine state on non-critical interrupts. When a non-critical interrupt is taken, MSR contents are placed into SRR1. When **rfi** executes, the contents of SRR1 are restored into MSR. SRR1 bits that correspond to reserved MSR bits are also reserved. (See Section 2.3.1, “Machine State Register (MSR),” for more information.) SRR0 and SRR1 are not affected by **rfci** or **rfdi**. Reserved MSR bits may be altered by **rfi**, **rfci**, or **rfdi**.

	32	63
Field	MSR state information	
Reset	Undefined on *m_por* assertion, unchanged on *p_reset_b* assertion	
R/W	R/W	
SPR	SPR 27	

Figure 2-12. Save/Restore Register 1 (SRR1)

2.7.1.3 Critical Save/Restore Register 0 (CSRR0)

CSRR0 is provided to save and restore machine state on critical interrupts. It is used by critical interrupts in the same way SRR0 is used for non-critical interrupts: to hold the address of the instruction to which control is passed at the end of the interrupt handler.

On a critical interrupt, CSRR0, shown in Figure 2-13, holds the address of the instruction where the interrupted process should resume. The instruction is interrupt specific; consult Chapter 5, “Interrupts and Exceptions,” for more information. When **rfci** executes, instruction execution continues at the address in CSRR0. CSRR0 and CSRR1 are not affected by **rfi** or **rfdi**.

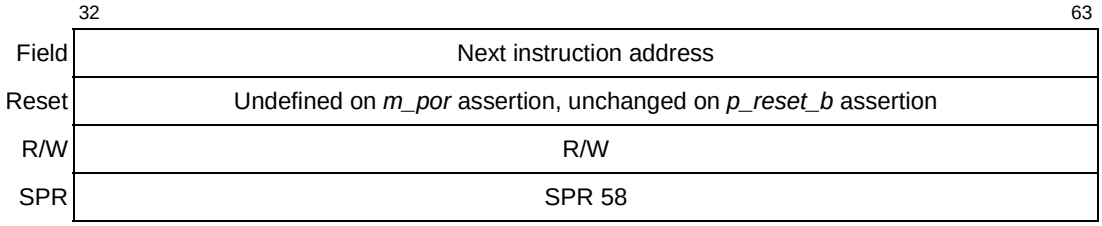


Figure 2-13. Critical Save/Restore Register 0 (CSRR0)

2.7.1.4 Critical Save/Restore Register 1 (CSRR1)

CSRR1 (Figure 2-14) is provided to save and restore machine state on critical interrupts. When a critical interrupt is taken, MSR contents are placed into CSRR1. When **rfci** executes, the contents of CSRR1 are restored into MSR. CSRR1 bits that correspond to reserved MSR bits are also reserved. (See Section 2.3.1, “Machine State Register (MSR),” for more information.) CSRR0 and CSRR1 are not affected by **rfi** or **rfdi**. Reserved MSR bits may be altered by **rfi**, **rfci**, or **rfdi**.

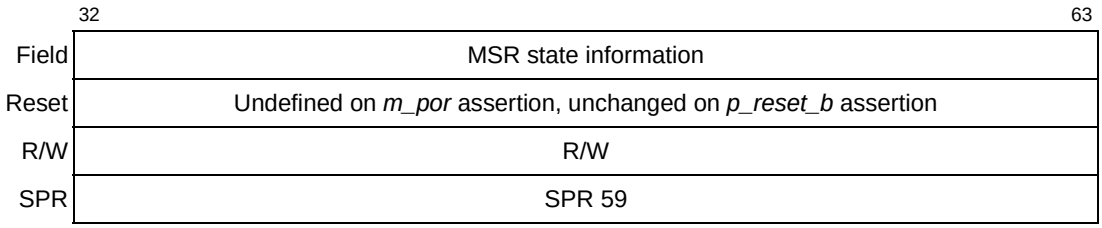


Figure 2-14. Critical Save/Restore Register 1 (CSRR1)

2.7.1.5 Data Exception Address Register (DEAR)

DEAR, shown in Figure 2-15, is loaded with the effective address of a data access (caused by a load, store, or cache management instruction) that results in an alignment, data TLB miss, or DSI exception. The DEAR register can be read or written using the **mfspr** and **mtspr** instructions.

	32	63
Field	Exception address	
Reset	Undefined on *m_por* assertion, unchanged on *p_reset_b* assertion	
R/W	R/W	
SPR	SPR 61	

Figure 2-15. Data Exception Address Register (DEAR)

2.7.1.6 Interrupt Vector Prefix Register (IVPR)

The IVPR, shown in Figure 2-16, is used during interrupt processing for determining the starting address for the software interrupt handler. The value contained in the vector offset field of the IVOR selected for a particular interrupt type is concatenated with the value in the IVPR to form an instruction address from which execution is to begin.

	32	47	48	63
Field	Vector Base		—	
Reset	Undefined on *m_por* assertion, unchanged on *p_reset_b* assertion			
R/W	R/W			
SPR	SPR 63			

Figure 2-16. Interrupt Vector Prefix Register (IVPR)

IVPR fields are defined in Table 2-10.

Table 2-10. IVPR Field Descriptions

Bits	Name	Description
32–47	Vector Base	Used to define the base location of the vector table, aligned to a 64-Kbyte boundary. This field provides the high-order 16 bits of the location of all interrupt handlers. The contents of the IVOR*n* appropriate for the type of exception being processed are concatenated with the IVPR vector base to form the address of the handler in memory.
48–63	—	Reserved, should be cleared.

2.7.1.7 Interrupt Vector Offset Registers (IVORs)

IVORs, shown in Figure 2-17, hold the quad-word index from the base address provided by the IVPR for each interrupt type.

	32 — 47	48 — 59	60	61 — 63
Field	—	Vector offset	—	CS
Reset	Undefined on *m_por* assertion, unchanged on *p_reset_b* assertion			
R/W	R/W			
SPR	(See Table 2-12.)			

Figure 2-17. Interrupt Vector Offset Registers (IVOR)

The IVOR fields are defined in Table 2-11.

Table 2-11. IVOR Field Descriptions

Bits	Name	Setting Description
32–47	—	Reserved, should be cleared.
48–59	Vector offset	Provides a quad-word index from the base address provided by the IVPR to locate an interrupt handler.
60	—	Reserved, should be cleared.
61–63	CS	Context selector (e200z6-specific). When multiple hardware contexts are supported, this field is used to select an operating context for the interrupt handler. This value is loaded into the CURCTX field of the context control register (CTXCR) as part of the interrupt vectoring process. When multiple hardware contexts are not supported, CS is not implemented and is read as zero.

SPR numbers corresponding to IVOR16–IVOR31 are reserved. IVOR32–IVOR47 and IVOR60–IVOR63 are reserved. SPR numbers for IVOR32–IVOR63 are allocated for implementation-dependent use (IVOR32–IVOR34 (SPR 528–530) are defined by the EIS). IVOR assignments are shown in Table 2-12.

Table 2-12. IVOR Assignments

IVOR Number	SPR	Interrupt Type
IVOR0	400	Critical input
IVOR1	401	Machine check
IVOR2	402	Data storage
IVOR3	403	Instruction storage
IVOR4	404	External input
IVOR5	405	Alignment
IVOR6	406	Program
IVOR7	407	Floating-point unavailable
IVOR8	408	System call
IVOR9	409	Auxiliary processor unavailable. This interrupt is defined by the EIS but not supported in the e200z6.
IVOR10	410	Decrementer

Table 2-12. IVOR Assignments (continued)

IVOR Number	SPR	Interrupt Type
IVOR11	411	Fixed-interval timer interrupt
IVOR12	412	Watchdog timer interrupt
IVOR13	413	Data TLB error
IVOR14	414	Instruction TLB error
IVOR15	415	Debug
IVOR16–IVOR31	—	Reserved for future architectural use
IVOR32	528	SPE APU unavailable (EIS–defined)
IVOR33	529	SPE floating-point data exception (EIS–defined)
IVOR34	530	SPE floating-point round exception (EIS–defined)
IVOR35–IVOR63	—	Allocated for implementation-dependent use

2.7.1.8 Exception Syndrome Register (ESR)

The exception syndrome register (ESR) provides a syndrome to distinguish exceptions that can generate the same interrupt type. The e200z6 adds some implementation-specific bits to this register, as shown in Figure 2-18.

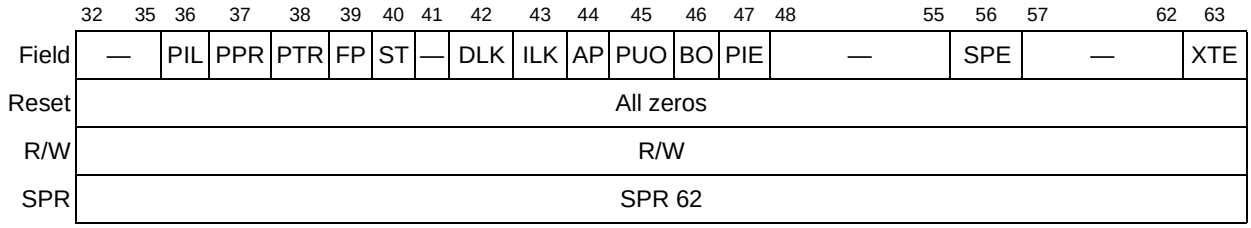


Figure 2-18. Exception Syndrome Register (ESR)

NOTE

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

The ESR fields are described in Table 2-13.

Table 2-13. ESR Field Descriptions

Bit(s)	Name	Description	Associated Interrupt Type
32–35	—	Reserved, should be cleared.	—
36	PIL	Illegal instruction exception	Program
37	PPR	Privileged instruction exception	Program
38	PTR	Trap exception	Program
39	FP	Floating-point operation	Alignment, data storage, data TLB, program
40	ST	Store operation	Alignment, data storage, data TLB
41	—	Reserved, should be cleared.	—
42	DLK	Data cache locking	Data storage
43	ILK	Instruction cache locking	Data storage`
44	AP	Auxiliary processor operation. (unused in the e200z6)	Alignment, data storage, data TLB, program
45	PUO	Unimplemented operation exception	Program
46	BO	Byte ordering exception	Data storage
47	PIE	Program imprecise exception—Unused in the e200z6 (Reserved, should be cleared.)	—
48–55	—	Reserved, should be cleared.	—
56	SPE	SPE APU operation	SPE unavailable, SPE floating-point data exception, SPE floating-point round exception, alignment, data storage, data TLB
57–62	—	Reserved, should be cleared.	—
63	XTE	External termination error (precise)	Data storage, instruction storage

2.7.2 e200z6-Specific Interrupt Registers

In addition to the Book E–defined interrupt registers, the e200z6 implements additional registers. DSRR0 and DSRR1 (see Section 2.7.2.1, “Debug Save/Restore Register 0 (DSRR0)” and Section 2.7.2.2, “Debug Save/Restore Register 1 (DSRR1)”) are provided to facilitate handling debug interrupts, and the EIS-defined MCSR (see Section 2.7.2.3, “Machine Check Syndrome Register (MCSR)”) is provided to aid in handling machine check interrupts.

2.7.2.1 Debug Save/Restore Register 0 (DSRR0)

On a debug interrupt, DSRR0, shown in Figure 2-19, holds the address of the instruction where the interrupted process should resume. The instruction is interrupt specific; see Section 5.6.16, “Debug Interrupt (IVOR15),” and particularly Table 5-23, for more

information. When **rfdi** executes, instruction execution continues at the address in DSRR0. DSRR0 and DSRR1 are not affected by **rfi** or **rfci**.

	32	63
Field	Next instruction address	
Reset	Undefined on *m_por* assertion, unchanged on *p_reset_b* assertion	
R/W	R/W	
SPR	SPR 574	

Figure 2-19. Debug Save/Restore Register 0 (DSRR0)

2.7.2.2 Debug Save/Restore Register 1 (DSRR1)

DSRR1 (Figure 2-20) is provided to save and restore machine state on debug interrupts. When a debug interrupt is taken, MSR contents are placed into DSRR1. When **rfdi** executes, the contents of DSRR1 are restored into MSR. DSRR1 bits that correspond to reserved MSR bits are also reserved. (See Section 2.3.1, “Machine State Register (MSR),” for more information.) DSRR0 and DSRR1 are not affected by **rfi** or **rfci**. Reserved MSR bits may be altered by **rfi**, **rfci**, or **rfdi**.

	32	63
Field	MSR state information	
Reset	Undefined on *m_por* assertion, unchanged on *p_reset_b* assertion	
R/W	R/W	
SPR	SPR 575	

Figure 2-20. Debug Save/Restore Register 1 (DSRR1)

2.7.2.3 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 2-21.

	32	33	34	35	36	37	60	61	62	63
Field	MCP	—	CP_PERR	CPERR	EXCP_ERR	—		BUS_WRERR	—	
Reset	All zeros									
R/W	R/W									
SPR	SPR 572									

Figure 2-21. Machine Check Syndrome Register (MCSR)

Table 2-14 describes MCSR fields. The MCSR indicates the source of a machine check condition is recoverable. When a syndrome bit in the MCSR is set, the core complex asserts *p_mcp_out* for system information.

Table 2-14. MCSR Field Descriptions

Bits	Name	Description	Recoverable
32	MCP	Machine check input signal	Maybe
33	—	Reserved, should be cleared.	—
34	CP_PERR	Cache push parity error	Unlikely
35	CPERR	Cache parity error	Precise
36	EXCP_ERR	ISI, ITLB, or bus error on first instruction fetch for an exception handler	Precise
37–60	—	Reserved, should be cleared.	—
61	BUS_WRERR	Write bus error on buffered store or cache line push	Unlikely
62–63	—	Reserved, should be cleared.	—

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

Software-use SPRs (SPRG0–SPRG7 and USPRG0, shown in Figure 2-22) have no defined functionality. These are as follows:

- SPRG0–SPRG2—These registers can be accessed only in supervisor mode.
- SPRG3—This register can be written only in supervisor mode. It is readable in supervisor mode, but it is implementation dependent whether it can be read in user mode. It is not readable in user mode on the e200z6.
- SPRG4–SPRG7—These registers can be written only in supervisor mode. They are readable in supervisor or user mode.
- USPRG0—This register can be accessed in supervisor or user mode.

	32				63
Field	Software-determined information				
Reset	Undefined on *m_por* assertion, unchanged on *p_reset_b* assertion				
SPR R/W	SPRG0	272	Read/Write	Supervisor	
	SPRG1	273	Read/Write	Supervisor	
	SPRG2	274	Read/Write	Supervisor	
	SPRG3	259	Read only	User [1]/Supervisor	
		275	Read/Write	Supervisor	
	SPRG4	260	Read only	User/Supervisor	
		276	Read/Write	Supervisor	
	SPRG5	261	Read only	User/Supervisor	
		277	Read/Write	Supervisor	
	SPRG6	262	Read only	User/Supervisor	
		278	Read/Write	Supervisor	
	SPRG7	263	Read only	User/Supervisor	
		279	Read/Write	Supervisor	
	USPRG0	256	Read/Write	User/Supervisor	

Figure 2-22. Software-Use SPRs (SPRG0–SPRG7 and USPRG0)

[1] User-mode access to SPRG3 is defined by Book E as implementation dependent. It is not supported in the e200z6.

Software-use SPRs are read into a GPR by using **mfspr** and are written by using **mtspr**.

2.9 Timer Registers

The time base (TB), decrementer (DEC), fixed-interval timer (FIT), and watchdog timer provide timing functions for the system. The relationship of these timer facilities to each other is shown in Figure 2-23 and are described as follows:

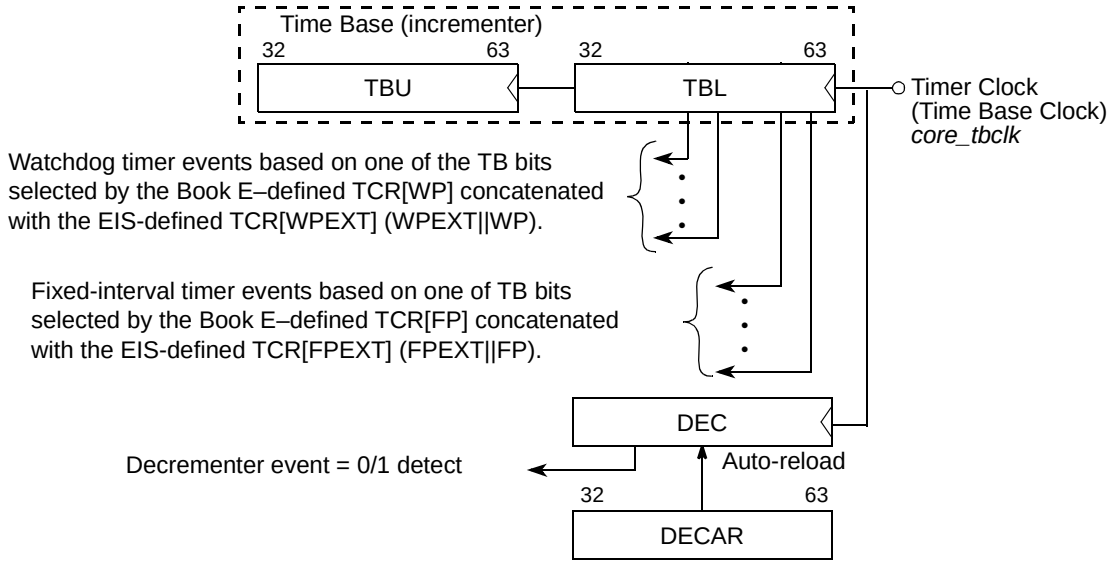


Figure 2-23. Relationship of Timer Facilities to the Time Base

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

  The DEC is typically used as a general-purpose software timer.
- The time base for the TB and DEC is selected by the time base enable (TBEN) and select time base clock (SEL_TBCLK) bits in HID0, as follows:
  - — If HID0[TBEN] = 1 and HID0[SEL_TBCLK] = 0, the time base and decrementer are based on processor clock.
  - — If HID0[TBEN] = 1 and HID0[SEL_TBCLK] = 1, the time base and decrementer are based on the *p_tbclk* input.
- Software can select one from of four TB bits to signal a fixed-interval interrupt whenever the bit transitions from 0 to 1. It is typically used to trigger periodic system maintenance functions. Bits that may be selected are implementation-dependent.
- The watchdog timer. also a selected TB bit, provides a way to signal a critical exception when the selected bit transitions from 0 to 1. It is typically used for system

error recovery. If software does not respond in time to the initial interrupt by clearing the associated status bits in the TSR before the next expiration of the watchdog timer interval, a watchdog timer-generated processor reset may result, if so enabled.

All timer facilities must be initialized during start-up.

2.9.1 Timer Control Register (TCR)

The TCR, shown in Figure 2-24, provides control information for the CPU timer facilities. The EREF provides a detailed description of the TCR. TCR[WRC] functions are implementation dependent. In addition, the e200z6 core implements two implementation-specific fields, TCR[WPEXT] and TCR[FPEXT].

	32 33	34 35	36	37	38 39	40	41	42	43 46	47 50	51 63
Field	WP	WRC	WIE	DIE	FP	FIE	ARE	—	WPEXT	FPEXT	—
Reset	All zeros										
R/W	R/W										
SPR	SPR 340										

Figure 2-24. Timer Control Register (TCR)

The TCR fields are described in Table 2-15.

Table 2-15. TCR Field Descriptions

Bits	Name	Description
32–33	WP	Watchdog timer period, When concatenated with WPEXT, specifies one of 64 bit locations of the time base used to signal a watchdog timer exception on a transition from 0 to 1. See Table 2-16. TCR[WPEXT]||TCR[WP] == 000000 selects TBU[32] (msb of TBU). TCR[WPEXT]||TCR[WP] == 111111 selects TBL[63] (lsb of TBL).
34–35	WRC	Watchdog timer reset control. WRC may be set by software, but cannot be cleared by software (except by a software-induced reset). Once written to a non-zero value, WRC may no longer be altered by software. 00 No watchdog timer reset can occur 01 Force processor checkstop on second time-out of the watchdog timer 10 Assert processor reset output (*p_resetout_b*) on second time-out of watchdog timer 11 Reserved
36	WIE	Watchdog timer interrupt enable 0 Watchdog timer interrupts disabled 1 Watchdog timer interrupts enabled
37	DIE	Decrementer interrupt enable 0 Decrementer interrupts disabled 1 Decrementer interrupts enabled
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. See Table 2-16. TCR[FPEXT]||TCR[FP] == 000000 selects TBU[32] (msb of TBU). TCRFP[EXT]||TCR[FP] == 111111 selects TBL[63] (lsb of TBL).

Table 2-15. TCR Field Descriptions (continued)

Bits	Name	Description
40	FIE	Fixed-interval interrupt enable 0 Fixed-interval interrupts disabled 1 Fixed-interval interrupts enabled
41	ARE	Auto-reload enable. Controls whether the value in DECAR is reloaded into DEC when the DEC value reaches 0000_0001. 0 Auto-reload disabled 1 Auto-reload enabled
42	—	Reserved, should be cleared.
43–46	WPEXT	Watchdog timer period extension (see above description for WP). WPEXT | WP select one of the 64 TB bits used to signal a watchdog timer exception.
47–50	FPEXT	Fixed-interval timer period extension (see description for FP). FPEXT | FP select one of the 64 TB bits used to signal a fixed-interval timer exception.
51–63	—	Reserved, should be cleared.

Table 2-16 shows how the concatenations of FPEXT | FP and WPEXT | WP can select any of the 64 bits in the time base register and the resulting timeout periods at 80 MHz.

Table 2-16. Timeout Period Selection (at 80 MHz)

FPEXT WPEXT	FP WP	TB Bit	Number of Clocks/Timeout	Timeout at 80 MHz (Secs)	FPEXT WPEXT	FP WP	TB Bit	Number of Clocks/Timeout	Timeout at 80 MHz (Secs)
0000	00	0	1.84467E+19	2.30584E+11	1000	00	32	4294967296	53.6870912
0000	01	1	9.22337E+18	1.15292E+11	1000	01	33	2147483648	26.8435456
0000	10	2	4.61169E+18	57646075230	1000	10	34	1073741824	13.4217728
0000	11	3	2.30584E+18	28823037615	1000	11	35	536870912	6.7108864
0001	00	4	1.15292E+18	14411518808	1001	00	36	268435456	3.3554432
0001	01	5	5.76461E+17	7205759404	1001	01	37	134217728	1.6777216
0001	10	6	2.8823E+17	3602879702	1001	10	38	67108864	0.8388608
0001	11	7	1.44115E+17	1801439851	1001	11	39	33554432	0.4194304
0010	00	8	7.20576E+16	900719925.5	1010	00	40	16777216	0.2097152
0010	01	9	3.60288E+16	450359962.7	1010	01	41	8388608	0.1048576
0010	10	10	1.80144E+16	225179981.4	1010	10	42	4194304	0.0524288
0010	11	11	9.0072E+15	112589990.7	1010	11	43	2097152	0.0262144
0011	00	12	4.5036E+15	56294995.34	1011	00	44	1048576	0.0131072
0011	01	13	2.2518E+15	28147497.67	1011	01	45	524288	0.0065536
0011	10	14	1.1259E+15	14073748.84	1011	10	46	262144	0.0032768
0011	11	15	5.6295E+14	7036874.418	1011	11	47	131072	0.0016384
0100	00	16	2.81475E+14	3518437.209	1100	00	48	65536	0.0008192

Table 2-16. Timeout Period Selection (at 80 MHz) (continued)

FPEXT WPEXT	FP WP	TB Bit	Number of Clocks/Timeout	Timeout at 80 MHz (Secs)	FPEXT WPEXT	FP WP	TB Bit	Number of Clocks/Timeout	Timeout at 80 MHz (Secs)
0100	01	17	1.40737E+14	1759218.604	1100	01	49	32768	0.0004096
0100	10	18	7.03687E+13	879609.3022	1100	10	50	16384	0.0002048
0100	11	19	3.51844E+13	439804.6511	1100	11	51	8192	0.0001024
0101	00	20	1.75922E+13	219902.3256	1101	00	52	4096	0.0000512
0101	01	21	8.79609E+12	109951.1628	1101	01	53	2048	0.0000256
0101	10	22	4.39805E+12	54975.58139	1101	10	54	1024	0.0000128
0101	11	23	2.19902E+12	27487.79069	1101	11	55	512	0.0000064
0110	00	24	1.09951E+12	13743.89535	1110	00	56	256	0.0000032
0110	01	25	5.49756E+11	6871.947674	1110	01	57	128	0.0000016
0110	10	26	2.74878E+11	3435.973837	1110	10	58	64	0.0000008
0110	11	27	1.37439E+11	1717.986918	1110	11	59	32	0.0000004
0111	00	28	68719476736	858.9934592	1111	00	60	16	0.0000002
0111	01	29	34359738368	429.4967296	1111	01	61	8	0.0000001
0111	10	30	17179869184	214.7483648	1111	10	62	4	0.00000005
0111	11	31	8589934592	107.3741824	1111	11	63	2	0.000000025

2.9.2 Timer Status Register (TSR)

The timer status register (TSR) provides status information for the CPU timer facilities. For more information about TSR, refer to the EREF. The TSR[WRS] field is defined to be implementation-dependent and is described below. The TSR is shown in Figure 2-25.

NOTE

Register fields designated as write-1-to-clear are cleared only by writing ones to them. Writing zeros to them has no effect.

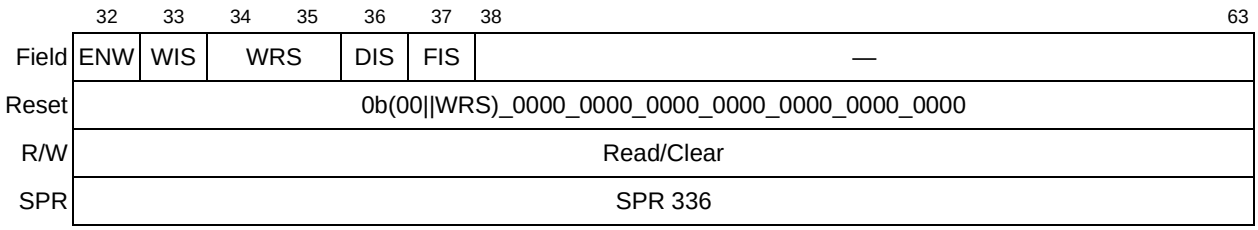


Figure 2-25. Timer Status Register (TSR)

The TSR fields are described in Table 2-17.

Table 2-17. Timer Status Register Field Descriptions

Bits	Name	Description
32	ENW	Enable next watchdog time. When a watchdog timer time-out occurs while WIS = 0 and the next watchdog time-out is enabled (ENW = 1), a watchdog timer exception is generated and logged by setting WIS. This is referred to as a watchdog timer first time out. A watchdog timer interrupt occurs if enabled by TCR[WIE] and MSR[CE]. To avoid another watchdog timer interrupt once MSR[CE] is reenabled (assuming TCR[WIE] is not cleared instead), the interrupt handler must reset TSR[WIS] by executing an **mtspr**, setting WIS and any other bits to be cleared and a 0 in all other bits. The data written to the TSR is not direct data, but is a mask. A 1 causes the bit to be cleared; a 0 has no effect. 0 Action on next watchdog timer time-out is to set TSR[ENW]. 1 Action on next watchdog timer time-out is governed by TSR[WIS].
33	WIS	Watchdog timer interrupt status. See the ENW description for more information about how WIS is used. 0 A watchdog timer event has not occurred. 1 A watchdog timer event occurred. When MSR[CE] = 1 and TCR[WIE] = 1, a watchdog timer interrupt is taken.
34–35	WRS	Watchdog timer reset status 00 No second timeout of watchdog timer has occurred 01 Force processor checkstop on second timeout of watchdog timer has occurred 10 Assert processor reset output (*p_resetout_b*) on second timeout of watchdog timer has occurred 11 Reserved
36	DIS	Decrementer interrupt status. 0 A decrementer event has not occurred. 1 A decrementer event occurred. When MSR[EE] = TCR[DIE] = 1, a decrementer interrupt is taken.
37	FIS	Fixed-interval timer interrupt status. 0 A fixed-interval timer event has not occurred. 1 A fixed-interval timer event occurred. When MSR[EE] = 1 and TCR[FIE] = 1, a fixed-interval timer interrupt is taken.
38–63	—	Reserved, should be cleared.

NOTE

The TSR can be read using **mfspr r**D**,TSR**. The TSR cannot be directly written to. Instead, TSR bits corresponding to 1 bits in GPR(**r**S) can be cleared using **mtspr TSR,r**S.

2.9.3 Time Base (TBU and TBL)

The time base (TB), seen in Figure 2-26, is composed of two 32-bit registers, the time base upper (TBU) concatenated on the right with the time base lower (TBL). The time base registers provide timing functions for the system and are enabled by setting HID0[TBEN]. The decrementer (DEC) updates at the same frequency, which is selected in HID0[SEL_TBCLK]. TB is a volatile resource and must be initialized during start-up.

For more information, see Section 2.9, “Timer Registers.”

	32 63	32 63
Field	TBU	TBL
Reset	Undefined on *m_por* assertion, unchanged on *p_reset_b* assertion	Undefined on *m_por* assertion, unchanged on *p_reset_b* assertion
R/W	User read/Supervisor write	User read/Supervisor write
SPR	269 Read/285 Write	268 Read/284 Write

Figure 2-26. Time Base Upper/Lower Registers (TBU/TBL)

The TB is interpreted as a 64-bit unsigned integer that is incremented periodically. Each increment adds 1 to the least-significant bit. The frequency at which the integer is updated is implementation-dependent.

TBL increments until its value becomes 0xFFFF_FFFF ($2^{32} - 1$). At the next increment, its value becomes 0x0000_0000 and TBU is incremented. This process continues until the TBU value becomes 0xFFFF_FFFF and the TBL value becomes 0xFFFF_FFFF (TB is interpreted as 0xFFFF_FFFF_FFFF_FFFF ($2^{64} - 1$)). At the next increment, the TBU value becomes 0x0000_0000 and the TBL value becomes 0x0000_0000. There is no interrupt (or any other indication) when this occurs.

The period depends on the driving frequency. For example, if TB is driven by 100 MHz divided by 32, the TB period is as follows:

$$T_{TB} = 2^{64} \times \frac{32}{100\ \text{MHz}} = 5.90 \times 10^{12}\ \text{seconds}\ \text{(approximately 187,000 years)}$$

The TB is implemented such that the following requirements are satisfied.

- Loading a GPR from the TB has no effect on the accuracy of the TB.
- Storing a GPR to the TB replaces the value in the TB with the value in the GPR.

Book E does not specify a relationship between the frequency at which the TB is updated and other frequencies, such as the CPU clock or bus clock in a Book E system. The TB update frequency is not required to be constant. One of the following is required to ensure that system software can keep time of day and operate interval timers:

- The system provides an (implementation-dependent) interrupt to software whenever the update frequency of the TB changes and a way to determine the current update frequency.
- The update frequency of the TB is under the control of system software.

NOTE

Disabling the TB or making reading the time base privileged prevents the TB from being used to implement a covert channel in a secure system.

NOTE

If the operating system initializes the TB on power-on to some reasonable value and the update frequency of the TB is constant, the TB can be used as a source of values that increase at a constant rate, such as for time stamps in trace entries.

Even if the update frequency is not constant, values read from the TB are monotonically increasing (except when the TB wraps from $2^{64} - 1$ to 0). If a trace entry is recorded each time the update frequency changes, the sequence of TB values can be post-processed to become actual time values.

Successive readings of the TB may return identical values.

It is intended that the TB be useful for timing reasonably short sequences of code (a few hundred instructions) and for low-overhead time stamps for tracing.

2.9.4 Decrementer Register

The 32-bit decrementer (DEC), shown in Figure 2-27, is a decrementing counter that is enabled by setting HID0[TBEN]. The decrementer and time base update at the same frequency, which is selected in HID0[SEL_TBCLK]. It provides way to signal a decrementer interrupt after a specified period unless one of the following occurs:

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

For more information, see Section 2.9, “Timer Registers.”

DEC is typically used as a general-purpose software timer. The decrementer auto-reload register, is used to automatically reload a programmed value into DEC, as described in Section 2.9.5, “Decrementer Auto-Reload Register (DECAR).”

	32	63
Field	Decrementer value	
Reset	Undefined on *m_por* assertion, unchanged on *p_reset_b* assertion	
R/W	R/W	
SPR	SPR 22	

Figure 2-27. Decrementer Register (DEC)

2.9.5 Decrementer Auto-Reload Register (DECAR)

The decrementer auto-reload register is shown in Figure 2-28. If the auto-reload function is enabled (TCR[ARE] = 1), the auto-reload value in DECAR is written to DEC when DEC

decrements from 0x0000_0001 to 0x0000_0000. Note that writing DEC with zeros by using an **mtspr[DEC]** does not automatically generate a decrementer exception.

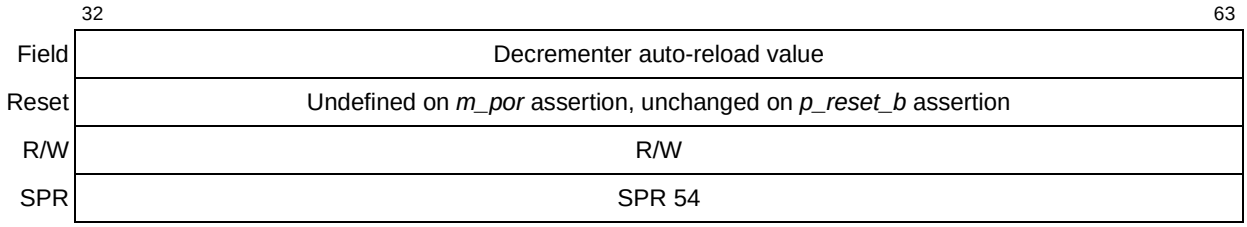


Figure 2-28. Decrementer Auto-Reload Register (DECAR)

2.10 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 by software is conditioned by the external debug mode control bit (DBCR0[EDM]) which can be set by the hardware debug port. If DBCR0[EDM] is set, software is prevented from modifying debug register values. Execution of an **mtspr** instruction targeting a debug register will not cause modifications to occur. In addition, since the external debugger hardware may be manipulating debug register values, the state of these registers is not guaranteed to be consistent if read by software with an **mfspr** instruction, other than DBCR0[EDM] itself.

2.10.1 Debug Address and Value Registers

Instruction address compare registers IAC1–IAC4 are used to hold instruction addresses for comparison purposes. In addition, IAC2 and IAC4 hold mask information for IAC1 and IAC3 respectively when address bit match compare modes are selected.

NOTE

When performing instruction address compares, the low order two address bits of the instruction address and the corresponding IAC register are ignored.

Data address compare registers DAC1 and DAC2 are used to hold data access addresses for comparison purposes. In addition, DAC2 holds mask information for DAC1 when address bit match compare mode is selected.

2.10.1.1 Instruction Address Compare Registers (IAC1–IAC4)

The instruction address compare registers (IAC1–IAC4) are each 32 bits, with bits 62–63 being reserved, as shown in Figure 2-29.

	32 … 61	62	63
Field	Instruction address	—	
Reset	All zeros		
R/W	R/W		
SPR	SPR 312 (IAC1); SPR 313 (IAC2); SPR 314 (IAC3); SPR 315 (IAC4)		

Figure 2-29. Instruction Address Compare Registers (IAC1–IAC4)

A debug event may be enabled to occur upon an attempt to execute an instruction from an address specified in an IAC, inside or outside a range specified by IAC1 and IAC2 or, inside or outside a range specified by IAC3 and IAC4, or to blocks of addresses specified by the combination of the IAC1 and IAC2, or to blocks of addresses specified by the combination of the IAC3 and IAC4. Because all instruction addresses are required to be word-aligned, the two low-order bits of the IACs are reserved and do not participate in the comparison to the instruction address.

2.10.1.2 Data Address Compare Registers (DAC1–DAC2)

The data address compare 1 register (DAC1) and data address compare 2 register (DAC2), shown in Figure 2-30, are each 32 bits. A debug event may be enabled to occur upon loads, stores, or cache operations to an address specified in either DAC1 or DAC2, inside or outside a range specified by the DAC1 and DAC2, or to blocks of addresses specified by the combination of the DAC1 and DAC2.

	32 … 63
Field	Data address
Reset	All zeros
R/W	R/W
SPR	SPR 316 (DAC1); SPR 317 (DAC2)

Figure 2-30. Data Address Compare Registers (DAC1–DAC2)

The contents of DAC1 or DAC2 are compared to the address generated by a data access instruction.

2.10.2 Debug Counter Register (DBCNT)

The debug counter register (DBCNT) contains two 16-bit counters (CNT1 and CNT2) which can be configured to operate independently or can be 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 is 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 DBCR3. DBCNT is shown in Figure 2-31.

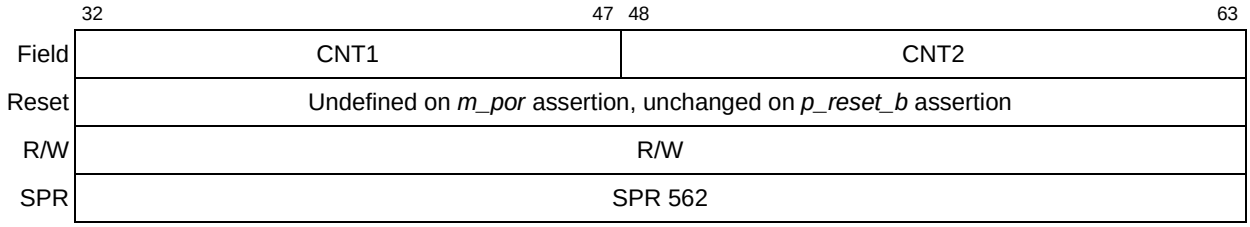


Figure 2-31. DBCNT Register

Refer to Section 2.10.3.4, “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.

2.10.3 Debug Control and Status Registers (DBCR0–DBCR3)

DBCR0–DBCR3 are used to enable debug events, reset the processor, control timer operation during debug events and set the debug mode of the processor. The debug status register (DBSR) records debug exceptions while internal or external debug mode is enabled.

The e200z6 requires that a context synchronizing instruction follow an **mtspr** that updates a DBCR 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** 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–DBCR3 are used, even though the corresponding events may be disabled (via DBCR0) from setting DBSR flags.

2.10.3.1 Debug Control Register 0 (DBCR0)

DBCR0 is used to enable debug modes and controls which debug events are allowed to set DBSR flags. The e200z6 adds bits to this register, as shown in Figure 2-32.

	32	33	34 35	36	37	38	39	40	41	42	43	44 45	46 47
Field	EDM	IDM	RST	ICMP	BRT	IRPT	TRAP	IAC1	IAC2	IAC3	IAC4	DAC1	DAC2
Reset	All zeros [1]												
R/W	R/W												

	48	49 52	53	54	55	56	57	58	59 62	63
	RET	—	DEVT1	DEVT2	DCNT1	DCNT2	CIRPT	CRET	—	FT
Reset	All zeros [1]									
R/W	R/W									
SPR	SPR 308									

Figure 2-32. DBCR0 Register

[1] DBCR0[EDM] is affected by *j_trst_b* or *m_por* assertion, and while in the test_logic_reset state, but not by *p_reset_b*. All other bits are reset by processor reset *p_reset_b* as well as by *m_por*.

Table 2-18 provides field definitions for DBCR0.

Table 2-18. DBCR0 Field Descriptions

Bits	Name	Description
32	EDM	External debug mode. This bit is read-only by software. Software may use EDM to determine if external debug has control over debug registers. The hardware debugger must set EDM before other DBCR0 bits (and other debug registers) can be altered. On the initial setting of EDM, all other bits are unchanged. EDM is writable only through the OnCE port. 0 External debug mode disabled. Internal debug events not mapped into external debug events. 1 External debug mode enabled. Events do not cause the CPU to vector to interrupt code. Software is not permitted to write to debug registers (DBCR0—DBCR3, DBSR, DBCNT, IAC1—IAC4, DAC1–DAC2).
		Programming notes: It is recommended that DBSR status bits be cleared before disabling external debug mode to avoid internal imprecise debug interrupts.
33	IDM	Internal debug mode 0 Debug exceptions are disabled. Debug events do not affect DBSR. 1 Debug exceptions are enabled. Enabled debug events update the DBSR. If MSR[DE] = 1, the occurrence of a debug event, or the recording of an earlier debug event in the DBSR when MSR[DE] was cleared, cause a debug interrupt.
34–35	RST	Reset control 00 No function 01 Reserved 10 *p_resetout_b* set by debug reset control. Allows external device to initiate processor reset. 11 Reserved
36	ICMP	Instruction complete debug event enable 0 ICMP debug events are disabled. 1 ICMP debug events are enabled.
37	BRT	Branch taken debug event enable 0 BRT debug events are disabled. 1 BRT debug events are enabled.

Table 2-18. DBCR0 Field Descriptions (continued)

Bits	Name	Description
38	IRPT	Interrupt taken debug event enable 0 IRPT debug events are disabled. 1 IRPT debug events are enabled.
39	TRAP	Trap taken debug event enable 0 TRAP debug events are disabled. 1 TRAP debug events are enabled.
40	IAC1	Instruction address compare 1 debug event enable 0 IAC1 debug events are disabled. 1 IAC1 debug events are enabled.
41	IAC2	Instruction address compare 2 debug event enable 0 IAC2 debug events are disabled. 1 IAC2 debug events are enabled.
42	IAC3	Instruction address compare 3 debug event enable 0 IAC3 debug events are disabled. 1 IAC3 debug events are enabled.
43	IAC4	Instruction address compare 4 debug event enable 0 IAC4 debug events are disabled. 1 IAC4 debug events are enabled.
44–45	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.
46–47	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
48	RET	Return debug event enable 0 RET debug events are disabled. 1 RET debug events are enabled.
49–52	—	Reserved
53	DEVT1	External debug event 1 enable 0 DEVT1 debug events are disabled. 1 DEVT1 debug events are enabled.
54	DEVT2	External debug event 2 enable 0 DEVT2 debug events are disabled. 1 DEVT2 debug events are enabled.
55	DCNT1	Debug counter 1 debug event enable 0 counter 1 debug events are disabled. 1 counter 1 debug events are enabled.
56	DCNT2	Debug counter 2 debug event enable 0 counter 2 debug events are disabled. 1 counter 2 debug events are enabled.

Table 2-18. DBCR0 Field Descriptions (continued)

Bits	Name	Description
57	CIRPT	Critical interrupt taken debug event enable 0 CIRPT debug events are disabled. 1 CIRPT debug events are enabled.
58	CRET	Critical return debug event enable 0 CRET debug events are disabled. 1 CRET debug events are enabled.
59–62	—	Reserved
63	FT	Freeze timers on debug event 0 Timebase timers are unaffected by set DBSR bits. 1 Disable clocking of timebase timers if any DBSR bit is set (except MRR or CNT1TRG).

2.10.3.2 Debug Control Register 1 (DBCR1)

DBCR1, shown in Figure 2-33, is used to configure instruction address compare operation.

	32 33	34 35	36 37	38 39	40 41	42 47	48 49	50 51	52 53	54 55	56 57	58 63
Field	IAC1US	IAC1ER	IAC2US	IAC2ER	IAC12M	—	IAC3US	IAC3ER	IAC4US	IAC4ER	IAC34M	—
Reset	All zeros											
R/W	R/W											
SPR	SPR 309											

Figure 2-33. Debug Control Register 1 (DBCR1)

Table 2-19 describes debug control register 1 fields.

Table 2-19. DBCR1 Field Descriptions

Bits	Name	Description
32–33	IAC1US	Instruction address compare 1 user/supervisor mode 00 IAC1 debug events are not affected by MSR[PR]. 01 Reserved 10 IAC1 debug events can only occur if MSR[PR] = 0 (supervisor mode). 11 IAC1 debug events can only occur if MSR[PR] = 1 (user mode).
34–35	IAC1ER	Instruction address compare 1 effective/real mode 00 IAC1 debug events are based on effective address. 01 Unimplemented in the e200z6 (Book E 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
36–37	IAC2US	Instruction address compare 2 user/supervisor mode 00 IAC2 debug events are not affected by MSR[PR]. 01 Reserved 10 IAC2 debug events can only occur if MSR[PR] = 0 (supervisor mode). 11 IAC2 debug events can only occur if MSR[PR] = 1 (user mode).

Table 2-19. DBCR1 Field Descriptions (continued)

Bits	Name	Description
38–39	IAC2ER	Instruction address compare 2 effective/real mode 00 IAC2 debug events are based on effective address. 01 Unimplemented in the e200z6 (Book E 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.
40–41	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 is 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.
42–47	—	Reserved
48–49	IAC3US	Instruction address compare 3 user/supervisor mode 00 IAC3 debug events are 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).
50–51	IAC3ER	Instruction address compare 3 effective/real mode 00 IAC3 debug events are based on effective address. 01 Unimplemented in the e200z6 (Book E 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.
52–53	IAC4US	Instruction address compare 4 user/supervisor mode 00 IAC4 debug events are not affected by MSR[PR]. 01 Reserved 10 IAC4 debug events can only occur if MSR[PR] = 0 (supervisor mode). 11 IAC4 debug events can only occur if MSR[PR] = 1 (user mode).
54–55	IAC4ER	Instruction address compare 4effective/real mode 00 IAC4 debug events are based on effective address. 01 Unimplemented in the e200z6 (Book E real address compare), no match can occur. 10 IAC4 debug events are based on effective address and can only occur if MSR[IS] = 0 11 IAC4 debug events are based on effective address and can only occur if MSR[IS] = 1

Table 2-19. DBCR1 Field Descriptions (continued)

Bits	Name	Description
56–57	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 is 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.
58–63	—	Reserved

2.10.3.3 Debug Control Register 2 (DBCR2)

DBCR2 is used to configure data address compare and data value compare operation. DBCR2 is shown in Figure 2-34.

	32 33	34 35	36 37	38 39	40 41	42	43	44 63
Field	DAC1US	DAC1ER	DAC2US	DAC2ER	DAC12M	DAC1LNK	DAC2LNK	—
Reset	All zeros							
R/W	R/W							
SPR	SPR 310							

Figure 2-34. DBCR2 Register

Table 2-20 describes debug control register 2 fields.

Table 2-20. DBCR2 Field Descriptions

Bits	Name	Description
32–33	DAC1US	Data address compare 1 user/supervisor mode 00 DAC1 debug events are not affected by MSR[PR]. 01 Reserved 10 DAC1 debug events can only occur if MSR[PR] = 0 (supervisor mode). 11 DAC1 debug events can only occur if MSR[PR] = 1 (User mode).
34–35	DAC1ER	Data address compare 1 effective/real mode 00 DAC1 debug events are based on effective address. 01 Unimplemented in the e200z6 (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.

Table 2-20. DBCR2 Field Descriptions (continued)

Bits	Name	Description
36–37	DAC2US	Data Address compare 2 user/supervisor mode. 00 DAC2 debug events are not affected by MSR[PR]. 01 Reserved 10 DAC2 debug events can only occur if MSR[PR] = 0 (supervisor mode). 11 DAC2 debug events can only occur if MSR[PR] = 1 (user mode).
38–39	DAC2ER	Data address compare 2 effective/real mode 00 DAC2 debug events are based on effective address. 01 Unimplemented in the e200z6 (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.
40–41	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 is 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.
42	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.
43	DAC2LNK	Data address compare 2 linked 0 No affect 1 DAC 2 debug events are linked to IAC3 debug events. IAC3 debug events do not affect DBSR. When linked to IAC3, DAC2 debug events are conditioned based on whether the instruction also generated an IAC3 debug event. DAC2 can only be linked if DAC12M specifies exact address compare because DAC2 debug events are not generated in the other compare modes.
44–63	—	Reserved for data value compare control (not supported by the e200z6)

2.10.3.4 Debug Control Register 3 (DBCR3)

DBCR3 is used to enable and configure the debug counter and debug counter events. For counter operation, the specific debug events that cause counters to decrement are specified in DBCR3.

NOTE

The corresponding events do not need to be (and probably should not be) enabled in DBCR0.

The IAC1–IAC4 and DAC1–DAC2 control fields in DBCR0 are ignored for counter operations and the control fields in DBCR3 determine when counting is enabled. DBCR1

and DBCR2 control fields are also used to determine the configuration of IAC1–IAC4 and DAC1–DAC2 operations for counting, even though the setting of bits in DBSR by corresponding events may be disabled via DBCR0. Multiple count-enabled events which occur during execution of an instruction typically cause only one decrement of a counter. As an example, if more than one IAC or DAC register hits and is enabled for counting, only one count can occur per counter. During execution of **lmw** and **stmw** instructions, multiple DAC*n* hits could occur. If the instruction is not interrupted before completion, a single decrement of a counter occurs.

NOTE

If the counters are operating independently, both may count for the same instruction.

The debug counter register (DBCNT) is configured by DBCR3[CONFIG] to operate either as separate 16-bit counter 1 and counter 2, or as a combined 32-bit counter (using control bits in DBCR3 for counter 1). Counters are enabled whenever any of their respective count enable event control bits are set and either DBCR0 or DBCR0[EDM] is set. 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 and is then enabled to begin operation. Triggering status for counter 1 is provided in the debug status register. Triggering mode is enabled by an **mtspr** DBCR3 which sets one or more of the trigger enable bits and also enables counter 1. The trigger can be re-armed by clearing the DBSR[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; 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 and RET

Limited support is provided for the following combinations:

- Any combination of IAC[1–4] with DAC[1–2] (linked or unlinked)

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 is 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 is 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 a DAC event within several instructions, it cannot be recognized immediately because 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 typically follows the IAC by up to several instructions.

Note that the counters operate regardless of whether counters are enabled to generate debug exceptions.

If counter 2 is used to trigger counter 1, counter 2 events should not normally be enabled in DBCR0 and are not blocked.

NOTE

Multiple IAC or DAC events are not counted during an **lmw** or **stmw** instruction, and no count occurs if either is interrupted by a critical input or external input interrupt before completion.

DBCR3 is an e200z6 implementation-specific register and is shown in Figure 2-35.

	32	33	34	35	36	37	38	39
Field	DEVT1C1	DEVT2C1	ICMPC1	IAC1C1	IAC2C1	IAC3C1	IAC4C1	DAC1RC1
Reset	All zeros							
R/W	R/W							

	40	41	42	43	44	45	46	47
Field	DAC1WC1	DAC2RC1	DAC2WC1	IRPTC1	RETC1	DEVT1C2	DEVT2C2	ICMPC2
Reset	All zeros							
R/W	R/W							

	48	49	50	51	52	53	54	55
	IAC1C2	IAC2C2	IAC3C2	IAC4C2	DAC1RC2	DAC1WC2	DAC2RC2	DAC2WC2
Reset	All zeros							
R/W	R/W							

	56	57	58	59	60	61	62	63
	DEVT1T1	DEVT2T1	IAC1T1	IAC3T1	DAC1RT1	DAC1WT1	CNT2T1	CONFIG
Reset	All zeros							
R/W	R/W							
SPR	SPR 561							

Figure 2-35. DBCR3 Register

Table 2-21 provides field definitions for debug control register 3

Table 2-21. DBCR3 Field Descriptions

Bits	Name	Description
32	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.
33	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.
34	ICMPC1	Instruction complete debug event count 1 enable 0 Counting ICMP debug events by counter 1 is disabled. 1 Counting ICMP debug events by counter 1 is enabled. **Note:** ICMP events are masked by MSR[DE] = 0 when operating in internal debug mode.
35	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.
36	IAC2C1	Instruction address compare2 debug event count 1 enable 0 Counting IAC2 debug events by counter 1 is disabled. 1 Counting IAC2 debug events by counter 1 is enabled.

Table 2-21. DBCR3 Field Descriptions (continued)

Bits	Name	Description
37	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.
38	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.
39	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.
40	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.
41	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.
42	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.
43	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.
44	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.
45	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.
46	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.
47	ICMPC2	Instruction complete debug event count 2 enable 0 Counting ICMP debug events by counter 2 is disabled. 1 Counting ICMP debug events by counter 2 is enabled. **Note:** ICMP events are masked by MSR[DE] = 0 when operating in internal debug mode.
48	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.
49	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.
50	IAC3C2	Instruction address compare 3 debug event count 2 enable 0 Counting IAC3 debug events by counter 2 is disabled. 1 Counting IAC3 debug events by counter 2 is enabled.

Table 2-21. DBCR3 Field Descriptions (continued)

Bits	Name	Description
51	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.
52	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.
53	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.
54	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.
55	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.
56	DEVT1T1	External debug event 1 trigger counter 1 enable 0 No effect 1 A DEVT1 debug event triggers counter 1 operation.
57	DEVT2T1	External debug event 2 trigger counter 1 enable 0 No effect 1 A DEVT2 debug event triggers counter 1 operation.
58	IAC1T1	Instruction address compare 1 trigger counter 1 enable 0 No effect 1 An IAC1 debug event triggers counter 1 operation.
59	IAC3T1	Instruction address compare 3 trigger counter 1 enable 0 No effect 1 An IAC3 debug event triggers counter 1 operation.
60	DAC1RT1	Data address compare 1 read trigger counter 1 enable 0 No effect 1 A DAC1R debug event triggers counter 1 operation.
61	DAC1WT1	Data address compare 1 write trigger counter 1 enable 0 No effect 1 A DAC1W debug event triggers counter 1 operation.
62	CNT2T1	Debug counter 2 trigger counter 1 enable 0 No effect 1 Counter 2 decrementing to 0 triggers counter 1 operation.
63	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 are counted if enabled in DBCR3. In general, DAC events should be disabled in DBCR0.

NOTE

Updates to DBCR0, DBSR, DBCR3, and DBCNT should be performed carefully if the counters are currently enabled for counting ICMP events. For these cases, an instruction that updates the counters or control over the counters may 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. This is due to the pipelined nature of the counter and control operation. As an example, if a counter was enabled to count ICMP events, MSR[DE] = 1, and the value of the counter is 1 before execution of an **mtspr** that is loading the counter with a different value, a counter event is generated following completion of the **mtspr**, even though the counter ends up being loaded with a new value. When the **mtspr** finishes executing, a debug event is posted, but the counter value holds the newly written count value. It is important to note that the new counter value is performed at the completion of an **mtspr** that modifies a counter, regardless of whether a debug event is generated based on the old counter value. To avoid this, it is recommended that DBCNT and DBCR3 values be modified only when there is no possibility of a counter-related debug event on the **mtspr** instruction. Modifying DBCR0 to affect counter event enabling/disabling may have similar issues, as may modifying DBSR[CNT1TRG].

Updates to DBCR0, DBSR, DBCR3 and DBCNT should be performed carefully if the counters are enabled for counting events. For these cases, an instruction that updates the counters or control over the counters can cause one or more counter events (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. This is due to the pipelined nature of the counter and control operation. As an example, if a counter was enabled to count ICMP events, MSR[DE] = 1, and the value of the counter is 1 before execution of an **mtspr** 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 prevents the particular event from being counted. When the **mtspr** finishes executing, a debug event is posted, but the DBCR3 value reflects the newly established

control, which may indicate that the particular event is not to cause a counter update. Modifying DBCR0 to affect counter event enabling/disabling may have similar issues, as may modifying DBSR[CNT1TRG].

2.10.4 Debug Status Register (DBSR)

DBSR contains status on debug events and the most recent processor reset. DBSR is set using hardware, and read and cleared using software. Clearing is done by writing to the DBSR with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The write data to the debug status register is not direct data, but a mask. A 1 causes the bit to be cleared, and a 0 has no effect. Debug status bits are set by debug events only while internal debug mode is enabled or external debug mode is enabled. When debug interrupts are enabled (MSR[DE] = 1, DBCR0[IDM] = 1, and DBCR0[EDM] = 0), a set bit in DBSR causes a debug interrupt to be generated. The debug interrupt handler is responsible for clearing DBSR bits before returning to normal execution. The DBSR register is shown in Figure 2-36.

	32	33	34 35	36	37	38	39	40	41	42	43	44	45	46	47
Field	IDE	UDE	MRR	ICMP	BRT	IRPT	TRAP	IAC1	IAC2	IAC3	IAC4	DAC1R	DAC1W	DAC2R	DAC2W
Reset	0001_0000_0000_0000														
R/W	Read/Clear														

	48	49 52	53	54	55	56	57	58	59 62	63
Field	RET	—	DEVT1	DEVT2	DCNT1	DCNT2	CIRPT	CRET	—	CNT1TRG
Reset	0000_0000_0000_0000									
R/W	Read/Clear									
SPR	SPR 304									

Figure 2-36. DBSR Register

Table 2-22 provides field definitions for the debug status register.

Table 2-22. DBSR Field Descriptions

Bits	Name	Description
32	IDE	Imprecise debug event Set if MSR[DE] = 0 and DBCR0[EDM] = 0 and a debug event causes its respective debug status register bit to be set. It may also be set if DBCR0[EDM] = 1 and an imprecise debug event occurs due to a DAC event on a load or store that is terminated with error, or if an ICMP event occurs in conjunction with a SPE FP round exception.
33	UDE	Unconditional debug event Set if an unconditional debug event occurred.

Table 2-22. DBSR Field Descriptions (continued)

Bits	Name	Description
34–35	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. 1*x* Reserved
36	ICMP	Instruction complete debug event. Set if an instruction complete debug event occurred.
37	BRT	Branch taken debug event. Set if an branch taken debug event occurred.
38	IRPT	Interrupt taken debug event. Set if an interrupt taken debug event occurred.
39	TRAP	Trap taken debug event. Set if a trap taken debug event occurred.
40	IAC1	Instruction address compare 1 debug event. Set if an IAC1 debug event occurred.
41	IAC2	Instruction address compare 2 debug event. Set if an IAC2 debug event occurred.
42	IAC3	Instruction address compare 3 debug event. Set if an IAC3 debug event occurred.
43	IAC4	Instruction address compare 4 debug event. Set if an IAC4 debug event occurred.
44	DAC1R	Data address compare 1 read debug event. Set if a read-type DAC1 debug event occurred while DBCR0[DAC1] = 0b10 or DBCR0[DAC1] = 0b11.
45	DAC1W	Data address compare 1 write debug event. Set if a write-type DAC1 debug event occurred while DBCR0[DAC1] = 0b01 or DBCR0[DAC1] = 0b11.
46	DAC2R	Data address compare 2 read debug event. Set if a read-type DAC2 debug event occurred while DBCR0[DAC2] = 0b10 or DBCR0[DAC2] = 0b11.
47	DAC2W	Data address compare 2 write debug event. Set if a write-type DAC2 debug event occurred while DBCR0[DAC2] = 0b01 or DBCR0[DAC2] = 0b11.
48	RET	Return debug event. Set if a Return debug event occurred.
49–52	—	Reserved, should be cleared.
53	DEVT1	External debug event 1 debug event. Set if a DEVT1 debug event occurred.
54	DEVT2	External debug event 2 debug event. Set if a DEVT2 debug event occurred.
55	DCNT1	Debug counter 1 debug event. Set if a DCNT1 debug event occurred.
56	DCNT2	Debug counter 2 debug event. Set if a DCNT2 debug event occurred.
57	CIRPT	Critical interrupt taken debug event. Set if a critical interrupt taken debug event occurred.
58	CRET	Critical return debug event. Set if a critical return debug event occurred.
59–62	—	Reserved, should be cleared.
63	CNT1TRG	Counter 1 triggered Set if debug counter 1 is triggered by a trigger event.

2.11 Hardware Implementation-Dependent Registers

Hardware implementation-dependent registers 0 and 1 (HID0 and HID1) are configuration registers provided to control various processor and system functions.

2.11.1 Hardware Implementation-Dependent Register 0 (HID0)

The HID0 register is used for various configuration and control functions. HID0 is shown in Figure 2-37.

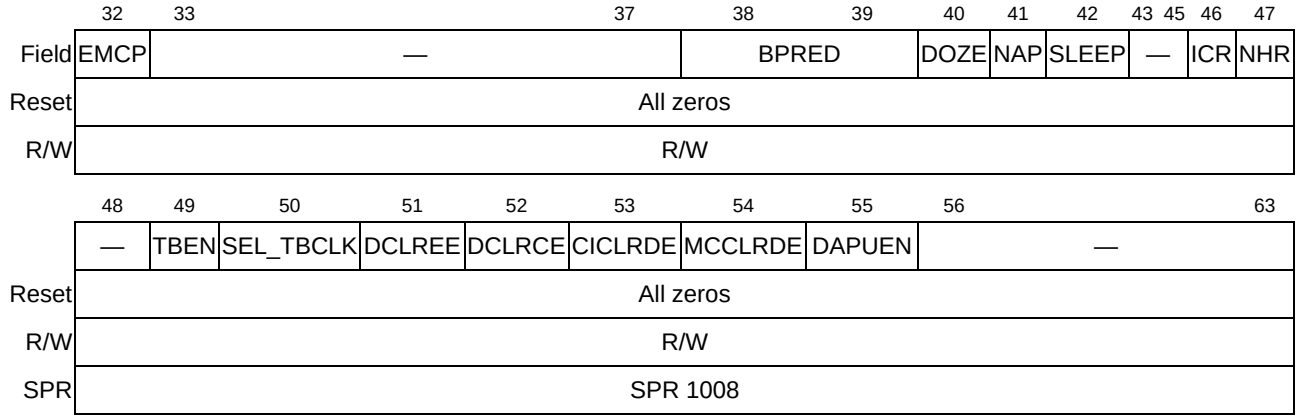


Figure 2-37. Hardware Implementation-Dependent Register 0 (HID0)

HID0 fields are described in Table 2-23.

Table 2-23. HID0 Field Descriptions

Bits	Name	Description
32	EMCP	Enable machine check signal (*p_mcp_b*). Used to mask out further machine check exceptions caused by assertion of *p_mcp_b*. 0 *p_mcp_b* is disabled. 1 *p_mcp_b* is enabled. If MSR[ME] = 0, asserting *p_mcp_b* causes checkstop. If MSR[ME] = 1, asserting *p_mcp_b* causes a machine check interrupt.
33–37	—	Reserved, should be cleared.
38–39	BPRED	Branch prediction (acceleration) control. Controls BTB lookahead for branch acceleration. 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. Used in conjunction with BUCSR. 00 Branch acceleration is enabled. 01 Branch acceleration is disabled for backward branches. 10 Branch acceleration is disabled for forward branches. 11 Branch acceleration is disabled for both branch directions.
40	DOZE	Configure for doze power management mode. Doze mode is invoked by setting MSR[WE] while this bit is set. 0 Doze mode is disabled. 1 Doze mode is enabled.
41	NAP	Configure for nap power management mode. Nap mode is invoked by setting MSR[WE] while this bit is set. 0 Nap mode is disabled. 1 Nap mode is enabled.
42	SLEEP	Configure for sleep power management mode. 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. 0 Sleep mode is disabled. 1 Sleep mode is enabled.

Table 2-23. HID0 Field Descriptions (continued)

Bits	Name	Description
43–45	—	Reserved, should be cleared.
46	ICR	Interrupt inputs clear reservation 0 External and critical input interrupts do not affect reservation status. 1 External and critical input interrupts clear an outstanding reservation.
47	NHR	Not hardware reset. Provided for software use. Set anytime by software, cleared by 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.
48	—	Reserved, should be cleared.
49	TBEN	Time base enable. Used to enable the time base and decrementer. 0 Time base is disabled. 1 Time base is enabled.
50	SEL_TBCLK	Select time base clock. Selects the time base clock source. 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 TBL and TBU are reinitialized following a change of time base clock source. 0 Time base is based on processor clock. 1 Time base is based on the *p_tbclk* input.
51	DCLREE	Debug interrupt clears MSR[EE]. Controls whether debug interrupts force external input interrupts to be disabled, or whether they remain unaffected. 0 MSR[EE] unaffected by debug interrupt 1 MSR[EE] cleared by debug interrupt
52	DCLRCE	Debug interrupt clears MSR[CE]. Controls whether debug interrupts force critical interrupts to be disabled, or whether they remain unaffected. 0 MSR[CE] unaffected by debug interrupt 1 MSR[CE] cleared by debug Interrupt
53	CICLRDE	Critical interrupt clears MSR[DE]. 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. 0 MSR[DE] unaffected by critical class interrupt 1 MSR[DE] cleared by critical class interrupt Note that if critical interrupt debug events are enabled (DBCR0[CIRPT] set, which should only be done when the debug APU is enabled), and MSR[DE] is set at the time of a (critical input, watchdog timer) critical interrupt, a debug event will be generated after the critical Interrupt Handler has been fetched, and the debug handler is executed first. In this case, DSRR0[DE] will have been cleared, such that after returning from the debug handler, the critical interrupt handler will not be run with MSR[DE] enabled.
54	MCCLRDE	Machine check interrupt clears MSR[DE]. Controls whether machine check interrupts force debug interrupts to be disabled or are unaffected. Note that if critical interrupt debug events are enabled (DBCR0[CIRPT] set, which should only be done when the debug APU is enabled), and MSR[DE] is set at the time of a machine check interrupt, a debug event is generated after the machine check interrupt handler has been fetched, and the debug handler is executed first. In this case, DSRR0[DE] will have been cleared, such that after returning from the debug handler, the machine check handler cannot be run if MSR[DE] = 1. 0 MSR[DE] unaffected by machine check interrupt 1 MSR[DE] cleared by machine check interrupt

Table 2-23. HID0 Field Descriptions (continued)

Bits	Name	Description
55	DAPUEN	Debug APU enable. 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. 0 Debug APU disabled. Debug interrupts use the critical interrupt resources CSRR0/CSRR1 for saving state, **rfci** is used for returning from a debug interrupt, and **rfdi** is treated as an illegal instruction. DCLREE, DCLRCE, CICLRDE, and MCCLRDE settings are ignored and are assumed to be 1s 1 Debug APU enabled Read and write access to DSRR0/DSRR1 via **mfspr** and **mtspr** is not affected by this bit.
56–63	—	Reserved, should be cleared.

2.11.2 Hardware Implementation-Dependent Register 1 (HID1)

The HID1 register is used for bus configuration and system control. HID1 is shown in Figure 2-38.

	32 — 55	56	57 — 63
Field	—	ATS	–
Reset	All zeros		
R/W	R/W		
SPR	SPR 1009		

Figure 2-38. Hardware Implementation-Dependent Register 1 (HID1)

HID1 fields are described in Table 2-24.

Table 2-24. HID1 Field Descriptions

Bits	Name	Description
32–55	—	Reserved, should be cleared.
56	ATS	Atomic status (read-only). Indicates state of the reservation bit in the load/store unit. See Section 3.3, “Memory Synchronization and Reservation Instructions,” for more detail.
57–63	—	Reserved, should be cleared.

2.12 Branch Target Buffer (BTB) Registers

This section describes the only register that controls the branch target buffer.

2.12.1 Branch Unit Control and Status Register (BUCSR)

The branch unit control and status register (BUCSR) is used for general control and status of the branch target buffer (BTB). BUCSR is shown in Figure 2-39.

	32 — 53	54	55 — 62	63
Field	—	BBFI	—	BPEN
Reset	All zeros			
R/W	R/W			
SPR	SPR 1013			

Figure 2-39. Branch Unit Control and Status Register (BUCSR)

BUCSR fields are described in Table 2-25.

Table 2-25. Branch Unit Control and Status Register

Bits	Name	Description
32–53	—	Reserved, should be cleared.
54	BBFI	Branch target buffer flash invalidate. When set, BBFI flash clears the valid bit of all BTB entries; clearing occurs regardless of the value of the enable bit (BPEN). **Note**: BBFI is always read as 0.
55–62	—	Reserved, should be cleared.
63	BPEN	Branch target buffer (BTB) enable. 0 BTB prediction disabled. No hits are generated from the BTB and no new entries are allocated. Entries are not automatically invalidated when BPEN is cleared; BBFI controls entry invalidation. 1 BTB prediction enabled (enables BTB to predict branches)

2.13 L1 Cache Configuration Registers

This section describes the registers that control and configure the L1 caches.

2.13.1 L1 Cache Control and Status Register 0 (L1CSR0)

The L1 cache control and status register 0 (L1CSR0) is a 32-bit register defined by the EIS. It is used for general control and status of the L1 data cache. L1CSR0 is accessed using an **mfspr** or **mtspr**. The SPR number for L1CSR0 is 1010 in decimal. L1CSR0 is shown in Figure 2-40.

The correct sequence necessary to change the value of L1CSR0 is as follows:

1. msync
2. isync
3. mtspr L1CSR0

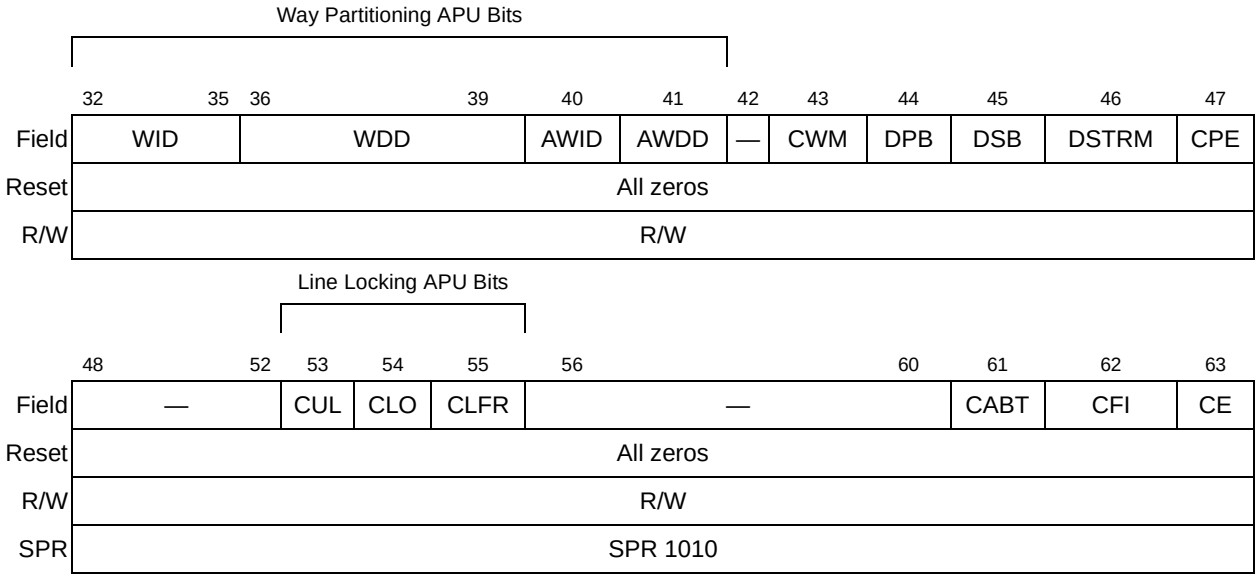


Figure 2-40. L1 Cache Control and Status Register 0 (L1CSR0)

L1CSR0 fields are described in Table 2-26.

Table 2-26. L1CSR0 Field Descriptions

Bits	Name	Description
32–35	WID	Way instruction disable. WID and WDD are used for locking ways of the cache and determining the cache replacement policy. 0 The corresponding way is available for replacement by instruction miss line fills. 1 The corresponding way 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, and bit 3 corresponds to way 3.
36–39	WDD	Way data disable. WID and WDD are used for locking ways of the cache and determining the cache replacement policy. 0 The corresponding way is available for replacement by data miss line fills. 1 The corresponding way is not available for replacement by data miss line fills. Bit 4 corresponds to way 0, bit 5 corresponds to way 1, bit 6 corresponds to way 2, bit 7 corresponds to way 3.
40	AWID	Additional ways instruction disable 0 Additional ways beyond 0–3 are available for replacement by instruction miss line fills. 1 Additional ways beyond 0–3 are not available for replacement by instruction miss line fills. For the 32-Kbyte 8-way cache, ways 4–7 are considered additional ways.
41	AWDD	Additional ways data disable 0 Additional ways beyond 0–3 are available for replacement by data miss line fills. 1 Additional ways beyond 0–3 are not available for replacement by data miss line fills. For the 32-Kbyte 8-way cache, ways 4–7 are considered additional ways.
42	—	Reserved, should be cleared.
43	CWM	Cache write mode. When set to write-through mode, the W page attribute from an optional MMU is ignored and all writes are treated as write through required. When set, write accesses are performed in copy-back mode unless the W page attribute from an optional MMU is set. 0 Cache operates in write-through mode. 1 Cache operates in copy-back mode.

Table 2-26. L1CSR0 Field Descriptions (continued)

Bits	Name	Description
44	DPB	Disable push buffer 0 Push buffer enabled 1 Push buffer disabled
45	DSB	Disable store buffer 0 Store buffer enabled 1 Store buffer disabled
46	DSTRM	Disable streaming 0 Streaming is enabled. 1 Streaming is disabled.
47	CPE	Cache parity enable 0 Parity checking is disabled. 1 Parity checking is enabled.
48–52	—	Reserved, should be cleared.
53	CUL	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 remain set until cleared by software writing 0 to this bit location.
54	CLO	Cache lock overflow. Indicates a lock overflow (overlocking) condition occurred. Set by hardware on an overlocking condition, and remains set until cleared by software writing 0 to this bit location.
55	CLFC	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 results in an undefined operation. Writing a 0 to this bit while a flash clear operation is in progress has no effect. Cache lock bits flash clear operations require approximately 134 cycles to complete. Clearing occurs regardless of the enable (CE) value.
56–60	—	Reserved, should be cleared.
61	CABT	Cache operation aborted. Indicates a cache invalidate or a cache lock bits flash clear operation was aborted prior to completion. Set by hardware on an aborted condition, and remains set until cleared by software writing 0 to this bit location.
62	CINV	Cache invalidate 0 No cache invalidate 1 Cache invalidation operation When written to a 1, a cache invalidation operation is initiated by hardware. Then invalidation is complete, CINV is reset to 0. Writing a 1 while invalidation is in progress causes an undefined operation. Writing a 0 to this bit while an invalidation operation is in progress is ignored. Cache invalidation operations require approximately 134 cycles to complete. Invalidation occurs regardless of the enable (CE) value.
63	CE	Cache enable. When disabled, cache lookups are not performed for normal load or store accesses. Other L1CSR0 cache control operations are still available. Also, store buffer operation is not affected by CE. 0 Cache is disabled 1 Cache is enabled

2.13.2 L1 Cache Configuration Register 0 (L1CFG0)

The L1 cache configuration register 0 (L1CFG0), shown in Figure 2-41, provides information about the configuration of the e200z6 L1 cache design.

	32 33	34	35	36	37 38	39 40	41 42	43	44	45 47
Field	CARCH	CWPA	CFAHA	CFISWA	—	CBSIZE	CREPL	CLA	CPA	CNWAY
Reset	01	1	0	1	00	00	10	1	1	000_0011_1 (8 way)/ 000_0001_1 (4 way)
R/W	Read only									

	48 52	53 63
Field	CNWAY	CSIZE
Reset	000_0011_1 (8 way)/ 000_0001_1 (4 way)	000_0010_0000 (32 Kbyte) 000_0001_0000 (16 Kbyte)
R/W	Read only	
SPR	SPR 515	

Figure 2-41. L1 Cache Configuration Register 0 (L1CFG0)

The L1CFG0 fields are described in Table 2-27.

Table 2-27. L1CFG0 Field Descriptions

Bits	Name	Description
32–33	CARCH	Cache architecture 01 The cache architecture is unified.
34	CWPA	Cache way partitioning available 1 The cache supports partitioning of way availability for I/D accesses.
35	CFAHA	Cache flush all by hardware available 0 The cache does not support flush all in hardware.
36	CFISWA	Cache flush/invalidate by set and way available 1 The cache supports flushing/invalidation by set and way via L1FINV0.
37–38	—	Reserved, should be cleared.
39–40	CBSIZE	Cache block size 00 The cache implements a block size of 32 bytes.
41–42	CREPL	Cache replacement policy 10 The cache implements a pseudo-round-robin replacement policy.
43	CLA	Cache locking APU available 1 The cache implements the line locking APU.
44	CPA	Cache parity available 1 The cache implements parity.
45–52	CNWAY	Number of ways in the data cache 0x03 The cache is 4-way set associative. 0x07 The cache is 8-way set associative.
53–63	CSIZE	Cache size 0x010 The size of the cache is 16 Kbytes. 0x020 The size of the cache is 32 Kbytes.

2.13.3 L1 Cache Flush and Invalidate Register (L1FINV0)

The L1FINV0 register provides software-based flush and invalidation control for the L1 cache supplied with this version of the e200z6 CPU core. A description of the L1FINV0 register can be found in Chapter 4, “L1 Cache.

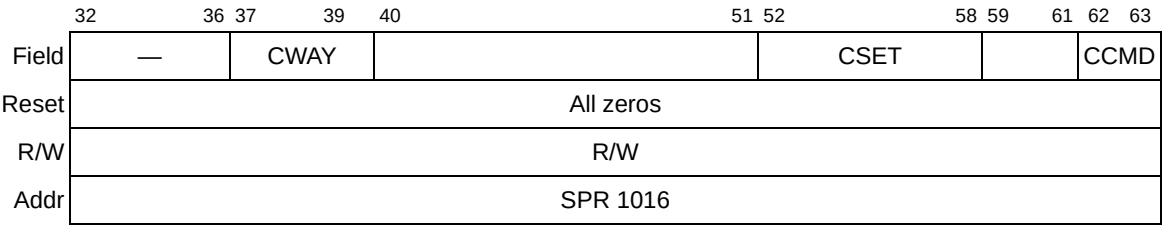


Figure 2-42. L1 Flush/Invalidate Register (L1FINV0)

The L1FINV0 fields are described in Table 2-28.

Table 2-28. L1FINV0 Field Descriptions

Bits	Name	Description
32-36	—	Reserved, should be cleared.
37-39	CWAY	Cache way Specifies the cache way to be selected
40-51	—	Reserved for set extension
52-58	CSET	Cache set Specifies the cache set to be selected
59-61	—	Reserved for set/command extension
62-63	CCMD	Cache command 00 The data contained in this entry is invalidated without flushing. 01 The data contained in this entry is flushed if dirty and valid without invalidation. 10 The data contained in this entry is flushed if dirty and valid and then is invalidated. 11 Reserved

2.14 MMU Registers

This section describes the registers used by the e200z6 for setting up and maintaining the TLBs in the MMU.

2.14.1 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 2-43.

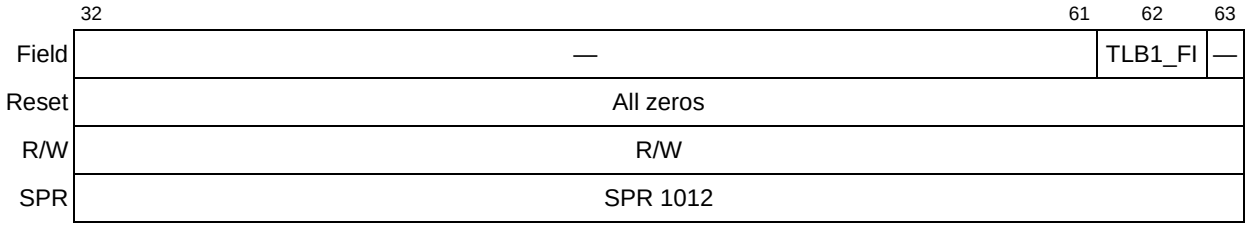


Figure 2-43. MMU Control and Status Register 0 (MMUCSR0)

The MMUCSR0 fields are described in Table 2-29.

Table 2-29. MMUCSR0 Field Descriptions

Bits	Name	Description
32–61	—	Reserved, should be cleared.
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 cleared to 0. Writing a 1 while an invalidation operation is in progress will result in an undefined operation. Writing a 0 to this bit while an invalidation operation is in progress will be ignored. TLB1 invalidation operations require 3 cycles to complete.
63	—	Reserved, should be cleared.

2.14.2 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 e200z6 MMU design. The MMUCFG register is shown in Figure 2-44.

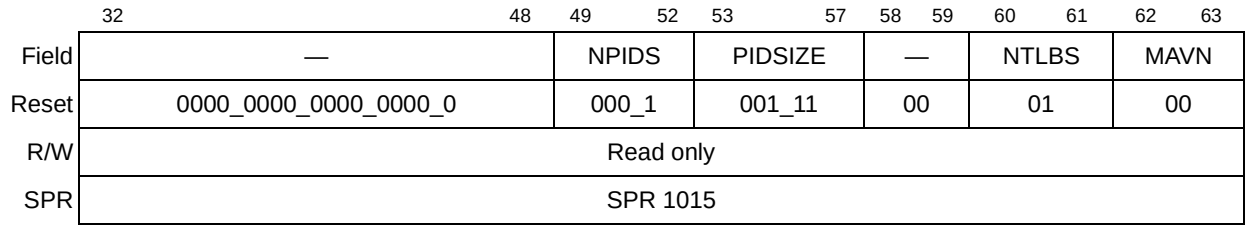


Figure 2-44. MMU Configuration Register 1 (MMUCFG)

The MMUCFG fields are described in Table 2-30.

Table 2-30. MMUCFG Field Descriptions

Bits	Name	Description
32–48	—	Reserved, should be cleared.
49–52	NPIDS	Number of PID registers 0001 This version of the MMU implements one PID register (PID0).
53–57	PIDSIZE	PID register size 00111 PID registers contain 8 bits in this version of the MMU.
58–59	—	Reserved, should be cleared.
60–61	NTLBS	Number of TLBs 01 This version of the MMU implements two TLB structures: a null TLB0 and a populated TLB1.
62–63	MAVN	MMU architecture version number 00 This version of the MMU implements version 1.0 of the Motorola Book E MMU architecture.

2.14.3 TLB Configuration Registers (TLB*n*CFG)

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

2.14.3.1 TLB Configuration Register 0 (TLB0CFG)

The TLB0 configuration register (TLB0CFG) is a 32-bit read-only register that provides information about the configuration of TLB0. The SPR number for TLB0CFG is 688 in decimal. Because the e200z6 MMU design does not implement TLB0, this register reads as all zeros. It is supplied to allow software to query it in a fashion compatible with other Motorola Book E designs. The TLB0CFG register is shown in Figure 2-45.

	32–39	40–43	44–47	48	49	50–51	52–63
Field	ASSOC	MINSIZE	MAXSIZE	IPROT	AVAIL	—	NENTRY
Reset	All zeros						
R/W	Read only						
SPR	SPR 688						

Figure 2-45. TLB Configuration Register 0 (TLB0CFG)

The TLB0CFG fields are described in Table 2-31.

Table 2-31. TLB0CFG Field Descriptions

Bits	Name	Description
32–39	ASSOC	Associativity
40–43	MINSIZE	Minimum page size
44–47	MAXSIZE	Maximum page size

Table 2-31. TLB0CFG Field Descriptions (continued)

Bits	Name	Description
48	IPROT	Invalidate protect capability
49	AVAIL	Page size availability
50–51	—	Reserved, should be cleared
52–63	NENTRY	Number of entries

2.14.3.2 TLB Configuration Register 1 (TLB1CFG)

The TLB1 Configuration Register (TLB1CFG) is a 32-bit read-only register. The SPR number for TLB1CFG is 689 decimal. TLB1CFG provides information about the configuration of TLB1 in the e200z6 MMU. The TLB1CFG register is shown in Figure 2-46.

	32–39	40–43	44–47	48	49	50–51	52–63
Field	ASSOC	MINSIZE	MAXSIZE	IPROT	AVAIL	—	NENTRY
Reset	0010_0000	0001	1001	1	1	00	0000_0010_0000
R/W	Read only						
SPR	SPR 689						

Figure 2-46. TLB Configuration Register 1 (TLB1CFG)

The TLB1CFG fields are described in Table 2-32.

Table 2-32. TLB1CFG Field Descriptions

Bits	Name	Description
32–39	ASSOC	Associativity 0x20 Indicates that TLB1 associativity is 32
40–43	MINSIZE	Minimum page size 0x1 Smallest page size is 4 Kbytes.
44–47	MAXSIZE	Maximum page size 0x9 Largest page size is 256 Mbytes.
48	IPROT	Invalidate protect capability 1 Invalidate protect capability is supported in TLB1.
49	AVAIL	Page size availability 1 All page sizes between MINSIZE and MAXSIZE are supported.
50–51	—	Reserved, should be cleared.
52–63	NENTRY	Number of entries 0x020 TLB1 contains 32 entries.

2.14.4 MMU Assist Registers (MAS0–MAS4, MAS6)

The e200z6 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 e200z6 does not implement the MAS5 register, present in other Motorola Book E designs, because the **tlbsx** instruction only searches based on a single SPID value.

Additional information on the MAS*n* registers is available in Section 6.6.5, “MMU Assist Registers (MAS).” The MAS0 register is shown in Figure 2-47.

	32 33	34 35	36 42	43 47	48 58	59 63
Field	—	TLBSEL	—	ESEL	—	NV
Reset	Undefined on *m_por* assertion, unchanged on *p_reset_b* assertion					
R/W	R/W					
SPR	SPR 624					

Figure 2-47. MAS Register 0 (MAS0) Format

MAS0 fields are defined in Table 2-33.

Table 2-33. MAS0—MMU Read/Write and Replacement Control

Bits	Name	Description
32–33	—	Reserved, should be cleared.
34–35	TLBSEL	Selects TLB for access 01 TLB1 (ignored by the e200z6, should be written to 01 for future compatibility)
36–42	—	Reserved, should be cleared.
43–47	ESEL	Entry select for TLB1
48–58	—	Reserved, should be cleared.
59–63	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 6-6).

The MAS1 register is shown in Figure 2-48.

	32	33	34 39	40 47	48 50	51	52 55	56 63
Field	VALID	IPROT	—	TID	—	TS	TSIZE	—
Reset	Undefined on *m_por* assertion, unchanged on *p_reset_b* assertion							
R/W	R/W							
SPR	SPR 625							

Figure 2-48. MMU Assist Register 1 (MAS1)

MAS1 fields are defined in Table 2-34.

Table 2-34. MAS1 —Descriptor Context and Configuration Control

Bits	Name	Description
32	VALID	TLB entry valid 0 This TLB entry is invalid. 1 This TLB entry is valid.
33	IPROT	Invalidation protect 0 Entry is not protected from invalidation. 1 Entry is protected from invalidation as described in Section 6.3.1, “IPROT Invalidation Protection in TLB1.” Protects TLB entry from invalidation by **tlbivax** (TLB1 only), or flash invalidates through MMUCSR0[TLB1_FI].
34–39	—	Reserved, should be cleared.
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.
48–50	—	Reserved, should be cleared.
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.
52–55	TSIZE	Entry page size Supported page sizes are: 0b0001 4 Kbytes 0b0110 4 Mbytes 0b0010 16 Kbytes 0b0111 16 Mbytes 0b0011 64 Kbytes 0b1000 64 Mbytes 0b0100 256 Kbytes 0b1001 256 Mbytes 0b0101 1 Mbyte All other values are undefined.
56–63	—	Reserved, should be cleared.

The MAS2 register is shown in Figure 2-49.

	32 … 51	52 … 58	59	60	61	62	63
Field	EPN	—	W	I	M	G	E
Reset	Undefined on *m_por* assertion, unchanged on *p_reset_b* assertion						
R/W	R/W						
SPR	SPR 626						

Figure 2-49. MMU Assist Register 2 (MAS2)

MAS2 fields are defined in Table 2-35.

Table 2-35. MAS2—EPN and Page Attributes

Bits	Name	Description
32–51	EPN	Effective page number
52–58	—	Reserved, should be cleared.
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.
60	I	Cache inhibited 0 This page is considered cacheable. 1 This page is considered cache-inhibited.
61	M	Memory coherence required.The e200z6 does not support the memory coherence required attribute, and thus it is ignored. 0 Memory coherence is not required. 1 Memory coherence is required.
62	G	Guarded. The e200z6ignores the guarded attribute (other than for generation of the *p_hprot[4:2]* attributes on an external access), since no speculative or out-of-order processing is performed. Refer to Section 4.16, “Page Table Control Bits,” for more information. 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).
63	E	Endianness. Determines endianness for the corresponding page. Refer to Section 3.2.4, “Byte Lane Specification,” for more information. 0 The page is accessed in big-endian byte order. 1 The page is accessed in true little-endian byte order.

The MAS3 register is shown in Figure 2-50.

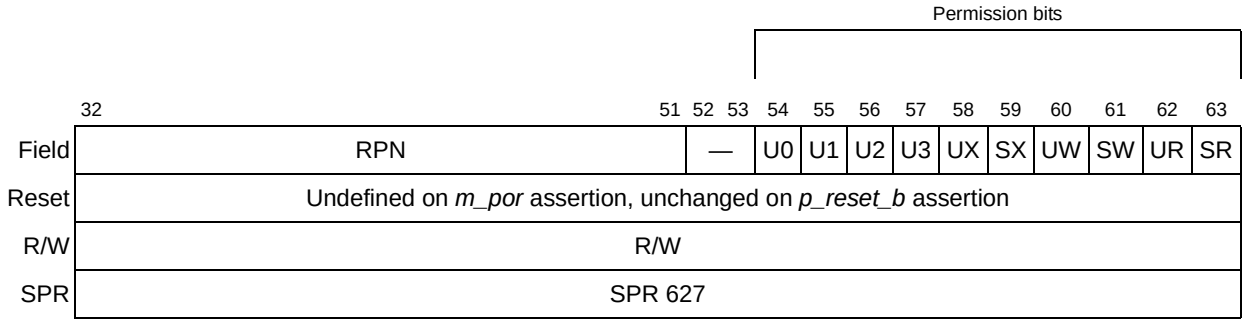


Figure 2-50. MMU Assist Register 3 (MAS3)

MAS3 fields are defined in Table 2-36

Table 2-36. MAS3—RPN and Access Control

Bits	Name	Description
32–51	RPN	Real page number Only bits that correspond to a page number are valid. Bits that represent offsets within a page are ignored and should be zero.
52–53	—	Reserved, should be cleared.
54–57	U0–U3	User bits
58–63	PERMIS	Permission bits (UX, SX, UW, SW, UR, SR)

The MAS4 register is shown in Figure 2-51.

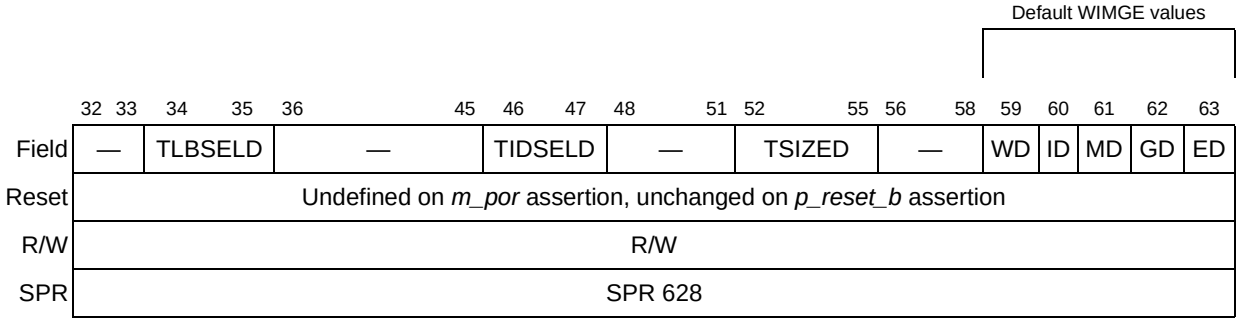


Figure 2-51. MMU Assist Register 4 (MAS4)

MAS4 fields are defined in Table 2-37.

Table 2-37. MAS4—Hardware Replacement Assist Configuration Register

Bits	Name	Description
32–33	—	Reserved, should be cleared.
34–35	TLBSELD	Default TLB selected 01 TLB1 (ignored by the e200z6, should be written to 01 for future compatibility)
36–45	—	Reserved, should be cleared.
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)
48–51	—	Reserved, should be cleared.
52–55	TSIZED	Default TSIZE value
56–58	—	Reserved, should be cleared.
59–63	DWIMGE	Default WIMGE values

The MAS6 register is shown in Figure 2-52.

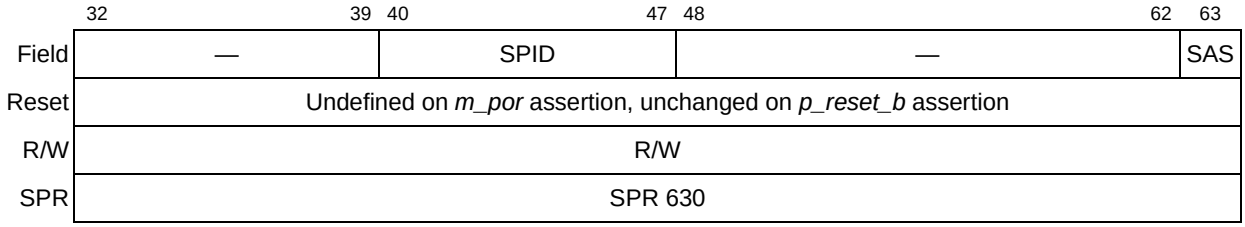


Figure 2-52. MMU Assist Register 6 (MAS6))

MAS6 fields are defined in Table 2-38.

Table 2-38. MAS6—TLB Search Context Register 0

Bits	Name	Description
32–39	—	Reserved, should be cleared.
40–47	SPID	PID value for searches
48–62	—	Reserved, should be cleared.
63	SAS	AS value for searches

2.14.5 Process ID Register (PID0)

The process ID register, PID0, is shown in Figure 2-53.

The Book E architecture defines that a process ID (PID) value be associated with each effective address (instruction or data) generated by the processor. Book E defines one PID register that maintains the value of the PID for the current process. The number of PIDs implemented is indicated by the value of MMUCFG[NPIDS]. (The e200z6 defines no additional PID registers.) PID values are used to construct virtual addresses for accessing memory.

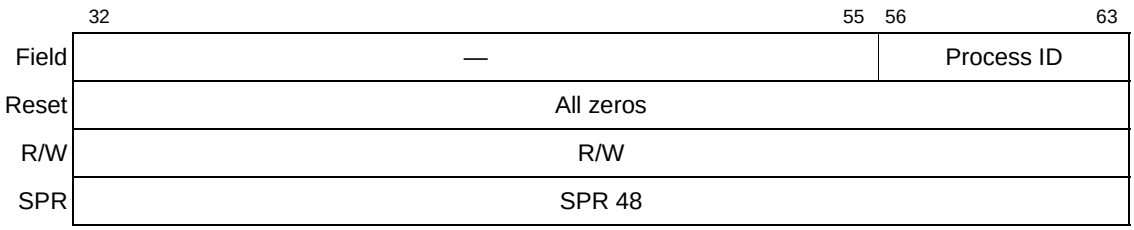


Figure 2-53. Process ID Register (PID0)

2.15 Support for Fast Context Switching

In order to provide real-time capabilities for embedded systems, future versions of the e200z6 core will include optional hardware support for fast context switching. The initial version of the e200z6 does not implement additional register contexts.

2.15.1 Context Control Register (CTXCR)

A new privileged 32-bit special purpose register (SPR) is defined in the e200z6 CPU core called the context control register (CTXCR). The CTXCR controls which context registers are mapped to the current context and holds current, alternate, and saved context information. CTXCR is readable by supervisor software to determine whether multiple contexts are supported in hardware, and if so, the number implemented. When multiple register contexts are present (CTXCR[NUMCTX] is non-zero), CTXCR is also writable; otherwise writes are ignored, and the register reads as all zeros. CTXCR is shown in Figure 2-54.

	32	33 34	35 37	38 40	41 43	44 46	47 55	56	57 59	60	61	62	63
Field	CTXEN	—	NUMCTX	CURCTX	SAVCTX	ALTCTX	—	R1CE	R1CSEL	XCE	LRCE	CTRCE	CRCE
Reset	000 || NUMCTX || 00_0000_0000_0000_0000_0000_0000												
R/W	Read/Write [1]												
SPR	SPR 560												

Figure 2-54. Context Control Register (CTXCR)

[1] Writes ignored if NUMCTX is 000 (register reads as all zeros).

Table 2-39. CTXCR Field Descriptions

Bits	Name	Description
32	CTXEN	Contexts enable—Enables the use of multiple contexts. 0 Only a single context is enabled, all other control fields in this register are ignored, and the current context is forced to 000. 1 Multiple context support is enabled. Current context is selected by the CURCTX field. This field is cleared to 0 on reset. Note that there is only a single implemented copy of the CTXEN bit shared among all n CTXCRs.
33–34	—	Reserved, should be cleared.
35–37	NUMCTX	Number of contexts This read-only field indicates the highest context number supported by the hardware. A value of 000 indicates one context is supported; a value of 111 indicates eight contexts are supported by the hardware. Writes to this field are ignored.
38–40	CURCTX	Current context number Defines the currently enabled context. This field is cleared to 0 on reset. When CTXEN = 0, CURCTX is forced to 000. Otherwise: • This field is set to the value obtained from the IVOR*n* register when an interrupt occurs. • This field is set to the value of the SAVCTX field on an **rfi**, **rfci**, or **rfdi** instruction. Note that there is only a single implemented copy of the CURCTX field shared among all n CTXCRs. Writing to this field changes the current context when multiple contexts are enabled. Care must be taken when modifying this value using an **mtspr** 560.
41–43	SAVCTX	Saved context number Defines the previously enabled context. This field is cleared to 0 on reset. This field is set to the CURCTX value on certain exceptions. This field is used to restore the CURCTX field on an **rfi**, **rfci**, or **rfdi** instruction.

Table 2-39. CTXCR Field Descriptions (continued)

Bits	Name	Description
44–46	ALTCTX	Alternate context number Defines an alternately enabled context. This field is used to define a context mapping for register groups.
47–55	—	Reserved, should be cleared.
56	R1CE	GPR R1 context enable Enables multiple GPR R1 contexts 0 GPR R1 is from context selected by field R1CSEL; CURCTX is ignored for GPR R1. 1 GPR R1 is from current context defined by CURCTX.
57–59	R1CSEL	GPR R1 context select Selects a context for GPR R1: 000 GPR R1 is from context 0. 001 GPR R1 is from context 1. ... 111 GPR R1 is from context 7. Results are undefined if this field is set to a value greater than the number of implemented contexts.
60	XCE	XER context enable Enables multiple XER contexts. 0 XER is always from context 0. 1 XER is from current context.
61	LRCE	Link register context enable Enables multiple LR contexts 0 LR is always from context 0. 1 LR is from current context.
62	CTRCE	Count register context enable Enables multiple CTR contexts 0 CTR is always from context 0. 1 CTR is from current context.
63	CRCE	Condition register context enable Enables multiple CR contexts 0 CR is always from context 0. 1 CR is from current context.

Software access to registers outside the current context is performed by setting of control bits in CTXCR which force selection of register groups from the context defined by the ALTCTX field. When multiple contexts are implemented (CTXCR[NUMCTX] is non-zero), alternate context control registers (CTXCR0–CTXCR7) are mapped indirectly to SPR 568 (ALTCTXCR) using the ALTCTX field of the current CTXCR. The current CTXCR is mapped to SPR 560. Supervisor mode accesses to ALTCTXCR when no alternate contexts are implemented will result in an illegal type program interrupt. Software should query the CTXCR first to determine the number of hardware supported contexts.

2.16 SPR Register Access

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

2.16.1 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.

Table 2-40. 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

2.16.2 Synchronization Requirements for SPRs

With the exception of the following registers, there are no synchronization requirements for accessing SPRs beyond those stated in PowerPC Book E. A complete description of synchronization requirements is contained in the EREF. Software requirements for synchronization before/after accessing these registers are shown in Table 2-41. The notation CSI in the table refers to context synchronizing instructions; these include **sc**, **isync**, **rfi**, **rfci**, and **rfdi**.

Table 2-41. Additional Synchronization Requirements for SPRs

Context Altering Event or Instruction		Required Before	Required After	Notes
mtmsr[UCLE]		None	CSI	
mfspr				
DBCNT	Debug counter register	**msync**	None	1
DBSR	Debug status register	**msync**	None	
HID0	Hardware implementation dependent register 0	None	None	
HID1	Hardware implementation dependent register 1	**msync**	None	
L1CSR0	L1 cache control and status register 0	**msync**	None	
L1FINV0	L1 cache flush and invalidate control register 0	**msync**	None	

Table 2-41. Additional Synchronization Requirements for SPRs (continued)

Context Altering Event or Instruction		Required Before	Required After	Notes
MMUCSR	MMU control and status register 0	CSI	None	
mtspr				
BUCSR	Branch unit control and status register	None	CSI	
CTXCR	Context control register	CSI	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	
DBSR	Debug status register	**msync**	None	
HID0	Hardware implementation dependent reg 0	CSI	CSI	
L1CSR0	L1 cache control and status register 0	**msync**	CSI	
L1FINV0	L1 cache flush and invalidate control register 0	**msync**	CSI	
MMUCSR	MMU control and status register 0	CSI	CSI	

Notes:
1. Not required if counter is not currently enabled

2.16.3 Special Purpose Register Summary

PowerPC Book E and implementation-specific SPRs for the e200z6 core are listed in the following table. All registers are 32 bits in size. Register bits are numbered from bit 32 to bit 63 (most significant to least significant). Shaded entries represent optional registers. An SPR 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 2-42. Special Purpose Registers

Mnemonic	Name	SPR Number	Access	Privileged	e200z6 Specific
ALTCTXCR	Alternate context control register	568	R/W [1]	Yes	Yes
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
CTXCR	Context control register	560	R/W [2]	Yes	Yes
DAC1	Data address compare 1	316	R/W	Yes	No

Table 2-42. Special Purpose Registers (continued)

Mnemonic	Name	SPR Number	Access	Privileged	e200z6 Specific
DAC2	Data address compare 2	317	R/W	Yes	No
DBCNT	Debug counter register	562	R/W	Yes	Yes
DBCR0	Debug control register 0	308	R/W	Yes	No
DBCR1	Debug control register 1	309	R/W	Yes	No
DBCR2	Debug control register 2	310	R/W	Yes	No
DBCR3	Debug control register 3	561	R/W	Yes	Yes
DBSR	Debug status register	304	Read/Clear [3]	Yes	No
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
DSRR0	Debug save/restore register 0	574	R/W	Yes	Yes
DSRR1	Debug save/restore register 1	575	R/W	Yes	Yes
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
IVOR0	Interrupt vector offset register 0	400	R/W	Yes	No
IVOR1	Interrupt vector offset register 1	401	R/W	Yes	No
IVOR2	Interrupt vector offset register 2	402	R/W	Yes	No
IVOR3	Interrupt vector offset register 3	403	R/W	Yes	No
IVOR4	Interrupt vector offset register 4	404	R/W	Yes	No
IVOR5	Interrupt vector offset register 5	405	R/W	Yes	No
IVOR6	Interrupt vector offset register 6	406	R/W	Yes	No
IVOR7	Interrupt vector offset register 7	407	R/W	Yes	No
IVOR8	Interrupt vector offset register 8	408	R/W	Yes	No
IVOR9 [4]	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

Table 2-42. Special Purpose Registers (continued)

Mnemonic	Name	SPR Number	Access	Privileged	e200z6 Specific
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 configuration register 0	515	Read only	No	Yes
L1CSR0	L1 cache control and status register 0	1010	R/W	Yes	Yes
L1FINV0	L1 cache flush and invalidate control register 0	1016	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
MCSR	Machine check syndrome register	572	R/W	Yes	Yes
MMUCFG	MMU configuration register	1015	Read only	Yes	Yes
MMUCSR0	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	Read only	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

Table 2-42. Special Purpose Registers (continued)

Mnemonic	Name	SPR Number	Access	Privileged	e200z6 Specific
SPRG7	SPR general 7	263	Read only	No	No
		279	R/W	Yes	No
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 [5]	Yes	No
USPRG0	User SPR general 0	256	R/W	No	No
XER	Integer exception register	1	R/W	No	No

Notes:

1 Only accessible when multiple contexts are implemented, otherwise treated as an illegal SPR.

2 Only writable when multiple contexts are implemented, otherwise writes are ignored

3 The debug status register (DBSR) is read using **mfspr**. DBSR cannot be directly written to. Instead, DBSR bits corresponding to 1 bits in the GPR can be cleared using **mtspr**.

4 IVOR9 is defined to handle the auxiliary processor unavailable. This interrupt is defined by the EIS but not supported in the e200z6; therefore, use of IVOR9 is not supported in the e200z6.

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

2.16.4 Reset Settings

Table 2-43 shows the state of the PowerPC Book E architected registers and other optional resources immediately following a system reset.

Table 2-43. Reset Settings for e200z6 Resources

Resource	System Reset Setting
Program counter	*p_rstbase[0:19]* || 0xFFC
GPRs	Unaffected [1]
CR	Unaffected [1]
BUCSR	0x0000_0000

Table 2-43. Reset Settings for e200z6 Resources (continued)

Resource	System Reset Setting
CSRR0	Unaffected [1]
CSRR1	Unaffected [1]
CTR	Unaffected [1]
CTXCR	0x0000_0000 || NUMCTX [2]
ALTCTXCR	Unaffected [1]
DAC1–DAC2	0x0000_0000
DBCNT	Unaffected [1]
DBCR0–DBCR3	0x0000_0000
DBSR	0x1000_0000
DEAR	Unaffected [1]
DEC	Unaffected [1]
DECAR	Unaffected [1]
DSRR0	Unaffected [1]
DSRR1	Unaffected [1]
ESR	0x0000_0000
HID0–HID1	0x0000_0000
IAC1–IAC4	0x0000_0000
IVOR0–IVOR15	Unaffected [1]
IVOR32–IVOR34	Unaffected [1]
IVPR	Unaffected [1]
L1CFG0 [3]	—
L1CSR0	0x0000_0000
L1FINV0	0x0000_0000
LR	Unaffected [1]
MAS0–MAS4, MAS6	Unaffected [1]
MCSR	0x0000_0000
MMUCFG [3]	—
MMUCSR0	0x0000_0000
MSR	0x0000_0000
PID0	0x0000_0000
PIR [3]	—
PVR [3]	—
SPEFSCR	0x0000_0000

Table 2-43. Reset Settings for e200z6 Resources (continued)

Resource	System Reset Setting
SPRG0–SPRG7	Unaffected [1]
SRR0	Unaffected [1]
SRR1	Unaffected [1]
SVR [3]	—
TBL	Unaffected [1]
TBU	Unaffected [1]
TCR	0x0000_0000
TLB0CFG– TLB1CFG	—
TSR	Undefined on power-on reset; otherwise, 0x(0b00||WRS)000_0000
USPRG0	Unaffected [1]
XER	0x0000_0000

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

[2] For CTXCR 0 only, others unaffected

[3] Read-only register

Chapter 3
Instruction Model

This chapter provides additional information about the Book E architecture as it relates specifically to the e200z6.

The e200z6 is a 32-bit implementation of the Book E architecture. This architecture specification includes a recognition that different processor implementations may require clarifications, extensions or deviations from the architectural descriptions. Book E instructions are described in the EREF.

3.1 Operand Conventions

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

3.1.1 Data Organization in Memory and Data Transfers

Bytes in memory are numbered consecutively starting with 0. Each number is the address of the corresponding byte.

Memory operands can be bytes, half words, words, or double words (consisting of two 32-bit elements) or, for the load/store multiple instruction type, a sequence of bytes or words. The address of a memory operand is the address of its first byte (that is, of its lowest-numbered byte). Operand length is implicit for each instruction.

3.1.2 Alignment and Misaligned Accesses

The e200z6 core provides hardware support for misaligned memory accesses; however, there is performance degradation for accesses that cross a 64-bit (8-byte) boundary. For loads that hit in the cache, the throughput of the load/store unit is degraded to 1 misaligned load every 2 cycles. Stores misaligned across a 64-bit (8 byte) boundary can be translated

at a rate of 2 cycles per store. Frequent use of misaligned memory accesses is discouraged because of the impact on performance.

NOTE

Accesses that cross a translation boundary may be restarted. A misaligned access that crosses a page boundary is restarted entirely if the second portion of the access causes a TLB miss. This may result in the first portion being accessed twice.

Accesses that cross a translation boundary where the endianness changes cause a byte ordering DSI exception.

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

3.1.3 e200z6 Floating-Point Implementation

The e200z6 core does not implement the floating-point instructions as they are defined in Book E. Attempts to execute a Book E–defined floating-point instruction result in an illegal instruction exception. However, the vector SPFP APU supports single-precision vector (64-bit, two 32-bit operand) instructions, and the scalar SPFP APU performs single-precision floating-point operations using the lower 32 bits of the GPRs. These instructions are described in Section 3.6.4, “Embedded Vector and Scalar Single-Precision Floating-Point APU Instructions.” Unlike the PowerPC UISA, the SPFP APUs store floating-point values as single-precision values in true 32-bit, single-precision format rather than in a 64-bit double-precision format used with FPRs.

3.2 Unsupported Instructions and Instruction Forms

Because the e200z6 is a 32-bit Book E core, all of the instructions defined for 64-bit implementations of the Book E architecture are illegal on the e200z6 and cause an illegal instruction exception type program interrupt. Some instructions have the following optional features indicated by square brackets:

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

The e200z6 core does not support the instructions listed in Table 3-1. An unimplemented instruction or floating-point unavailable exception is generated if the processor attempts to execute one of these instructions.

Table 3-1. Unsupported 32-Bit Book E Instructions

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

Table 3-1. Unsupported 32-Bit Book E Instructions (continued)

Name	Mnemonic
Move To FPSCR Field [Immediate] [and record CR]	**mtfsf**[**i**][**.**]
Store Floating-Point Double [with Update] [Indexed] [Extended]	**stfd**[**u**][**x**][**e**]
Store Floating-Point as Integer Word Indexed [Extended]	**stfiwx**[**e**]
Store Floating-Point Single [with Update] [Indexed] [Extended]	**stfs**[**u**][**x**][**e**]
Store String Word Immediate	**stswi**
Store String Word Indexed	**stswx**

3.3 Memory Synchronization and Reservation Instructions

Table 3-2 lists the e200z6 implementation details for the memory synchronization and load and store with reservation instructions.

Table 3-2. Memory Synchronization and Reservation Instructions—e200z6-Specific Details

Instructions	e200z6 Implementation
msync	Provides synchronization and memory barrier functions. **msync** completes only after all preceding instructions and data memory accesses complete. Subsequent instructions in the stream are not dispatched until after the **msync** ensures these functions have been performed.
mbar	**mbar** behaves identically to **msync**; the **mbar** MO field is ignored by the e200z6 core.
lwarx/stwcx.	Implemented as described in the EREF. If the EA for either instruction is not a multiple of four, an alignment interrupt is invoked. The e200z6 allows **lwarx** and **stwcx.** to access a page marked as write-through required or cache-inhibited without invoking a data storage interrupt. As Book E allows, the e200z6 does not require the EAs for a **stwcx.** and the preceding **lwarx** to be to the same reservation granule. Reservation granularity is implementation dependent. The e200z6 does not define a reservation granule explicitly; it is defined by external logic. When no external logic is provided, the e200z6 does not compare addresses; thus, the effective implementation granularity is null. The e200z6 implements an internal status flag, HID1[ATS], which is set when a **lwarx** completes without error. It remains set until it is cleared by one of the following: • A **stwcx.** executes without error • The e200z6 core *p_rsrv_clr* input is asserted. See Chapter 8, “External Core Complex Interfaces.” • The reservation is invalidated when an external interrupt is signaled and HID0[ICR] is set. The e200z6 treats **lwarx** and **stwcx.** accesses as though they were cache-inhibited and guarded, regardless of page attributes. A cache line corresponding to the address of a **lwarx** or **stwcx.** access is flushed to memory if it is modified, and then invalidated, before the access is issued to the bus. This allows external reservation logic to be built that properly signals a reservation failure. The e200z6 core input *p_xfail_b* is sampled at termination of a **stwcx.** store transfer to allow an external agent or mechanism to indicate that the **stwcx.** failed to update memory, even though a reservation existed for the store when it was issued. This is not considered an error and causes the condition codes for the **stwcx.** to be written as if it had no reservation. Also, any outstanding reservation is cleared.

3.4 Branch Prediction

The e200z6 instruction fetching mechanism uses a branch target buffer (BTB), which holds branch target addresses combined with a 2-bit saturating up-down counter scheme for branch prediction. These bits can take four values: strongly taken, weakly taken, weakly not taken, and strongly not taken.

Branch paths are predicted by a BTB 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 not taken for any branch whose fetch address misses in the BTB or hits in the BTB and is predicted not taken by the counter.
- Predict taken for any branch that hits in the BTB and is predicted taken by the counter.

Note that the static branch prediction bit defined by the Book E architecture in the BO operand is ignored.

3.5 Interruption of Instructions by Interrupt Requests

In general, the e200z6 core samples pending external input and critical input interrupt requests at instruction boundaries. However, in order to reduce interrupt latency, long-running instructions may be interrupted prior to completion. Instructions in this class include divides (**divw**[**uo**][**.**], **efsdiv**, **evfsdiv**, **evdivw**[**su**]), Load Multiple Word (**lmw**), and Store Multiple Word (**stmw**). When interrupted prior to completion, the value saved in SRR0/CSRR0 is the address of the interrupted instruction.

3.6 e200z6-Specific Instructions

The e200z6 core implements the following instructions that are not defined by the Book E architecture:

- The Motorola Book E integer select (**isel**) APU consists of the **isel** instruction, described in Section 3.6.1, “Integer Select APU.”
- The Return from Debug Interrupt instruction (**rfdi**) is defined by the Motorola Book E debug APU. This instruction is described in Section 3.6.2, “Debug APU.”
- The signal processing extension (SPE) APU provides a set of 64-bit SIMD instructions. These are listed in Section 3.6.3, “SPE APU Instructions,” and described in the EREF.
- The embedded vector and scalar single-precision floating-point APUs are listed along with supporting instructions in Section 3.6.4, “Embedded Vector and Scalar

Single-Precision Floating-Point APU Instructions." These instructions are described in detail in the EREF.

- The Motorola Book E cache line locking APU is described in Section 4.12, "Cache Line Locking/Unlocking APU."

3.6.1 Integer Select APU

The integer select APU defines the Integer Select (**isel**) instruction, which provides a means to select one of two registers and place the result in a destination register under the control of a predicate value supplied by a bit in the condition register. **isel** can be used to eliminate branches in software and in many cases improve performance; it 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 **isel** instruction is fully described in the EREF.

3.6.2 Debug APU

The e200z6 implements the Motorola Book E debug APU to support the ability to handle the debug interrupt as an additional interrupt level. To support this interrupt level, the Return from Debug Interrupt instruction (**rfdi**) is defined as part of the debug APU, along with a new pair of save/restore registers, DSRR0, and DSRR1.

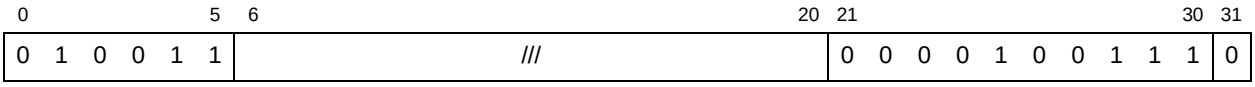
When the debug APU is enabled (HID0[DAPUEN] = 1), **rfdi** provides a way to return from a debug interrupt. See Section 2.11.1, "Hardware Implementation-Dependent Register 0 (HID0)," for more information about enabling the debug APU.

The instruction form and definition is as follows:

rfdi rfdi

Return from Debug Interrupt

rfdi




```
MSR ←DSRR1
PC ←DSRR0 0:61 || 0b00
```

rfdi is used to return from a debug interrupt or as a way of simultaneously establishing a new context and synchronizing on that new context.

The contents of debug save/restore register 1 (DSRR1) are placed into the MSR. If the new MSR value does not enable any pending exceptions, the next instruction is fetched, under control of the new MSR value, from the address DSRR0[0–29] || 0b00. If the new MSR 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 SRR0 or CSRR0 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 DSRR0 at the time of the execution of the **rfdi**).

Execution of this instruction is privileged and context synchronizing.

Registers altered:

- MSR

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

3.6.3 SPE APU Instructions

SPE APU instructions treat 64-bit GPRs as a vector of two 32-bit elements. (Some instructions also read or write 16-bit elements.) The SPE APU supports a number of forms of multiply and multiply-accumulate operations, and of add and subtract to accumulator operations. The SPE supports signed and unsigned forms, and optional fractional forms. For these instructions, the fractional form does not apply to unsigned forms because integer and fractional forms are identical for unsigned operands.

Table 3-3 shows how SPE APU vector multiply instruction mnemonics are structured.

Table 3-3. SPE APU Vector Multiply Instruction Mnemonic Structure

Prefix	Multiply Element		Data Type Element		Accumulate Element	
evm	ho	half odd (16x16→32)	usi	unsigned saturate integer	a	write to ACC
	he	half even (16x16→32)	umi	unsigned modulo integer	aa	write to ACC & added ACC
	hog	half odd guarded (16x16→32)	ssi	signed saturate integer	an	write to ACC & negate ACC
	heg	half even guarded (16x16→32)	ssf [1]	signed saturate fractional	aaw	write to ACC & ACC in words
	wh	word high (32x32→32)	smi	signed modulo integer	anw	write to ACC & negate ACC in words
	wl	word low (32x32→32)	smf[1]	signed modulo fractional		
	whg	word high guarded (32x32→32)				
	wlg	word low guarded (32x32→32)				
	w	word (32x32→64)				

[1] Low word versions of signed saturate and signed modulo fractional instructions are not supported. Attempting to execute an opcode corresponding to these instructions causes boundedly undefined results.

Table 3-4 defines mnemonic extensions for these instructions.

Table 3-4. 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 accumulator result
ho	Half word odd	16×16→32
hog	Half word odd guarded	16×16→32, 64-bit final accumulator result
w	Word	32×32→64
wh	Word high	32×32→32, high-order 32 bits of product
wl	Word low	32×32→32, low-order 32 bits of product
Data Type		
smf	Signed modulo fractional	(Wrap, no saturate)
smi	Signed modulo integer	(Wrap, no saturate)
ssf	Signed saturate fractional	
ssi	Signed saturate integer	
umi	Unsigned modulo integer	(Wrap, no saturate)
usi	Unsigned saturate integer	
Accumulate Options		
a	Update accumulator	Update accumulator (no add)
aa	Add to accumulator	Add result to accumulator (64-bit sum)
aaw	Add to accumulator (words)	Add word results to accumulator words (pair of 32-bit sums)
an	Add negated	Add negated result to accumulator (64-bit sum)
anw	Add negated to accumulator (words)	Add negated word results to accumulator words (pair of 32-bit sums)

Table 3-5 lists SPE APU instructions.

Table 3-5. SPE APU Vector Instructions

Instruction	Mnemonic	Syntax
Bit Reversed Increment [1]	**brinc**	**r**D,**r**A,**r**B
Initialize Accumulator	**evmra**	**r**D,**r**A
Multiply Half Words, Even, Guarded, Signed, Modulo, Fractional and Accumulate	**evmhegsmfaa**	**r**D,**r**A,**r**B
Multiply Half Words, Even, Guarded, Signed, Modulo, Fractional and Accumulate Negative	**evmhegsmfan**	**r**D,**r**A,**r**B
Multiply Half Words, Even, Guarded, Signed, Modulo, Integer and Accumulate	**evmhegsmiaa**	**r**D,**r**A,**r**B
Multiply Half Words, Even, Guarded, Signed, Modulo, Integer and Accumulate Negative	**evmhegsmian**	**r**D,**r**A,**r**B
Multiply Half Words, Even, Guarded, Unsigned, Modulo, Integer and Accumulate	**evmhegumiaa**	**r**D,**r**A,**r**B
Multiply Half Words, Even, Guarded, Unsigned, Modulo, Integer and Accumulate Negative	**evmhegumian**	**r**D,**r**A,**r**B
Multiply Half Words, Odd, Guarded, Signed, Modulo, Fractional and Accumulate	**evmhogsmfaa**	**r**D,**r**A,**r**B
Multiply Half Words, Odd, Guarded, Signed, Modulo, Fractional and Accumulate Negative	**evmhogsmfan**	**r**D,**r**A,**r**B
Multiply Half Words, Odd, Guarded, Signed, Modulo, Integer and Accumulate	**evmhogsmiaa**	**r**D,**r**A,**r**B
Multiply Half Words, Odd, Guarded, Signed, Modulo, Integer and Accumulate Negative	**evmhogsmian**	**r**D,**r**A,**r**B
Multiply Half Words, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate	**evmhogumiaa**	**r**D,**r**A,**r**B
Multiply Half Words, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate Negative	**evmhogumian**	**r**D,**r**A,**r**B
Vector Absolute Value	**evabs**	**r**D,**r**A
Vector Add Immediate Word	**evaddiw**	**r**D,**r**B,UIMM
Vector Add Signed, Modulo, Integer to Accumulator Word	**evaddsmiaaw**	**r**D,**r**A,**r**B
Vector Add Signed, Saturate, Integer to Accumulator Word	**evaddssiaaw**	**r**D,**r**A
Vector Add Unsigned, Modulo, Integer to Accumulator Word	**evaddumiaaw**	**r**D,**r**A
Vector Add Unsigned, Saturate, Integer to Accumulator Word	**evaddusiaaw**	**r**D,**r**A
Vector Add Word	**evaddw**	**r**D,**r**A,**r**B
Vector AND	**evand**	**r**D,**r**A,**r**B
Vector AND with Complement	**evandc**	**r**D,**r**A,**r**B
Vector Compare Equal	**evcmpeq**	**cr**D,**r**A,**r**B
Vector Compare Greater Than Signed	**evcmpgts**	**cr**D,**r**A,**r**B
Vector Compare Greater Than Unsigned	**evcmpgtu**	**cr**D,**r**A,**r**B
Vector Compare Less Than Signed	**evcmplts**	**cr**D,**r**A,**r**B
Vector Compare Less Than Unsigned	**evcmpltu**	**cr**D,**r**A,**r**B
Vector Convert Floating-Point from Signed Fraction	**evfscfsf**	**r**D,**r**B
Vector Convert Floating-Point from Signed Integer	**evfscfsi**	**r**D,**r**B
Vector Convert Floating-Point from Unsigned Fraction	**evfscfuf**	**r**D,**r**B
Vector Convert Floating-Point from Unsigned Integer	**evfscfui**	**r**D,**r**B
Vector Convert Floating-Point to Signed Fraction	**evfsctsf**	**r**D,**r**B
Vector Convert Floating-Point to Signed Integer	**evfsctsi**	**r**D,**r**B
Vector Convert Floating-Point to Signed Integer with Round toward Zero	**evfsctsiz**	**r**D,**r**B

Table 3-5. SPE APU Vector Instructions (continued)

Instruction	Mnemonic	Syntax
Vector Convert Floating-Point to Unsigned Fraction	**evfsctuf**	**r**D,**r**B
Vector Convert Floating-Point to Unsigned Integer	**evfsctui**	**r**D,**r**B
Vector Convert Floating-Point to Unsigned Integer with Round toward Zero	**evfsctuiz**	**r**D,**r**B
Vector Count Leading Sign Bits Word	**evcntlsw**	**r**D,**r**A
Vector Count Leading Zeros Word	**evcntlzw**	**r**D,**r**A
Vector Divide Word Signed	**evdivws**	**r**D,**r**A,**r**B
Vector Divide Word Unsigned	**evdivwu**	**r**D,**r**A,**r**B
Vector Equivalent	**eveqv**	**r**D,**r**A,**r**B
Vector Extend Sign Byte	**evextsb**	**r**D,**r**A
Vector Extend Sign Half Word	**evextsh**	**r**D,**r**A
Vector Floating-Point Absolute Value	**evfsabs**	**r**D,**r**A
Vector Floating-Point Add	**evfsadd**	**r**D,**r**A,**r**B
Vector Floating-Point Compare Equal	**evfscmpeq**	**cr**D,**r**A,**r**B
Vector Floating-Point Compare Greater Than	**evfscmpgt**	**cr**D,**r**A,**r**B
Vector Floating-Point Compare Less Than	**evfscmplt**	**cr**D,**r**A,**r**B
Vector Floating-Point Divide	**evfsdiv**	**r**D,**r**A,**r**B
Vector Floating-Point Multiply	**evfsmul**	**r**D,**r**A,**r**B
Vector Floating-Point Negate	**evfsneg**	**r**D,**r**A
Vector Floating-Point Negative Absolute Value	**evfsnabs**	**r**D,**r**A
Vector Floating-Point Subtract	**evfssub**	**r**D,**r**A,**r**B
Vector Floating-Point Test Equal	**evfststeq**	**cr**D,**r**A,**r**B
Vector Floating-Point Test Greater Than	**evfststgt**	**cr**D,**r**A,**r**B
Vector Floating-Point Test Less Than	**evfststlt**	**cr**D,**r**A,**r**B
Vector Load Double into Half Words	**evldh**	**r**D,**d(r**A**)**
Vector Load Double into Half Words Indexed	**evldhx**	**r**D,**r**A,**r**B
Vector Load Double into Two Words	**evldw**	**r**D,**d(r**A**)**
Vector Load Double into Two Words Indexed	**evldwx**	**r**D,**r**A,**r**B
Vector Load Double Word into Double Word	**evldd**	**r**D,**d(r**A**)**
Vector Load Double Word into Double Word Indexed	**evlddx**	**r**D,**r**A,**r**B
Vector Load Half Word into Half Word Odd Signed and Splat	**evlhhossplat**	**r**D,**d(r**A**)**
Vector Load Half Word into Half Word Odd Signed and Splat Indexed	**evlhhossplatx**	**r**D,**r**A,**r**B
Vector Load Half Word into Half Word Odd Unsigned and Splat	**evlhhousplat**	**r**D,**d(r**A**)**
Vector Load Half Word into Half Word Odd Unsigned and Splat Indexed	**evlhhousplatx**	**r**D,**r**A,**r**B
Vector Load Half Word into Half Words Even and Splat	**evlhhesplat**	**r**D,**d(r**A**)**
Vector Load Half Word into Half Words Even and Splat Indexed	**evlhhesplatx**	**r**D,**r**A,**r**B

Table 3-5. SPE APU Vector Instructions (continued)

Instruction	Mnemonic	Syntax
Vector Load Word into Half Words and Splat	**evlwhsplat**	**r**D,**d(r**A**)**
Vector Load Word into Half Words and Splat Indexed	**evlwhsplatx**	**r**D,**r**A,**r**B
Vector Load Word into Half Words Odd Signed (with sign extension)	**evlwhos**	**r**D,**d(r**A**)**
Vector Load Word into Half Words Odd Signed Indexed (with sign extension)	**evlwhosx**	**r**D,**r**A,**r**B
Vector Load Word into Two Half Words Even	**evlwhe**	**r**D,**d(r**A**)**
Vector Load Word into Two Half Words Even Indexed	**evlwhex**	**r**D,**r**A,**r**B
Vector Load Word into Two Half Words Odd Unsigned (zero-extended)	**evlwhou**	**r**D,**d(r**A**)**
Vector Load Word into Two Half Words Odd Unsigned Indexed (zero-extended)	**evlwhoux**	**r**D,**r**A,**r**B
Vector Load Word into Word and Splat	**evlwwsplat**	**r**D,**d(r**A**)**
Vector Load Word into Word and Splat Indexed	**evlwwsplatx**	**r**D,**r**A,**r**B
Vector Merge High	**evmergehi**	**r**D,**r**A,**r**B
Vector Merge High/Low	**evmergehilo**	**r**D,**r**A,**r**B
Vector Merge Low	**evmergelo**	**r**D,**r**A,**r**B
Vector Merge Low/High	**evmergelohi**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Modulo, Fractional	**evmhesmf**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Modulo, Fractional and Accumulate into Words	**evmhesmfaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Modulo, Fractional and Accumulate Negative into Words	**evmhesmfanw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Modulo, Fractional, Accumulate	**evmhesmfa**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Modulo, Integer	**evmhesmi**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Modulo, Integer and Accumulate into Words	**evmhesmiaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Modulo, Integer and Accumulate Negative into Words	**evmhesmianw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Modulo, Integer, Accumulate	**evmhesmia**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Saturate, Fractional	**evmhessf**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Saturate, Fractional and Accumulate into Words	**evmhessfaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Saturate, Fractional and Accumulate Negative into Words	**evmhessfanw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Saturate, Fractional, Accumulate	**evmhessfa**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Saturate, Integer and Accumulate into Words	**evmhessiaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Signed, Saturate, Integer and Accumulate Negative into Words	**evmhessianw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Unsigned, Modulo, Integer	**evmheumi**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Unsigned, Modulo, Integer and Accumulate into Words	**evmheumiaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Unsigned, Modulo, Integer and Accumulate Negative into Words	**evmheumianw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Unsigned, Modulo, Integer, Accumulate	**evmheumia**	**r**D,**r**A,**r**B

Table 3-5. SPE APU Vector Instructions (continued)

Instruction	Mnemonic	Syntax
Vector Multiply Half Words, Even, Unsigned, Saturate, Integer and Accumulate into Words	**evmheusiaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Even, Unsigned, Saturate, Integer and Accumulate Negative into Words	**evmheusianw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Modulo, Fractional	**evmhosmf**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Modulo, Fractional and Accumulate into Words	**evmhosmfaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Modulo, Fractional and Accumulate Negative into Words	**evmhosmfanw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Modulo, Fractional, Accumulate	**evmhosmfa**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Modulo, Integer	**evmhosmi**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Modulo, Integer and Accumulate into Words	**evmhosmiaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Modulo, Integer and Accumulate Negative into Words	**evmhosmianw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Modulo, Integer, Accumulate	**evmhosmia**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Saturate, Fractional	**evmhossf**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Saturate, Fractional and Accumulate into Words	**evmhossfaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Saturate, Fractional and Accumulate Negative into Words	**evmhossfanw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Saturate, Fractional, Accumulate	**evmhossfa**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Saturate, Integer and Accumulate into Words	**evmhossiaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Signed, Saturate, Integer and Accumulate Negative into Words	**evmhossianw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer	**evmhoumi**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer and Accumulate into Words	**evmhoumiaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer and Accumulate Negative into Words	**evmhoumianw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer, Accumulate	**evmhoumia**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Unsigned, Saturate, Integer and Accumulate into Words	**evmhousiaaw**	**r**D,**r**A,**r**B
Vector Multiply Half Words, Odd, Unsigned, Saturate, Integer and Accumulate Negative into Words	**evmhousianw**	**r**D,**r**A,**r**B
Vector Multiply Word High Signed, Modulo, Fractional	**evmwhsmf**	**r**D,**r**A,**r**B
Vector Multiply Word High Signed, Modulo, Fractional and Accumulate	**evmwhsmfa**	**r**D,**r**A,**r**B
Vector Multiply Word High Signed, Modulo, Integer	**evmwhsmi**	**r**D,**r**A,**r**B
Vector Multiply Word High Signed, Modulo, Integer and Accumulate	**evmwhsmia**	**r**D,**r**A,**r**B
Vector Multiply Word High Signed, Saturate, Fractional	**evmwhssf**	**r**D,**r**A,**r**B
Vector Multiply Word High Signed, Saturate, Fractional and Accumulate	**evmwhssfa**	**r**D,**r**A,**r**B
Vector Multiply Word High Unsigned, Modulo, Integer	**evmwhumi**	**r**D,**r**A,**r**B
Vector Multiply Word High Unsigned, Modulo, Integer and Accumulate	**evmwhumia**	**r**D,**r**A,**r**B
Vector Multiply Word Low Signed, Modulo, Integer and Accumulate in Words	**evmwlsmiaaw**	**r**D,**r**A,**r**B

Table 3-5. SPE APU Vector Instructions (continued)

Instruction	Mnemonic	Syntax
Vector Multiply Word Low Signed, Modulo, Integer and Accumulate Negative in Words	**evmwlsmianw**	**r**D,**r**A,**r**B
Vector Multiply Word Low Signed, Saturate, Integer and Accumulate in Words	**evmwlssiaaw**	**r**D,**r**A,**r**B
Vector Multiply Word Low Signed, Saturate, Integer and Accumulate Negative in Words	**evmwlssianw**	**r**D,**r**A,**r**B
Vector Multiply Word Low Unsigned, Modulo, Integer	**evmwlsmi**	**r**D,**r**A,**r**B
Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate	**evmwlumia**	**r**D,**r**A,**r**B
Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate in Words	**evmwlumiaaw**	**r**D,**r**A,**r**B
Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate Negative in Words	**evmwlumianw**	**r**D,**r**A,**r**B
Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate in Words	**evmwlusiaaw**	**r**D,**r**A,**r**B
Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate Negative in Words	**evmwlusianw**	**r**D,**r**A,**r**B
Vector Multiply Word Signed, Modulo, Fractional	**evmwsmf**	**r**D,**r**A,**r**B
Vector Multiply Word Signed, Modulo, Fractional and Accumulate	**evmwsmfa**	**r**D,**r**A,**r**B
Vector Multiply Word Signed, Modulo, Fractional and Accumulate	**evmwsmfaa**	**r**D,**r**A,**r**B
Vector Multiply Word Signed, Modulo, Fractional and Accumulate Negative	**evmwsmfan**	**r**D,**r**A,**r**B
Vector Multiply Word Signed, Modulo, Integer	**evmwsmi**	**r**D,**r**A,**r**B
Vector Multiply Word Signed, Modulo, Integer and Accumulate	**evmwsmia**	**r**D,**r**A,**r**B
Vector Multiply Word Signed, Modulo, Integer and Accumulate	**evmwsmiaa**	**r**D,**r**A,**r**B
Vector Multiply Word Signed, Modulo, Integer and Accumulate Negative	**evmwsmian**	**r**D,**r**A,**r**B
Vector Multiply Word Signed, Saturate, Fractional	**evmwssf**	**r**D,**r**A,**r**B
Vector Multiply Word Signed, Saturate, Fractional and Accumulate	**evmwssfa**	**r**D,**r**A,**r**B
Vector Multiply Word Signed, Saturate, Fractional and Accumulate [2]	**evmwssfaa**	**r**D,**r**A,**r**B
Vector Multiply Word Signed, Saturate, Fractional and Accumulate Negative [2]	**evmwssfan**	**r**D,**r**A,**r**B
Vector Multiply Word Unsigned, Modulo, Integer	**evmwumi**	**r**D,**r**A,**r**B
Vector Multiply Word Unsigned, Modulo, Integer and Accumulate	**evmwumia**	**r**D,**r**A,**r**B
Vector Multiply Word Unsigned, Modulo, Integer and Accumulate	**evmwumiaa**	**r**D,**r**A,**r**B
Vector Multiply Word Unsigned, Modulo, Integer and Accumulate Negative	**evmwumian**	**r**D,**r**A,**r**B
Vector NAND	**evnand**	**r**D,**r**A,**r**B
Vector Negate	**evneg**	**r**D,**r**A
Vector NOR	**evnor**	**r**D,**r**A,**r**B
Vector OR	**evor**	**r**D,**r**A,**r**B
Vector OR with Complement	**evorc**	**r**D,**r**A,**r**B
Vector Rotate Left Word	**evrlw**	**r**D,**r**A,**r**B
Vector Rotate Left Word Immediate	**evrlwi**	**r**D,**r**A,UIMM
Vector Round Word	**evrndw**	**r**D,**r**A
Vector Select	**evsel**	**r**D,**r**A,**r**B,**cr**S
Vector Shift Left Word	**evslw**	**r**D,**r**A,**r**B

Table 3-5. SPE APU Vector Instructions (continued)

Instruction	Mnemonic	Syntax
Vector Shift Left Word Immediate	**evslwi**	**r**D,**r**A,UIMM
Vector Shift Right Word Immediate Signed	**evsrwis**	**r**D,**r**A,UIMM
Vector Shift Right Word Immediate Unsigned	**evsrwiu**	**r**D,**r**A,UIMM
Vector Shift Right Word Signed	**evsrws**	**r**D,**r**A,**r**B
Vector Shift Right Word Unsigned	**evsrwu**	**r**D,**r**A,**r**B
Vector Splat Fractional Immediate	**evsplatfi**	**r**D,SIMM
Vector Splat Immediate	**evsplati**	**r**D,SIMM
Vector Store Double of Double	**evstdd**	**r**S,**d(r**A**)**
Vector Store Double of Double Indexed	**evstddx**	**r**S,**r**A,**r**B
Vector Store Double of Four Half Words	**evstdh**	**r**S,**d(r**A**)**
Vector Store Double of Four Half Words Indexed	**evstdhx**	**r**S,**r**A,**r**B
Vector Store Double of Two Words	**evstdw**	**r**S,**d(r**A**)**
Vector Store Double of Two Words Indexed	**evstdwx**	**r**S,**r**A,**r**B
Vector Store Word of Two Half Words from Even	**evstwhe**	**r**S,**d(r**A**)**
Vector Store Word of Two Half Words from Even Indexed	**evstwhex**	**r**S,**r**A,**r**B
Vector Store Word of Two Half Words from Odd	**evstwho**	**r**S,**d(r**A**)**
Vector Store Word of Two Half Words from Odd Indexed	**evstwhox**	**r**S,**r**A,**r**B
Vector Store Word of Word from Even	**evstwwex**	**r**S,**d(r**A**)**
Vector Store Word of Word from Even Indexed	**evstwwex**	**r**S,**r**A,**r**B
Vector Store Word of Word from Odd	**evstwwo**	**r**S,**d(r**A**)**
Vector Store Word of Word from Odd Indexed	**evstwwox**	**r**S,**r**A,**r**B
Vector Subtract from Word	**evsubfw**	**r**D,**r**A,**r**B
Vector Subtract Immediate from Word	**evsubifw**	**r**D,UIMM,**r**B
Vector Subtract Signed, Modulo, Integer to Accumulator Word	**evsubfsmiaaw**	**r**D,**r**A
Vector Subtract Signed, Saturate, Integer to Accumulator Word	**evsubfssiaaw**	**r**D,**r**A
Vector Subtract Unsigned, Modulo, Integer to Accumulator Word	**evsubfumiaaw**	**r**D,**r**A
Vector Subtract Unsigned, Saturate, Integer to Accumulator Word	**evsubfusiaaw**	**r**D,**r**A
Vector XOR	**evxor**	**r**D,**r**A,**r**B

[1] An implementation can restrict the number of bits specified in a mask. The e200z6 limits it to 16 bits, which allows the user to perform bit-reversed address computations for 65536-byte samples.

[2] Although the e500 records any overflow resulting from the addition/subtraction portion of these instructions, a saturate value is not saved to **r**D or the accumulator.

3.6.4 Embedded Vector and Scalar Single-Precision Floating-Point APU Instructions

The vector and scalar SPFP APUs perform floating-point operations on single-precision operands. These operations are IEEE-compliant with software exception handlers and offer a simpler exception model than the floating-point instructions defined by the PowerPC ISA. Instead of FPRs, these instructions use GPRs to offer improved performance for converting between floating-point, integer, and fractional values. Sharing GPRs allows vector floating-point instructions to use SPE load and store instructions.

The two SPFP APUs are described as follows:

- Vector SPFP instructions operate on a vector of two 32-bit, single-precision floating-point numbers that reside in the upper and lower halves of the 64-bit GPRs. These instructions are listed in Table 3-6 alongside their scalar equivalents.
- Scalar SPFP instructions operate on single 32-bit operands that reside in the lower 32 bits of the GPRs. These instructions are listed in Table 3-6.

NOTE

Note that both the vector and scalar versions of the instructions have the same syntax.

Table 3-6. Vector and Scalar SPFP APU Floating-Point Instructions

Instruction	Mnemonic		Syntax
	Scalar	Vector	
Convert Floating-Point from Signed Fraction	**efscfsf**	**evfscfsf**	**r**D,**r**B
Convert Floating-Point from Signed Integer	**efscfsi**	**evfscfsi**	**r**D,**r**B
Convert Floating-Point from Unsigned Fraction	**efscfuf**	**evfscfuf**	**r**D,**r**B
Convert Floating-Point from Unsigned Integer	**efscfui**	**evfscfui**	**r**D,**r**B
Convert Floating-Point to Signed Fraction	**efsctsf**	**evfsctsf**	**r**D,**r**B
Convert Floating-Point to Signed Integer	**efsctsi**	**evfsctsi**	**r**D,**r**B
Convert Floating-Point to Signed Integer with Round toward Zero	**efsctsiz**	**evfsctsiz**	**r**D,**r**B
Convert Floating-Point to Unsigned Fraction	**efsctuf**	**evfsctuf**	**r**D,**r**B
Convert Floating-Point to Unsigned Integer	**efsctui**	**evfsctui**	**r**D,**r**B
Convert Floating-Point to Unsigned Integer with Round toward Zero	**efsctuiz**	**evfsctuiz**	**r**D,**r**B
Floating-Point Absolute Value	**efsabs**	**evfsabs**	**r**D,**r**A
Floating-Point Add	**efsadd**	**evfsadd**	**r**D,**r**A,**r**B
Floating-Point Compare Equal	**efscmpeq**	**evfscmpeq**	**cr**D,**r**A,**r**B
Floating-Point Compare Greater Than	**efscmpgt**	**evfscmpgt**	**cr**D,**r**A,**r**B
Floating-Point Compare Less Than	**efscmplt**	**evfscmplt**	**cr**D,**r**A,**r**B
Floating-Point Divide	**efsdiv**	**evfsdiv**	**r**D,**r**A,**r**B

Table 3-6. Vector and Scalar SPFP APU Floating-Point Instructions (continued)

Instruction	Mnemonic		Syntax
	Scalar	Vector	
Floating-Point Multiply	**efsmul**	**evfsmul**	**r**D,**r**A,**r**B
Floating-Point Negate	**efsneg**	**evfsneg**	**r**D,**r**A
Floating-Point Negative Absolute Value	**efsnabs**	**evfsnabs**	**r**D,**r**A
Floating-Point Subtract	**efssub**	**evfssub**	**r**D,**r**A,**r**B
Floating-Point Test Equal	**efststeq**	**evfststeq**	**cr**D,**r**A,**r**B
Floating-Point Test Greater Than	**efststgt**	**evfststgt**	**cr**D,**r**A,**r**B
Floating-Point Test Less Than	**efststlt**	**evfststlt**	**cr**D,**r**A,**r**B

3.6.4.1 Options for Embedded Floating-Point APU Implementations

Table 3-7 lists implementation options allowed by the embedded floating-point architecture and how the e200z6 handles those options.

Table 3-7. Embedded Floating–Point APU Options

Option	e200z6 Implementation
Overflow and underflow conditions may be signaled by doing exponent evaluation of the operation. If by examining the exponents, an overflow or underflow could occur, the implementation may choose to signal an overflow or underflow. It is recommended that future implementations do not use this estimation and signal overflow or underflow when they actually occur.	The e200z6 follows the recommendation and doesn't use the estimation.
If an operand for a calculation or conversion is denormalized, the implementation may choose to use a same-signed zero value in place of the denormalized operand.	The e200z6 uses a same-signed zero value in place of the denormalized operand.
+Infinity and -Infinity rounding modes are not required to be handled by an implementation. If an implementation does not support ±Infinity rounding modes and the rounding mode is set to be +Infinity or -Infinity, an embedded floating-point round interrupt occurs after every floating-point instruction for which rounding may occur, regardless of the value of FINXE, unless an embedded floating-point data interrupt also occurs and is taken.	The e200z6 supports rounding to ±Infinity.
For absolute value, negate, negative absolute value operations, an implementation may choose either to simply perform the sign bit operation ignoring exceptions or to compute the operation and handle exceptions and saturation where appropriate.	The sign bit operation is performed; exceptions are not taken.
SPEFSCR FGH and FXH bits are undefined upon the completion of a scalar floating-point operation. An implementation may choose to zero them or leave them unchanged.	The e200z6 always clears these bits for such operations.
An implementation may choose to only implement sticky bit setting by hardware for FDBZS and FINXS allowing software to manage the other sticky bits. It is recommended that all future implementations implement all sticky bit setting in hardware.	The e200z6 implements all sticky bit settings in hardware.

3.7 Unimplemented SPRs and Read-Only SPRs

The e200z6 fully decodes the SPR field of **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 **mtspr**, 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.

3.8 Invalid Instruction Forms

Table 3-8 describes invalid instruction forms.

Table 3-8. Invalid Instruction Forms

Instructions	Descriptions
Load and store with update instructions	Book E defines as an invalid form the case when a load with update instruction specifies the same register in the **r**D and **r**A field of the instruction. For this invalid case, the e200z6 core performs the instruction and updates the register with the load data. In addition, if **r**A = 0 for any load or store with update instruction, the e200z6 core updates **r**A (GPR0).
Load Multiple Word (**lmw**) instruction	Book E defines as invalid any form of the **lmw** instruction in which **r**A is in the range of registers to be loaded, including the case in which **r**A = 0. On the e200z6, invalid forms of **lmw** execute as follows: • Case 1: **r**A is in the range of **r**D, **r**A ≠ 0. In this case address generation for individual loads to register targets is done using the architectural value of **r**A which existed when beginning execution of this **lmw** instruction. **r**A 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 **r**A has been overwritten. • Case 2: rA = 0 and **r**D = 0. In this case address generation for all loads to register targets **r**D = 0 to **r**D = 31 is done substituting the value of 0 for **r**A.
Branch Conditional to Count Register [and Link] instructions	Book E defines as invalid any **bcctr** or **bcctrl** instruction that specifies the decrement and test CTR (BO[2] = 0) option. The e200z6 executes instructions with these invalid forms by decrementing the CTR and branching 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.
Instructions with non-zero reserved fields	Book E defines certain bit fields in various instructions as reserved and specifies that these fields be set to zero. Following the Book E recommendation, the e200z6 ignores the value of the reserved field (bit 31) in X-form integer load and store instructions. The e200z6 ignores the value of the reserved ‘z’ bits in the BO field of branch instructions. For all other instructions, the e200z6 generates an illegal instruction exception if a reserved field is non-zero.

3.9 Instruction Summary

Table 3-9 and Table 3-10 list all 32-bit Book E instructions, as well as e200z6-specific instructions, sorted by mnemonic. Instructions not listed here are not supported by the e200z6 core and signal an illegal, unimplemented, or floating-point unavailable exception. Implementation-dependent instructions are noted with a footnote.

3.9.1 Instruction Index Sorted by Mnemonic

Table 3-9 lists instructions by mnemonic.

Table 3-9. Instructions Sorted by Mnemonic

Format	Opcode		Mnemonic	Instruction
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)		
X	011111	01000 01010 0	**add**	Add
X	011111	01000 01010 1	**add.**	Add & record CR
X	011111	00000 01010 0	**addc**	Add Carrying
X	011111	00000 01010 1	**addc.**	Add Carrying & record CR
X	011111	10000 01010 0	**addco**	Add Carrying & record OV
X	011111	10000 01010 1	**addco.**	Add Carrying & record OV & CR
X	011111	00100 01010 0	**adde**	Add Extended with CA
X	011111	00100 01010 1	**adde.**	Add Extended with CA & record CR
X	011111	10100 01010 0	**addeo**	Add Extended with CA & record OV
X	011111	10100 01010 1	**addeo.**	Add Extended with CA & record OV & CR
D	001110	----- ----- -	**addi**	Add Immediate
D	001100	----- ----- -	**addic**	Add Immediate Carrying
D	001101	----- ----- -	**addic.**	Add Immediate Carrying & record CR
D	001111	----- ----- -	**addis**	Add Immediate Shifted
X	011111	00111 01010 0	**addme**	Add to Minus One Extended with CA
X	011111	00111 01010 1	**addme.**	Add to Minus One Extended with CA & record CR
X	011111	10111 01010 0	**addmeo**	Add to Minus One Extended with CA & record OV
X	011111	10111 01010 1	**addmeo.**	Add to Minus One Extended with CA & record OV & CR
X	011111	11000 01010 0	**addo**	Add & record OV
X	011111	11000 01010 1	**addo.**	Add & record OV & CR

Legend:
- Don't care, usually part of an operand field
/ Reserved bit, invalid instruction form if encoded as 1
? Allocated for implementation-dependent use.

Table 3-9. Instructions Sorted by Mnemonic (continued)

Format	Opcode		Mnemonic	Instruction
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)		
X	011111	00110 01010 0	**addze**	Add to Zero Extended with CA
X	011111	00110 01010 1	**addze.**	Add to Zero Extended with CA & record CR
X	011111	10110 01010 0	**addzeo**	Add to Zero Extended with CA & record OV
X	011111	10110 01010 1	**addzeo.**	Add to Zero Extended with CA & record OV & CR
X	011111	00000 11100 0	**and**	AND
X	011111	00000 11100 1	**and.**	AND & record CR
X	011111	00001 11100 0	**andc**	AND with Complement
X	011111	00001 11100 1	**andc.**	AND with Complement & record CR
D	011100	----- ----- -	**andi.**	AND Immediate & record CR
D	011101	----- ----- -	**andis.**	AND Immediate Shifted & record CR
I	010010	----- ----0 0	**b**	Branch
I	010010	----- ----1 0	**ba**	Branch Absolute
B	010000	----- ----0 0	**bc**	Branch Conditional
B	010000	----- ----1 0	**bca**	Branch Conditional Absolute
XL	010011	10000 10000 0	**bcctr**	Branch Conditional to Count Register
XL	010011	10000 10000 1	**bcctrl**	Branch Conditional to Count Register & Link
B	010000	----- ----0 1	**bcl**	Branch Conditional & Link
B	010000	----- ----1 1	**bcla**	Branch Conditional & Link Absolute
XL	010011	00000 10000 0	**bclr**	Branch Conditional to Link Register
XL	010011	00000 10000 1	**bclrl**	Branch Conditional to Link Register & Link
I	010010	----- ----0 1	**bl**	Branch & Link
I	010010	----- ----1 1	**bla**	Branch & Link Absolute
X	011111	00000 00000 /	**cmp**	Compare
D	001011	----- ----- -	**cmpi**	Compare Immediate
X	011111	00001 00000 /	**cmpl**	Compare Logical
D	001010	----- ----- -	**cmpli**	Compare Logical Immediate
X	011111	00000 11010 0	**cntlzw**	Count Leading Zeros Word
X	011111	00000 11010 1	**cntlzw.**	Count Leading Zeros Word & record CR
XL	010011	01000 00001 /	**crand**	Condition Register AND

Legend:
- Don't care, usually part of an operand field
/ Reserved bit, invalid instruction form if encoded as 1
? Allocated for implementation-dependent use.

Table 3-9. Instructions Sorted by Mnemonic (continued)

Format	Opcode		Mnemonic	Instruction
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)		
XL	010011	00100 00001 /	**crandc**	Condition Register AND with Complement
XL	010011	01001 00001 /	**creqv**	Condition Register Equivalent
XL	010011	00111 00001 /	**crnand**	Condition Register NAND
XL	010011	00001 00001 /	**crnor**	Condition Register NOR
XL	010011	01110 00001 /	**cror**	Condition Register OR
XL	010011	01101 00001 /	**crorc**	Condition Register OR with Complement
XL	010011	00110 00001 /	**crxor**	Condition Register XOR
X	011111	10111 10110 /	**dcba**	Data Cache Block Allocate
X	011111	00010 10110 /	**dcbf**	Data Cache Block Flush
X	011111	01110 10110 /	**dcbi**	Data Cache Block Invalidate
X	011111	01100 00110 /	**dcblc** [1]	Data Cache Block Lock Clear
X	011111	00001 10110 /	**dcbst**	Data Cache Block Store
X	011111	01000 10110 /	**dcbt**	Data Cache Block Touch
X	011111	00101 00110 /	**dcbtlst**[1]	Data Cache Block Touch and Lock Set
X	011111	00111 10110 /	**dcbtst**	Data Cache Block Touch for Store
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
X	011111	01111 01011 0	**divw**	Divide Word
X	011111	01111 01011 1	**divw.**	Divide Word & record CR
X	011111	11111 01011 0	**divwo**	Divide Word & record OV
X	011111	11111 01011 1	**divwo.**	Divide Word & record OV & CR
X	011111	01110 01011 0	**divwu**	Divide Word Unsigned
X	011111	01110 01011 1	**divwu.**	Divide Word Unsigned & record CR
X	011111	11110 01011 0	**divwuo**	Divide Word Unsigned & record OV
X	011111	11110 01011 1	**divwuo.**	Divide Word Unsigned & record OV & CR
X	011111	01000 11100 0	**eqv**	Equivalent
X	011111	01000 11100 1	**eqv.**	Equivalent & record CR
X	011111	11101 11010 0	**extsb**	Extend Sign Byte
X	011111	11101 11010 1	**extsb.**	Extend Sign Byte & record CR

Legend:
- – Don't care, usually part of an operand field
- / Reserved bit, invalid instruction form if encoded as 1
- ? Allocated for implementation-dependent use.

Table 3-9. Instructions Sorted by Mnemonic (continued)

Format	Opcode		Mnemonic	Instruction
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)		
X	011111	11100 11010 0	**extsh**	Extend Sign Half Word
X	011111	11100 11010 1	**extsh.**	Extend Sign Half Word & record CR
X	011111	11110 10110 /	**icbi**	Instruction Cache Block Invalidate
X	011111	00111 00110 /	**icblc**[1]	Instruction Cache Block Lock Clear
X	011111	00000 10110 /	**icbt**	Instruction Cache Block Touch
X	011111	01111 00110 /	**icbtls**[1]	Instruction Cache Block Touch and Lock Set
X	011111	----- 01111 /	**isel**[2]	Integer Select
XL	010011	00100 10110 /	**isync**	Instruction Synchronize
D	100010	----- ----- -	**lbz**	Load Byte & Zero
D	100011	----- ----- -	**lbzu**	Load Byte & Zero with Update
X	011111	00011 10111 /	**lbzux**	Load Byte & Zero with Update Indexed
X	011111	00010 10111 /	**lbzx**	Load Byte & Zero Indexed
D	101010	----- ----- -	**lha**	Load Half Word Algebraic
D	101011	----- ----- -	**lhau**	Load Half Word Algebraic with Update
X	011111	01011 10111 /	**lhaux**	Load Half Word Algebraic with Update Indexed
X	011111	01010 10111 /	**lhax**	Load Half Word Algebraic Indexed
X	011111	11000 10110 /	**lhbrx**	Load Half Word Byte-Reverse Indexed
D	101000	----- ----- -	**lhz**	Load Half Word & Zero
D	101001	----- ----- -	**lhzu**	Load Half Word & Zero with Update
X	011111	01001 10111 /	**lhzux**	Load Half Word & Zero with Update Indexed
X	011111	01000 10111 /	**lhzx**	Load Half Word & Zero Indexed
D	101110	----- ----- -	**lmw**	Load Multiple Word
X	011111	00000 10100 /	**lwarx**[3]	Load Word & Reserve Indexed
X	011111	10000 10110 /	**lwbrx**	Load Word Byte-Reverse Indexed
D	100000	----- ----- -	**lwz**	Load Word & Zero
D	100001	----- ----- -	**lwzu**	Load Word & Zero with Update
X	011111	00001 10111 /	**lwzux**	Load Word & Zero with Update Indexed
X	011111	00000 10111 /	**lwzx**	Load Word & Zero Indexed
X	011111	11010 10110 /	**mbar**[3]	Memory Barrier

Legend:
- – Don't care, usually part of an operand field
- / Reserved bit, invalid instruction form if encoded as 1
- ? Allocated for implementation-dependent use.

Table 3-9. Instructions Sorted by Mnemonic (continued)

Format	Opcode		Mnemonic	Instruction
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)		
XL	010011	00000 00000 /	**mcrf**	Move Condition Register Field
X	011111	10000 00000 /	**mcrxr**	Move to Condition Register from XER
X	011111	00000 10011 /	**mfcr**	Move From Condition Register
X	011111	00010 10011 /	**mfmsr**	Move From Machine State Register
XFX	011111	01010 10011 /	**mfspr**	Move From Special Purpose Register
X	011111	10010 10110 /	**msync**[3]	Memory Synchronize
XFX	011111	00100 10000 /	**mtcrf**	Move To Condition Register Fields
X	011111	00100 10010 /	**mtmsr**	Move To Machine State Register
XFX	011111	01110 10011 /	**mtspr**	Move To Special Purpose Register
X	011111	/0010 01011 0	**mulhw**	Multiply High Word
X	011111	/0010 01011 1	**mulhw.**	Multiply High Word & record CR
X	011111	/0000 01011 0	**mulhwu**	Multiply High Word Unsigned
X	011111	/0000 01011 1	**mulhwu.**	Multiply High Word Unsigned & record CR
D	000111	----- ----- -	**mulli**	Multiply Low Immediate
X	011111	00111 01011 0	**mullw**	Multiply Low Word
X	011111	00111 01011 1	**mullw.**	Multiply Low Word & record CR
X	011111	10111 01011 0	**mullwo**	Multiply Low Word & record OV
X	011111	10111 01011 1	**mullwo.**	Multiply Low Word & record OV & CR
X	011111	01110 11100 0	**nand**	NAND
X	011111	01110 11100 1	**nand.**	NAND & record CR
X	011111	00011 01000 0	**neg**	Negate
X	011111	00011 01000 1	**neg.**	Negate & record CR
X	011111	10011 01000 0	**nego**	Negate & record OV
X	011111	10011 01000 1	**nego.**	Negate & record OV & record CR
X	011111	00011 11100 0	**nor**	NOR
X	011111	00011 11100 1	**nor.**	NOR & record CR
X	011111	01101 11100 0	**or**	OR
X	011111	01101 11100 1	**or.**	OR & record CR
X	011111	01100 11100 0	**orc**	OR with Complement

Legend:
- Don't care, usually part of an operand field
/ Reserved bit, invalid instruction form if encoded as 1
? Allocated for implementation-dependent use.

Table 3-9. Instructions Sorted by Mnemonic (continued)

Format	Opcode		Mnemonic	Instruction
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)		
X	011111	01100 11100 1	**orc.**	OR with Complement & record CR
D	011000	----- ----- -	**ori**	OR Immediate
D	011001	----- ----- -	**oris**	OR Immediate Shifted
XL	010011	00001 10011 /	**rfci**	Return From Critical Interrupt
XL	010011	00001 00111 /	**rfdi** [4]	Return From Debug Interrupt
XL	010011	00001 10010 /	**rfi**	Return From Interrupt
M	010100	----- ----- 0	**rlwimi**	Rotate Left Word Immed then Mask Insert
M	010100	----- ----- 1	**rlwimi.**	Rotate Left Word Immed then Mask Insert & record CR
M	010101	----- ----- 0	**rlwinm**	Rotate Left Word Immed then AND with Mask
M	010101	----- ----- 1	**rlwinm.**	Rotate Left Word Immed then AND with Mask & record CR
M	010111	----- ----- 0	**rlwnm**	Rotate Left Word then AND with Mask
M	010111	----- ----- 1	**rlwnm.**	Rotate Left Word then AND with Mask & record CR
SC	010001	///// ////1 /	**sc**	System Call
X	011111	00000 11000 0	**slw**	Shift Left Word
X	011111	00000 11000 1	**slw.**	Shift Left Word & record CR
X	011111	11000 11000 0	**sraw**	Shift Right Algebraic Word
X	011111	11000 11000 1	**sraw.**	Shift Right Algebraic Word & record CR
X	011111	11001 11000 0	**srawi**	Shift Right Algebraic Word Immediate
X	011111	11001 11000 1	**srawi.**	Shift Right Algebraic Word Immediate & record CR
X	011111	10000 11000 0	**srw**	Shift Right Word
X	011111	10000 11000 1	**srw.**	Shift Right Word & record CR
D	100110	----- ----- -	**stb**	Store Byte
D	100111	----- ----- -	**stbu**	Store Byte with Update
X	011111	00111 10111 /	**stbux**	Store Byte with Update Indexed
X	011111	00110 10111 /	**stbx**	Store Byte Indexed
D	101100	----- ----- -	**sth**	Store Half Word
X	011111	11100 10110 /	**sthbrx**	Store Half Word Byte-Reverse Indexed
D	101101	----- ----- -	**sthu**	Store Half Word with Update
X	011111	01101 10111 /	**sthux**	Store Half Word with Update Indexed

Legend:
- Don't care, usually part of an operand field
/ Reserved bit, invalid instruction form if encoded as 1
? Allocated for implementation-dependent use.

Table 3-9. Instructions Sorted by Mnemonic (continued)

Format	Opcode		Mnemonic	Instruction
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)		
X	011111	01100 10111 /	**sthx**	Store Half Word Indexed
D	101111	----- ----- -	**stmw**	Store Multiple Word
D	100100	----- ----- -	**stw**	Store Word
X	011111	10100 10110 /	**stwbrx**	Store Word Byte-Reverse Indexed
X	011111	00100 10110 1	**stwcx.**[3]	Store Word Conditional Indexed & record CR
D	100101	----- ----- -	**stwu**	Store Word with Update
X	011111	00101 10111 /	**stwux**	Store Word with Update Indexed
X	011111	00100 10111 /	**stwx**	Store Word Indexed
X	011111	00001 01000 0	**subf**	Subtract From
X	011111	00001 01000 1	**subf.**	Subtract From & record CR
X	011111	00000 01000 0	**subfc**	Subtract From Carrying
X	011111	00000 01000 1	**subfc.**	Subtract From Carrying & record CR
X	011111	10000 01000 0	**subfco**	Subtract From Carrying & record OV
X	011111	10000 01000 1	**subfco.**	Subtract From Carrying & record OV & CR
X	011111	00100 01000 0	**subfe**	Subtract From Extended with CA
X	011111	00100 01000 1	**subfe.**	Subtract From Extended with CA & record CR
X	011111	10100 01000 0	**subfeo**	Subtract From Extended with CA & record OV
X	011111	10100 01000 1	**subfeo.**	Subtract From Extended with CA & record OV & CR
D	001000	----- ----- -	**subfic**	Subtract From Immediate Carrying
X	011111	00111 01000 0	**subfme**	Subtract From Minus One Extended with CA
X	011111	00111 01000 1	**subfme.**	Subtract From Minus One Extended with CA & record CR
X	011111	10111 01000 0	**subfmeo**	Subtract From Minus One Extended with CA & record OV
X	011111	10111 01000 1	**subfmeo.**	Subtract From Minus One Extended with CA & record OV & CR
X	011111	10001 01000 0	**subfo**	Subtract From & record OV
X	011111	10001 01000 1	**subfo.**	Subtract From & record OV & CR
X	011111	00110 01000 0	**subfze**	Subtract From Zero Extended with CA
X	011111	00110 01000 1	**subfze.**	Subtract From Zero Extended with CA & record CR
X	011111	10110 01000 0	**subfzeo**	Subtract From Zero Extended with CA & record OV

Legend:
- Don't care, usually part of an operand field
/ Reserved bit, invalid instruction form if encoded as 1
? Allocated for implementation-dependent use.

Table 3-9. Instructions Sorted by Mnemonic (continued)

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

Legend:
- Don't care, usually part of an operand field
/ Reserved bit, invalid instruction form if encoded as 1
? Allocated for implementation-dependent use.

1 Motorola Book E cache locking APU, refer to Section 4.12, "Cache Line Locking/Unlocking APU."

2 Motorola Book E integer select APU, refer to Section 3.6.1, "Integer Select APU

3 See Section 3.3, "Memory Synchronization and Reservation Instructions"

4 See Section 3.6.2, "Debug APU"

3.9.2 Instruction Index Sorted by Opcode

Table 3-10 lists instructions by opcode.

Table 3-10. Instructions Sorted by Opcode

Format	Opcode		Mnemonic	Instruction
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)		
D	000011	----- ----- -	**twi**	Trap Word Immediate
D	000111	----- ----- -	**mulli**	Multiply Low Immediate
D	001000	----- ----- -	**subfic**	Subtract From Immediate Carrying
D	001010	----- ----- -	**cmpli**	Compare Logical Immediate
D	001011	----- ----- -	**cmpi**	Compare Immediate
D	001100	----- ----- -	**addic**	Add Immediate Carrying
D	001101	----- ----- -	**addic.**	Add Immediate Carrying & record CR
D	001110	----- ----- -	**addi**	Add Immediate
D	001111	----- ----- -	**addis**	Add Immediate Shifted
B	010000	----- ----0 0	**bc**	Branch Conditional
B	010000	----- ----0 1	**bcl**	Branch Conditional & Link
B	010000	----- ----1 0	**bca**	Branch Conditional Absolute
B	010000	----- ----1 1	**bcla**	Branch Conditional & Link Absolute
SC	010001	///// ////1 /	**sc**	System Call
I	010010	----- ----0 0	**b**	Branch
I	010010	----- ----0 1	**bl**	Branch & Link
I	010010	----- ----1 0	**ba**	Branch Absolute
I	010010	----- ----1 1	**bla**	Branch & Link Absolute
XL	010011	00000 00000 /	**mcrf**	Move Condition Register Field
XL	010011	00000 10000 0	**bclr**	Branch Conditional to Link Register
XL	010011	00000 10000 1	**bclrl**	Branch Conditional to Link Register & Link
XL	010011	00001 00001 /	**crnor**	Condition Register NOR
XL	010011	00001 00111 /	**rfdi**	Return From Debug Interrupt
XL	010011	00001 10010 /	**rfi**	Return From Interrupt
XL	010011	00001 10011 /	**rfci**	Return From Critical Interrupt
XL	010011	00100 00001 /	**crandc**	Condition Register AND with Complement
XL	010011	00100 10110 /	**isync**	Instruction Synchronize
XL	010011	00110 00001 /	**crxor**	Condition Register XOR
XL	010011	00111 00001 /	**crnand**	Condition Register NAND

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 3-10. Instructions Sorted by Opcode (continued)

Format	Opcode		Mnemonic	Instruction
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)		
XL	010011	01000 00001 /	**crand**	Condition Register AND
XL	010011	01001 00001 /	**creqv**	Condition Register Equivalent
XL	010011	01101 00001 /	**crorc**	Condition Register OR with Complement
XL	010011	01110 00001 /	**cror**	Condition Register OR
XL	010011	10000 10000 0	**bcctr**	Branch Conditional to Count Register
XL	010011	10000 10000 1	**bcctrl**	Branch Conditional to Count Register & Link
M	010100	----- ----- 0	**rlwimi**	Rotate Left Word Immed then Mask Insert
M	010100	----- ----- 1	**rlwimi.**	Rotate Left Word Immed then Mask Insert & record CR
M	010101	----- ----- 0	**rlwinm**	Rotate Left Word Immed then AND with Mask
M	010101	----- ----- 1	**rlwinm.**	Rotate Left Word Immed then AND with Mask & record CR
M	010111	----- ----- 0	**rlwnm**	Rotate Left Word then AND with Mask
M	010111	----- ----- 1	**rlwnm.**	Rotate Left Word then AND with Mask & record CR
D	011000	----- ----- -	**ori**	OR Immediate
D	011001	----- ----- -	**oris**	OR Immediate Shifted
D	011010	----- ----- -	**xori**	XOR Immediate
D	011011	----- ----- -	**xoris**	XOR Immediate Shifted
D	011100	----- ----- -	**andi.**	AND Immediate & record CR
D	011101	----- ----- -	**andis.**	AND Immediate Shifted & record CR
X	011111	----- 01111 /	**isel**	Integer Select
X	011111	00000 00000 /	**cmp**	Compare
X	011111	00000 00100 /	**tw**	Trap Word
X	011111	00000 01000 0	**subfc**	Subtract From Carrying
X	011111	00000 01000 1	**subfc.**	Subtract From Carrying & record CR
X	011111	00000 01010 0	**addc**	Add Carrying
X	011111	00000 01010 1	**addc.**	Add Carrying & record CR
X	011111	/0000 01011 0	**mulhwu**	Multiply High Word Unsigned
X	011111	/0000 01011 1	**mulhwu.**	Multiply High Word Unsigned & record CR
X	011111	00000 10011 /	**mfcr**	Move From Condition Register
X	011111	00000 10100 /	**lwarx**	Load Word & Reserve Indexed

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 3-10. Instructions Sorted by Opcode (continued)

Format	Opcode		Mnemonic	Instruction
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)		
X	011111	00000 10110 /	**icbt**	Instruction Cache Block Touch
X	011111	00000 10111 /	**lwzx**	Load Word & Zero Indexed
X	011111	00000 11000 0	**slw**	Shift Left Word
X	011111	00000 11000 1	**slw.**	Shift Left Word & record CR
X	011111	00000 11010 0	**cntlzw**	Count Leading Zeros Word
X	011111	00000 11010 1	**cntlzw.**	Count Leading Zeros Word & record CR
X	011111	00000 11100 0	**and**	AND
X	011111	00000 11100 1	**and.**	AND & record CR
X	011111	00001 00000 /	**cmpl**	Compare Logical
X	011111	00001 01000 0	**subf**	Subtract From
X	011111	00001 01000 1	**subf.**	Subtract From & record CR
X	011111	00001 10110 /	**dcbst**	Data Cache Block Store
X	011111	00001 10111 /	**lwzux**	Load Word & Zero with Update Indexed
X	011111	00001 11100 0	**andc**	AND with Complement
X	011111	00001 11100 1	**andc.**	AND with Complement & record CR
X	011111	/0010 01011 0	**mulhw**	Multiply High Word
X	011111	/0010 01011 1	**mulhw.**	Multiply High Word & record CR
X	011111	00010 10011 /	**mfmsr**	Move From Machine State Register
X	011111	00010 10110 /	**dcbf**	Data Cache Block Flush
X	011111	00010 10111 /	**lbzx**	Load Byte & Zero Indexed
X	011111	00011 01000 0	**neg**	Negate
X	011111	00011 01000 1	**neg.**	Negate & record CR
X	011111	00011 10111 /	**lbzux**	Load Byte & Zero with Update Indexed
X	011111	00011 11100 0	**nor**	NOR
X	011111	00011 11100 1	**nor.**	NOR & record CR
X	011111	00100 00011 /	**wrtee**	Write External Enable
X	011111	00100 00110 /	**dcbtstls**[1]	Data Cache Block Touch for Store and Lock Set
X	011111	00100 01000 0	**subfe**	Subtract From Extended with CA
X	011111	00100 01000 1	**subfe.**	Subtract From Extended with CA & record CR

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 3-10. Instructions Sorted by Opcode (continued)

Format	Opcode		Mnemonic	Instruction
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)		
X	011111	00100 01010 0	**adde**	Add Extended with CA
X	011111	00100 01010 1	**adde.**	Add Extended with CA & record CR
XFX	011111	00100 10000 /	**mtcrf**	Move To Condition Register Fields
X	011111	00100 10010 /	**mtmsr**	Move To Machine State Register
X	011111	00100 10110 1	**stwcx.**	Store Word Conditional Indexed & record CR
X	011111	00100 10111 /	**stwx**	Store Word Indexed
X	011111	00101 00011 /	**wrteei**	Write External Enable Immediate
X	011111	00101 00110 /	**dcbtls**[1]	Data Cache Block Touch and Lock Set
X	011111	00101 10111 /	**stwux**	Store Word with Update Indexed
X	011111	00110 01000 0	**subfze**	Subtract From Zero Extended with CA
X	011111	00110 01000 1	**subfze.**	Subtract From Zero Extended with CA & record CR
X	011111	00110 01010 0	**addze**	Add to Zero Extended with CA
X	011111	00110 01010 1	**addze.**	Add to Zero Extended with CA & record CR
X	011111	00110 10111 /	**stbx**	Store Byte Indexed
X	011111	00111 00110 /	**icblc**[1]	Instruction Cache Block Lock Clear
X	011111	00111 01000 0	**subfme**	Subtract From Minus One Extended with CA
X	011111	00111 01000 1	**subfme.**	Subtract From Minus One Extended with CA & record CR
X	011111	00111 01010 0	**addme**	Add to Minus One Extended with CA
X	011111	00111 01010 1	**addme.**	Add to Minus One Extended with CA & record CR
X	011111	00111 01011 0	**mullw**	Multiply Low Word
X	011111	00111 01011 1	**mullw.**	Multiply Low Word & record CR
X	011111	00111 10110 /	**dcbtst**	Data Cache Block Touch for Store
X	011111	00111 10111 /	**stbux**	Store Byte with Update Indexed
X	011111	01000 01010 0	**add**	Add
X	011111	01000 01010 1	**add.**	Add & record CR
X	011111	01000 10110 /	**dcbt**	Data Cache Block Touch
X	011111	01000 10111 /	**lhzx**	Load Half Word & Zero Indexed
X	011111	01000 11100 0	**eqv**	Equivalent
X	011111	01000 11100 1	**eqv.**	Equivalent & record CR

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 3-10. Instructions Sorted by Opcode (continued)

Format	Opcode		Mnemonic	Instruction
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)		
X	011111	01001 10111 /	**lhzux**	Load Half Word & Zero with Update Indexed
X	011111	01001 11100 0	**xor**	XOR
X	011111	01001 11100 1	**xor.**	XOR & record CR
XFX	011111	01010 10011 /	**mfspr**	Move From Special Purpose Register
X	011111	01010 10111 /	**lhax**	Load Half Word Algebraic Indexed
X	011111	01011 10111 /	**lhaux**	Load Half Word Algebraic with Update Indexed
X	011111	01100 00110 /	**dcblc** [1]	Data Cache Block Lock Clear
X	011111	01100 10111 /	**sthx**	Store Half Word Indexed
X	011111	01100 11100 0	**orc**	OR with Complement
X	011111	01100 11100 1	**orc.**	OR with Complement & record CR
X	011111	01101 10111 /	**sthux**	Store Half Word with Update Indexed
X	011111	01101 11100 0	**or**	OR
X	011111	01101 11100 1	**or.**	OR & record CR
X	011111	01110 01011 0	**divwu**	Divide Word Unsigned
X	011111	01110 01011 1	**divwu.**	Divide Word Unsigned & record CR
XFX	011111	01110 10011 /	**mtspr**	Move To Special Purpose Register
X	011111	01110 10110 /	**dcbi**	Data Cache Block Invalidate
X	011111	01110 11100 0	**nand**	NAND
X	011111	01110 11100 1	**nand.**	NAND & record CR
X	011111	01111 00110 /	**icbtls**[1]	Instruction Cache Block Touch and Lock Set
X	011111	01111 01011 0	**divw**	Divide Word
X	011111	01111 01011 1	**divw.**	Divide Word & record CR
X	011111	10000 00000 /	**mcrxr**	Move to Condition Register from XER
X	011111	10000 01000 0	**subfco**	Subtract From Carrying & record OV
X	011111	10000 01000 1	**subfco.**	Subtract From Carrying & record OV & CR
X	011111	10000 01010 0	**addco**	Add Carrying & record OV
X	011111	10000 01010 1	**addco.**	Add Carrying & record OV & CR
X	011111	10000 10110 /	**lwbrx**	Load Word Byte-Reverse Indexed
X	011111	10000 11000 0	**srw**	Shift Right Word

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 3-10. Instructions Sorted by Opcode (continued)

Format	Opcode		Mnemonic	Instruction
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)		
X	011111	10000 11000 1	**srw.**	Shift Right Word & record CR
X	011111	10001 01000 0	**subfo**	Subtract From & record OV
X	011111	10001 01000 1	**subfo.**	Subtract From & record OV & CR
X	011111	10001 10110 /	**tlbsync**	TLB Synchronize
X	011111	10010 10110 /	**msync**	Memory Synchronize
X	011111	10011 01000 0	**nego**	Negate & record OV
X	011111	10011 01000 1	**nego.**	Negate & record OV & record CR
X	011111	10100 01000 0	**subfeo**	Subtract From Extended with CA & record OV
X	011111	10100 01000 1	**subfeo.**	Subtract From Extended with CA & record OV & CR
X	011111	10100 01010 0	**addeo**	Add Extended with CA & record OV
X	011111	10100 01010 1	**addeo.**	Add Extended with CA & record OV & CR
X	011111	10100 10110 /	**stwbrx**	Store Word Byte-Reverse Indexed
X	011111	10110 01000 0	**subfzeo**	Subtract From Zero Extended with CA & record OV
X	011111	10110 01000 1	**subfzeo.**	Subtract From Zero Extended with CA & record OV & CR
X	011111	10110 01010 0	**addzeo**	Add to Zero Extended with CA & record OV
X	011111	10110 01010 1	**addzeo.**	Add to Zero Extended with CA & record OV & CR
X	011111	10111 01000 0	**subfmeo**	Subtract From Minus One Extended with CA & record OV
X	011111	10111 01000 1	**subfmeo.**	Subtract From Minus One Extended with CA & record OV & CR
X	011111	10111 01010 0	**addmeo**	Add to Minus One Extended with CA & record OV
X	011111	10111 01010 1	**addmeo.**	Add to Minus One Extended with CA & record OV & CR
X	011111	10111 01011 0	**mullwo**	Multiply Low Word & record OV
X	011111	10111 01011 1	**mullwo.**	Multiply Low Word & record OV & CR
X	011111	10111 10110 /	**dcba**	Data Cache Block Allocate
X	011111	11000 01010 0	**addo**	Add & record OV
X	011111	11000 01010 1	**addo.**	Add & record OV & CR
X	011111	11000 10010 /	**tlbivax**	TLB Invalidate Virtual Address Indexed
X	011111	11000 10110 /	**lhbrx**	Load Half Word Byte-Reverse Indexed
X	011111	11000 11000 0	**sraw**	Shift Right Algebraic Word
X	011111	11000 11000 1	**sraw.**	Shift Right Algebraic Word & record CR

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 3-10. Instructions Sorted by Opcode (continued)

Format	Opcode		Mnemonic	Instruction
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)		
X	011111	11001 11000 0	**srawi**	Shift Right Algebraic Word Immediate
X	011111	11001 11000 1	**srawi.**	Shift Right Algebraic Word Immediate & record CR
X	011111	11010 10110 /	**mbar**	Memory Barrier
X	011111	11100 10010 /	**tlbsx**	TLB Search Indexed
X	011111	11100 10110 /	**sthbrx**	Store Half Word Byte-Reverse Indexed
X	011111	11100 11010 0	**extsh**	Extend Sign Half Word
X	011111	11100 11010 1	**extsh.**	Extend Sign Half Word & record CR
X	011111	11101 10010 /	**tlbre**	TLB Read Entry
X	011111	11101 11010 0	**extsb**	Extend Sign Byte
X	011111	11101 11010 1	**extsb.**	Extend Sign Byte & record CR
X	011111	11110 01011 0	**divwuo**	Divide Word Unsigned & record OV
X	011111	11110 01011 1	**divwuo.**	Divide Word Unsigned & record OV & CR
X	011111	11110 10010 /	**tlbwe**	TLB Write Entry
X	011111	11110 10110 /	**icbi**	Instruction Cache Block Invalidate
X	011111	11111 01011 0	**divwo**	Divide Word & record OV
X	011111	11111 01011 1	**divwo.**	Divide Word & record OV & CR
X	011111	11111 10110 /	**dcbz**	Data Cache Block set to Zero
D	100000	----- ----- -	**lwz**	Load Word & Zero
D	100001	----- ----- -	**lwzu**	Load Word & Zero with Update
D	100010	----- ----- -	**lbz**	Load Byte & Zero
D	100011	----- ----- -	**lbzu**	Load Byte & Zero with Update
D	100100	----- ----- -	**stw**	Store Word
D	100101	----- ----- -	**stwu**	Store Word with Update
D	100110	----- ----- -	**stb**	Store Byte
D	100111	----- ----- -	**stbu**	Store Byte with Update
D	101000	----- ----- -	**lhz**	Load Half Word & Zero
D	101001	----- ----- -	**lhzu**	Load Half Word & Zero with Update
D	101010	----- ----- -	**lha**	Load Half Word Algebraic
D	101011	----- ----- -	**lhau**	Load Half Word Algebraic with Update

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 3-10. Instructions Sorted by Opcode (continued)

Format	Opcode		Mnemonic	Instruction
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)		
D	101100	----- ----- -	**sth**	Store Half Word
D	101101	----- ----- -	**sthu**	Store Half Word with Update
D	101110	----- ----- -	**lmw**	Load Multiple Word
D	101111	----- ----- -	**stmw**	Store Multiple Word

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 4.12, "Cache Line Locking/Unlocking APU." Full decriptions of these instrucitons are provided in the EREF.

Chapter 4
L1 Cache

This chapter describes the organization of the on-chip L1 cache, cache control instructions, and various cache operations. It describes the interaction between the caches, the load/store unit (LSU), the instruction unit, and the memory subsystem. This chapter also describes the replacement algorithm used for the L1 cache.

Signals mentioned in this chapter are described in Chapter 8, “External Core Complex Interfaces.”

4.1 Overview

The L1 cache incorporates the following features:

- 32-Kbyte unified cache design
- Virtually indexed, physically tagged
- 32-byte (8-word) line size
- 64-bit data, 32-bit address
- Pseudo–round-robin replacement algorithm
- Eight-entry store buffer
- One-entry push (copy back) buffer
- One-entry line-fill buffer that provides critical double-word forwarding for both data accesses and instruction fetching
- Hit under fill/copy back
- Parity protection

The e200z6 processor supports a 32-Kbyte, 8-way set-associative, unified (instruction and data) cache with a 32-byte line size. The cache improves system performance by providing low-latency data to the e200z6 instruction and data pipelines; this decouples processor performance from system memory performance. The cache is virtually indexed and physically tagged. The e200z6 does not provide hardware support for cache coherency in a multiple-master environment. Software must be used to maintain coherency with other possible bus masters.

Both instruction and data accesses are performed using a single bus connected to the cache. Addresses from the processor to the cache are virtual addresses used to index the cache array. The memory management unit (MMU) provides the virtual-to-physical translation used for cache tag comparison. If the physical address matches a valid tag entry, the access hits in the cache. For a read operation, the cache supplies the data to the processor; for a write operation, the data from the processor updates the cache. If the access does not match a valid cache tag entry (misses in the cache) or a write access must be written through to memory, the cache generates a bus cycle on the system bus.

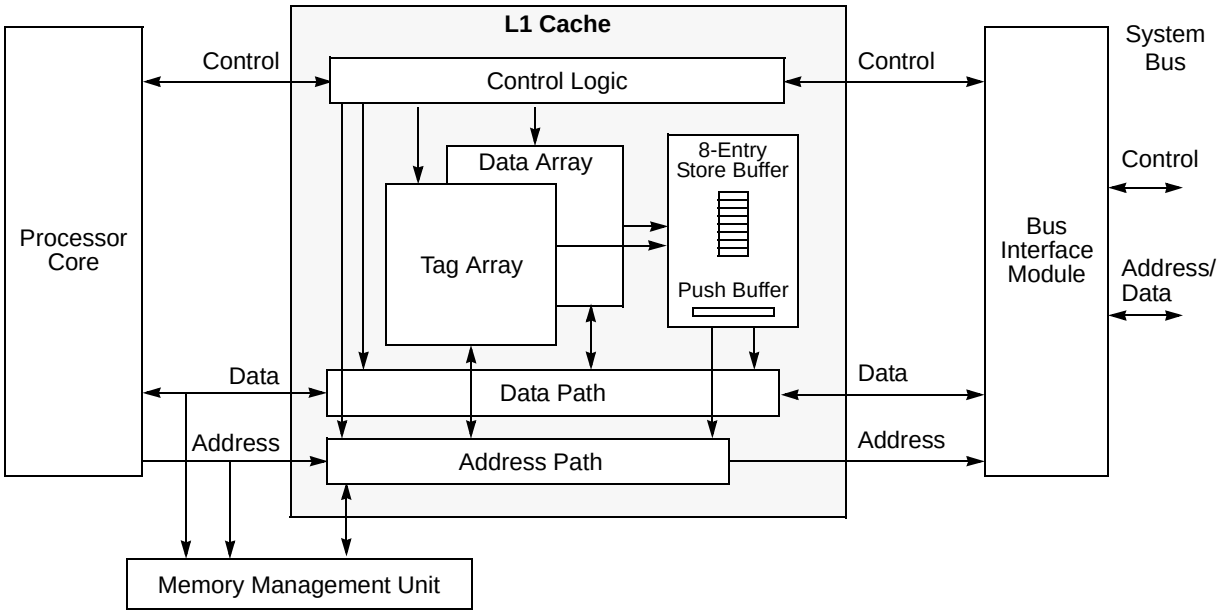


Figure 4-1. e200z6 Unified Cache

4.2 32-Kbyte Cache Organization

The e200z6 cache is organized as eight ways of 128 sets with each line containing 32 bytes (4 double words) of storage. Figure 4-2 shows the cache organization and the terminology used, along with the cache line format. Virtual address bits A[20–26] provide an index to select a set. Ways are selected according to the rules of set association.

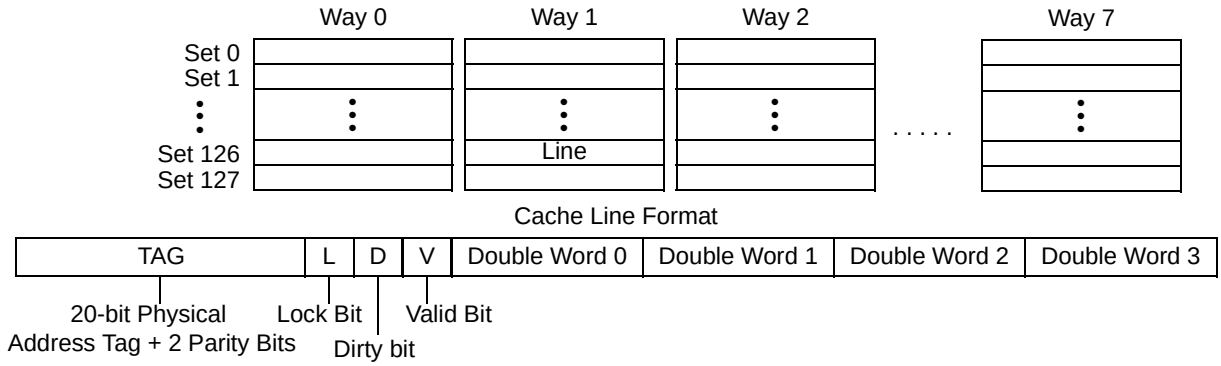


Figure 4-2. Cache Organization and Line Format

Each line consists of a physical address tag, status bits, and 4 double words of data with byte parity. Address bits A[27–29] select the word within the line.

4.2.1 32-Kbyte Cache Line Tag Format

Each cache line tag entry contains the physical address tag, a lock bit, dirty bit, and a valid bit. The format of a tag entry is shown in Figure 4-3.

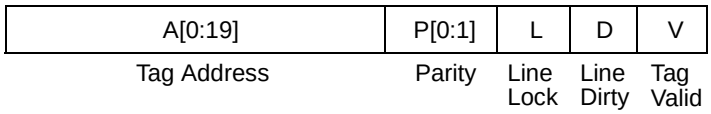


Figure 4-3. Cache Tag Format

Each field of the tag entry is detailed in Table 4-4.

Table 4-1. Tag Entry Field Descriptions

Field	Description
A[0:19]	Physical address tag. The physical address corresponding to the data contained in this line.
P[0:1]	Tag parity
L	Lock bit 0 The line has not been locked. 1 The line has been locked and is not available for replacement.
D	Dirty bit 0 The data contained in this entry has not been modified. 1 The data contained in this entry has been modified and is not consistent with physical memory.
V	Valid bit 0 This bit signifies that the cache line is invalid and a tag match should not occur. 1 This bit signifies that the cache line is valid.

4.3 Cache Lookup

Once enabled, the unified cache is searched for a tag match on all instruction fetches and data accesses from the CPU. If a match is found, the cached data is forwarded on a read access to the instruction fetch unit or the load/store unit (data access), or is updated on a write access, and may also be written-through to memory if required.

When a read miss occurs, if there is a TLB hit and the I bit of the hitting TLB entry is clear, the translated physical address is used to fetch a 4 double-word cache line beginning with the requested double word (critical double word first). The line is fetched and placed into the appropriate cache block and the critical double word is forwarded to the CPU. Subsequent double words may be streamed to the CPU if they have been requested and streaming is enabled via L1CSR0. Write misses do not allocate cache entries.

During a cache line fill, double words received from the bus are placed into a cache line fill buffer and may be forwarded (streamed) to the CPU if such a 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.

The cache always fills an entire line, thereby providing validity on a line-by-line basis. A cache line is always in one of the following states: invalid, valid, or modified (and valid). For invalid lines, the V bit is zero, causing the line to be ignored during lookups. Valid lines have their V bit set and D bit cleared, indicating the line contains valid data consistent with memory. Modified cache lines have the D and V bits set, indicating that the line has valid entries that have not been written to memory. In addition, a cache line may be locked (L bit set) indicating the line is not available for replacement.

Figure 4-4 shows the general flow of cache operation for the 32-Kbyte cache. To determine if the address is already allocated in the cache, the following steps are performed:

1. The cache set index, virtual address bits A[20:26], are used to select one cache set. A set is defined as the grouping of eight lines (one from each way), corresponding to the same index into the cache array.
2. Physical address bits A[0:19] are used as a tag reference or to update the cache line tag field.
3. The tags from the selected cache set are compared with the tag reference. If any tag matches the tag reference and the tag status is valid, a cache hit has occurred.
4. Virtual address bits A[27:28] are used to select one of the 4 double words in each line. A cache hit indicates that the selected double word in that line contains valid data (for a read access), or can be written with new data depending on the status of the W access control bit from the MMU (for a write access).

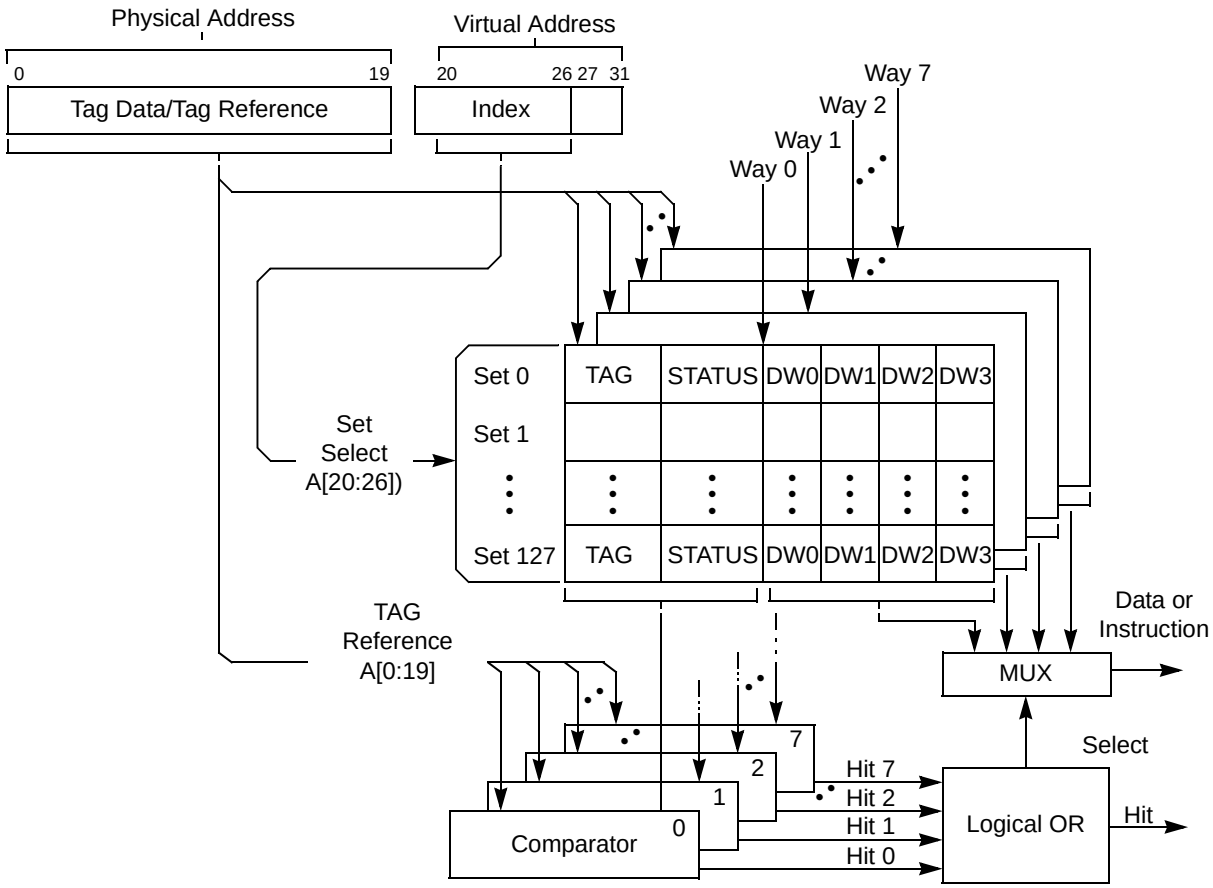


Figure 4-4. 32-Kbyte Cache Lookup Flow

4.4 Cache Control

The L1 cache control and status register (L1CSR0) provides control bits to enable/disable the cache and to invalidate it of all entries. In addition, availability of each way of the cache may be selectively controlled for use by instruction accesses and data accesses. This way control provides cache way locking capability, as well as controlling way availability on a cache line replacement. Ways 0–3 may be selectively disabled for instruction miss replacements and data miss replacements by using the WID and WDD control bits. In an eight-way cache, control for ways 4–7 is grouped, using a single disable bit (AWID and AWDD) for each type of replacement.

The following registers are defined for cache configuration and control:

- L1 cache control and status register—Used for general control and status of the L1 cache. See Section 2.13.1, “L1 Cache Control and Status Register 0 (L1CSR0).”
- L1 cache configuration register—Provides configuration information for the L1 cache supplied with this version of the e200z6 core. See Section 2.13.2, “L1 Cache Configuration Register 0 (L1CFG0).”

- L1 cache flush and invalidate register. Provides software-based flush and invalidation control for the L1 cache supplied with this version of the e200z6 core. See Section 2.13.3, “L1 Cache Flush and Invalidate Register (L1FINV0).”

4.5 Cache Coherency

Cache coherency is supported through software operations to invalidate lines, to flush modified lines to memory, or to invalidate modified lines. The cache may operate in either write-through or copy-back mode, and in conjunction with an MMU, it may designate certain accesses as write-through or copy-back. To ensure coherency, cache misses force the push and store buffers to empty before the access. No other hardware coherency support is provided.

4.6 Address Aliasing

The cache is physically addressed, thus eliminating any problems associated with potential cache synonyms due to effective address aliasing.

4.7 Cache Parity

Cache parity is supported for both the tag and data arrays. Two bits of parity are provided for the tag entry. Byte parity is supported for the data arrays. Parity checking is controlled by L1CSR0[CPE]. When parity checking is enabled, parity is checked on each cache access, whether for lookup or for modified line replacement. If a parity error is detected on any portion of the accessed data, a parity error is signaled, regardless of whether a cache hit or miss occurs. Parity errors are never signaled if L1CSR0[CPE] = 0. Signaling of a parity error causes a machine check exception and a syndrome bit to be set in the machine check syndrome register (MCSR). Section 5.6.2, “Machine Check Interrupt (IVOR1),” describes machine check exceptions, interrupts, and checkstop conditions. Section 2.7.2.3, “Machine Check Syndrome Register (MCSR),” describes how to use MCSR to determine the source of machine check exceptions.

4.8 Operation of the Cache

The following sections describe cache behavior.

4.8.1 Cache at Reset

L1CSR0[CE] is cleared on power-on reset or normal reset, disabling the cache. Section 4.8.2, “Cache Enable/Disable,” describes behavior when the cache is disabled.

Reset does not invalidate the cache lines; therefore, the cache should be invalidated explicitly after a hardware reset. After initial power-up, cache contents are undefined. The

L, D, and V bits may be set on some lines, necessitating invalidation of the cache by software before being enabled. See Section 4.8.6, “Cache Invalidation.”

4.8.2 Cache Enable/Disable

The cache is enabled or disabled by using the cache enable bit, L1CSR0[CE]. L1CSR0[CE] is cleared on power-on reset or normal reset, disabling the cache. If the cache is disabled, cache tag status bits are ignored and the cache is not accessed for normal loads, stores, or instruction fetches. All normal accesses are propagated to the system bus as single-beat (non-burst) transactions.

Note that the state of the cache-inhibited access attribute (I) remains independent of the state of L1CSR0[CE]. Disabling the cache does not affect the MMU translation logic. Translation attributes are still used when generating *p_hprot[4:2]* information on the system bus. (*p_hprot* signals are described in *Section 8.3, “Signal Descriptions*.”)

The store buffer is available even if the cache is disabled.

Altering the CE bit must be preceded by an **isync** and **msync** to prevent the cache from being disabled or enabled in the middle of a data or instruction access. In addition, the cache may need to be globally flushed before it is disabled to prevent coherency problems when it is re-enabled.

Disabling the cache affects all cache operations. Cache management instructions (except for **mtspr** L1FINV0 and **mtspr** L1CSR0) do not affect the cache when it is disabled.

4.8.3 Cache Fills

Cache line fills are requested when a cacheable load miss occurs. Store misses do not allocate cache lines. The burst fill is performed critical double word first on the bus. The critical double word is forwarded to the requesting unit before being written to the cache, thus minimizing stalls due to fill delays. Cache line fills load a 4-double-word line-fill buffer, and actual updates to the cache array are delayed until the next cache line fill is initiated. Read accesses may hit in the line buffer and data supplied from the buffer to the CPU. On writes, hits to the buffer cause it to be updated with the write data, although these writes stall until the buffer has been filled.

Data may be streamed to the CPU as it arrives from the bus if a corresponding request is pending. In addition, the cache supports hit under fill, allowing subsequent CPU accesses to be satisfied by cache hits while the remainder of the line fill completes. This non-blocking capability improves performance by hiding a portion of the line fill latency when data already in the cache or line-fill buffer is subsequently requested by the CPU.

Cache fill operations are performed as four-beat wrapping bursts (WRAP4 bursts) on the system bus. WRAP4 burst operations are described in Chapter 8, “External Core Complex

Interfaces." If an ERROR response is received on any element of the burst, the burst is terminated and the cache line are marked invalid.

4.8.4 Cache Line Replacement

On a cache read 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 entire 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 by the type of the miss access (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.

Modified lines selected for replacement must be copied back to main memory. This is performed by first storing the replaced line in a 32-byte push buffer (if it is enabled) while the missed data is fetched. After the new line is filled, the buffer contents are written to memory beginning with double word 0. If the push buffer is disabled, the copy back of a modified line precedes a line fill for the missed read address. These copy-back transfers begin with the quad word corresponding to the offset in the cache line of the requested data.

The replacement pointer is initialized to point to way 0 on a reset or on a cache-invalidate-all operation.

4.8.5 Cache-Inhibited Accesses

If a cache miss occurs while caching is inhibited, all accesses are performed as single-beat transactions on the system bus. Cache-inhibited status is ignored on all cache hits. For cache-inhibited accesses, the processor termination is withheld until the store buffer is flushed of all entries, the push buffer is emptied, and the store completes to memory (see Section 4.9, "Push and Store Buffers").

4.8.6 Cache Invalidation

The e200z6 supports full invalidation of the cache under software control. The cache may be invalidated through the cache invalidate control bit L1CSR0[CINV]. This function is available even when the cache is disabled.

Reset does not invalidate the cache automatically. Software must use the CINV control for invalidation after a reset. Proper use of this bit is to determine that it is clear and then set it with a pair of **mfspr mtspr** operations. A 0-to-1 transition on CINV initiates a flash invalidation that lasts for multiple (approximately 134) CPU cycles. Once set, CINV is cleared by hardware after the operation is complete. It remains set during the invalidation

interval and may be tested by software to determine when the operation has completed. An **mtspr** operation to L1CSR0 that attempts to change the state of CINV during invalidation does not affect the state of that bit.

During the process of performing the invalidation, the cache does not respond to accesses, and remains busy. Interrupts may still be recognized and processed, potentially aborting the invalidation operation. When this occurs, L1CSR0[ABT] is set to indicate unsuccessful completion of the operation. Software should read L1CSR0 to determine that the operation has completed (L1CSR0[CINV] cleared), and then check the status of L1CSR0[ABT] to determine completion status.

Individual cache blocks may be invalidated using **icbi**, **dcbi**, or **dcbf**. For these instructions to operate normally the cache must be enabled.

4.8.7 Cache Flush/Invalidate by Set and Way

The e200z6, shown in Figure 4-1, supports software-controlled cache flushing. The cache may be flushed and/or invalidated by index and way through an **mtspr l1finv0** instruction.

The L1 flush and invalidate control register 0 is used to select a set and way to be flushed/invalidated. No tag match is required. This function is available even if the cache is disabled. See Section 2.13.3, “L1 Cache Flush and Invalidate Register (L1FINV0).”

External termination errors that occur on the push of a modified cache line cause a machine check condition.

4.9 Push and Store Buffers

The push buffer reduces latency for requested new data on a cache miss by temporarily holding displaced modified data while the new data is fetched from memory. The push buffer contains 32 bytes of storage (one displaced cache line).

If a cache miss displaces a modified line and the push buffer is enabled, the line-fill request is immediately forwarded to the external bus. While waiting for the response, the current contents of the modified cache line are placed into the push buffer. When the line-fill transaction (burst read) completes, the cache controller can generate the appropriate burst write bus transaction to write the contents of the push buffer into memory.

In the disabled push buffer case, modified line replacement is performed by first generating a burst write transaction, copying out the entire modified line starting with the double word in the modified line corresponding to the missed address. After completion of the modified line write, a line fill is requested beginning with the critical double word (miss address).

To maximize performance, the store buffer contains a FIFO that can defer pending write misses or writes marked as write-through. The store buffer can buffer as many as 8 words (32 bytes) for this purpose. The store buffer may be disabled for debug purposes. Store

buffer operation is independent of L1CSR0[CE]. When the store buffer is enabled, cacheable store operations that miss the cache or are marked as write through are placed in the store buffer, and the CPU access is terminated. To properly drive the *p_hprot[4:1]* outputs on a buffered store access, each buffer entry contains 32 bits of physical address, 32 bits of data, size information, and 3 bits of access attribute information (W, G, and S/U). *Section 8.3, "Signal Descriptions*," describes the *p_hprot* signals.

Once the push or store buffer has valid data, the internal bus controller uses the next available external bus cycle to generate the appropriate write cycles. In the event that another cache fill is required (for example, cache read miss to process) during the continued instruction execution by the processor pipeline, the pipeline stalls until the push and store buffers are empty before generating the required external bus transaction.

The store buffer is always emptied before a cache line push to avoid memory consistency issues. After the push buffer has been loaded, a subsequent store may be buffered, but is not written to memory until the push has completed.

For cache-inhibited accesses, processor termination is withheld until all store buffer entries are flushed, the push buffer is emptied, and the store completes to memory.

A write to L1CSR0 may be used to force the push and store buffers to empty before proceeding with the actual L1CSR0 update. Additionally, **msync** and **mbar** also cause these buffers to be emptied before completion.

If an external transfer ERROR response occurs while emptying the store buffer, a machine check exception is signaled to the CPU, and a store for the next entry to be written (if any) is initiated. If a transfer error occurs for a push buffer transaction, the push of the remaining portion of the cache line is aborted, and a machine check exception is signaled to the CPU. This is also the case for a modified line copyback with the push buffer disabled, or a cache control operation that causes a line to be pushed. Following the transfer error, the line is marked invalid. If a transfer error can to be returned by the system on a push or a buffered store, and if this could cause a problem, the address must be marked cache inhibited.

For cache flush operations, if a transfer error occurs on a cache line flush, the push of the remaining portion of the cache line is aborted, the line remains marked modified and valid.

4.10 Cache Management Instructions

Table 4-2 describes the implementation of cache management instructions in the e200z6. Full descriptions of these instructions are provided in the EREF.

Table 4-2. Cache Management Instructions

Instruction	e200z6 Implementation Description
Instruction Cache Block Invalidate (**icbi**)	The e200z6 maps the **icbi** instruction to **dcbf**.
Instruction Cache Block Touch (**icbt**)	Treated as a no-op.
Data Cache Block Allocate (**dcba**)	Treated as a no-op.
Data Cache Block Flush (**dcbf**)	If **dcbf** addresses a byte in a modified cache line, the line is copied back to memory and subsequently invalidated, regardless of whether it was copied back or locked. If a cache line fill is in progress and the line-fill data corresponds to the EA associated with a **dcbf**, the cache is not updated with line-fill data. This instruction is treated as a load for the purposes of access protection. If the cache is disabled, this instruction is treated as a no-op.
Data Cache Block Invalidate (**dcbi**)	If **dcbi** addresses a byte in a line present in the cache, the line is invalidated, regardless of lock status. If **dcbi** addresses a modified line, no copy back occurs. If a cache line fill is in progress and the line-fill data corresponds to the EA associated with a **dcbi**, the cache is not updated with line-fill data. This instruction is privileged. It is treated as a store for the purposes of access protection. If the cache is disabled, this instruction is treated as a no-op in supervisor mode.
Data Cache Block Store (**dcbst**)	If **dcbst** addresses a byte in a modified line in the cache, the line is copied back to memory and then marked clean. The lock status is unchanged This instruction is treated as a load for the purposes of access protection. If the cache is disabled, this instruction is treated as a no-op.
Data Cache Block Touch (**dcbt**)	Treated as a no-op.
Data Cache Block Touch for Store (**dcbtst**)	Treated as a no-op.
Data Cache Block Set to Zero (**dcbz**)	If **dcbz** addresses a byte in a line in the cache, the line is zeroed, marked as modified, and remains valid. Lock status remains unchanged. If the line is not present and the address is cacheable, the line is established in the cache (without fetching from memory), is zeroed, and marked as modified and valid. This instruction is treated as a store for the purposes of access protection. **dcbz** causes an alignment exception if the EA is marked by the MMU as cache-inhibited and a cache miss occurs, or if the EA is marked by the MMU as write-through required, or if the cache is disabled or is operating in write-through mode, or if an overlocking condition prevents the allocation of a line into the cache.

4.11 Touch Instructions

Due to the limitations of using the **icbt**, **dcbt**, and **dcbtst** instructions, a program that uses these instructions improperly may actually see a degradation in performance from their use. To avoid this, the e200z6 treats these instructions as no-ops.

4.12 Cache Line Locking/Unlocking APU

The e200z6 supports the Motorola Book E cache line locking APU, which defines user-mode instructions to perform cache locking/unlocking. Three of the instructions are for data cache locking control (**dcblc**, **dcbtls**, and **dcbtstls**) and the remaining instructions are for instruction cache locking control (**icblc** and **icbtls**).

For the e200z6 unified cache, the instruction and data versions of these instructions operate similarly, although separate exception syndrome register (ESR) bits are supported for some error conditions. No state is maintained to indicate whether a cache line was locked with an **icbtls**, **dcbtls**, or **dcbtstls** instruction; thus the **icblc** and **dcblc** unlock a matching cache line regardless of how the lock bit was originally set.

The **dcbtls**, **dcbtstls**, and **dcblc** lock instructions are treated as reads for checking access permissions when translated by the TLB, and exceptions are taken for data TLB errors or data storage interrupts. The **icbtls** and **icblc** instructions require either execute (X) or read (R) permission when translated by the TLB. Exceptions are taken using data TLB errors or data storage interrupts, not ITLB or ISI.

The user-mode cache lock enable bit, MSR[UCLE], may be used to restrict user-mode cache line locking. If MSR[UCLE] is zero, any cache lock instruction executed in user-mode takes a cache-locking DSI exception and sets either ESR[DLK] or ESR[ILK]. If MSR[UCLE] is set, cache-locking instructions can be executed in user-mode and they do not take a DSI for cache-locking. However, they may still cause a DSI for access violations or precise external termination errors.

In the following cases, attempting to set a lock fails even if no DSI or DTLB exceptions occur:

- The target address is marked cache inhibited and a cache miss occurs.
- The cache is disabled.
- The cache target indicated by the CT field (bits 6–10) of the instruction is not zero.

In these cases, the lock set instruction is treated as a no-op, and the cache unable to lock bit, L1CSR0[CUL], is set.

Assuming no exception conditions occur (DSI or DTLB error), for **dcbtls**, **dcbtstls**, and **icbtls** an attempt is made to lock the corresponding cache line. If a miss occurs and all 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 causes an overlocking situation. In this case, the new line is not cached and the cache overlock bit, L1CSR0[CLO], is set to indicate that an overlocking situation occurred. This does not cause an exception condition.

The CUL conditions have priority over the CLO condition.

If multiple no-op or exception conditions arise on a cache lock instruction, the results are determined by the order of precedence described in Table 4-4.

It is possible to lock all ways of a given cache set. If an attempt is made to perform a non-locking line fill for a new address in the same cache set, the new line is not put into the cache. It is satisfied on the bus using a single-beat transfer instead of normal burst transfers. If **dcbz** is executed and all ways available for allocation are locked, an alignment exception is generated and no line is put into the cache.

Cache line locking interacts with the ability to control replacement of lines in certain cache ways via the L1CSR0 WID, AWID, WDD, and AWDD control bits. If any cache line locking instruction (**icbtls**, **dcbtls**, **dcbtstls**) executes and finds a matching line in the cache, the line's lock bit is set regardless of the WID, AWID, WDD, and AWDD settings. In this case, no replacement has been made. However, for cache misses which occur while executing a cache line lock set instruction, the only candidate lines available for locking are those which correspond to ways of the cache which have not been disabled for the particular type of line-locking instruction (controlled by WDD and AWDD for **dcbtls** and **dcbtstls**, controlled by WID and AWID for **icbtls**). Thus, an overlocking condition may result even though fewer than eight lines with the same index are locked.

The cache-locking DSI handler must decide whether or not to lock a given cache line based upon available cache resources. If the locking instruction is a set lock instruction, and if the handler decides to lock the line, it should do the following:

- Add the line address to its list of locked lines.
- Execute the appropriate set lock instruction to lock the cache line.
- Modify save/restore register 0 to point to the instruction immediately after the locking instruction that caused the DSI.
- Execute an **rfi**.

If the locking instruction is a clear lock instruction, and if the handler decides to unlock the line, it should do the following:

- Remove the line address from its list of locked lines.
- Execute the appropriate clear lock instruction to unlock the cache line.
- Modify save/restore register 0 to point to the instruction immediately after the locking instruction that caused the DSI.
- Execute an **rfi**.

Table 4-3 describes the e200z6 implementation of the cache locking APU instructions, which are fully described in the EREF. Note that the e200z6 only supports a cache target (CT) value of 0.

Table 4-3. Cache Locking APU Instructions

Instruction	e200z6 Implementation Description
Data Cache Block Touch and Lock Set (**dcbtls**)	If CT ≠ 0 or if the cache is disabled, **dcbtls** is no-oped and L1CSR0[CUL] is set indicating an unable-to-lock condition. No other exceptions are reported.
Data Cache Block Touch for Store and Lock Set (**dcbtstls**)	The e200z6 handles **dcbtstls** and **dcbtls** identically because no hardware coherency mechanisms are implemented for the cache.
Data Cache Block Lock Clear (**dcblc**)	If CT ≠ 0 or if the cache is disabled, the **dcblc** is no-oped. No other exceptions are reported.
Instruction Cache Block Touch and Lock Set (**icbtls**)	If CT ≠ 0 or if the cache is disabled, the **icbtls** is no-oped and L1CSR0[CUL] is set indicating an unable-to-lock condition. No other exceptions are reported.
Instruction Cache Block Lock Clear (**icblc**)	If CT ≠ 0 or if the cache is disabled, the **icblc** is no-oped. No other exceptions are reported.

4.12.1 Effects of Other Cache Instructions on Locked Lines

The **icbt**, **dcba**, **dcbz**, **dcbst**, **dcbt,** and **dcbtst** instructions do no affect the state of a cache line's lock bit.

The **dcbf, dcbi** and **icbi** instructions flush/invalidate and unlock a L1 cache line.

4.12.2 Flash Clearing of Lock Bits

The e200z6 supports flash clearing of cache lock bits under software control by using the cache flash clear locks control bit, L1CSR0[CFCL].

Lock bits are not cleared automatically at power-up (*m_por*) or normal reset (*p_reset_b*). Software must use CLFC to clear the lock bits after reset. This bit should be read (using **mfspr**) to determine that it is clear and then set (using **mtspr**). A 0-to-1 transition on CLFC initiates a flash clearing of the lock bits, which lasts approximately 134 CPU cycles. CLFC remains set (and may be read) until the operation completes, at which point it is cleared by hardware. During invalidation, an **mtspr** to L1CSR0 cannot affect the state of CLFC.

During the flash clearing process, the cache does not respond to accesses and remains busy. If an interrupt is recognized and processed, causing the flash clearing operation to fail, L1CSR0[ABT] is set. Software should read L1CSR0 to determine if the operation completed (CLFC = 0) and then read L1CSR0[ABT] to determine completion status.

4.13 Cache Instructions and Exceptions

All cache-management instructions (except **icbt**, **dcba**, **dcbt** and **dcbtst**, which are treated as no-ops) can generate TLB miss exceptions if the effective address cannot be translated, or may generate DSI exceptions due to permission violations. In addition, **dcbz** may generate an alignment interrupt as described in Table 4-2.

The cache-locking instructions (**dcblc, dcbtls**, **dcbtstls**, **icblc**, and **icbtls**) generate DSI exceptions if MSR[UCLE] is clear and the locking instruction is executed in user mode (MSR[PR] = 1). Data cache locking instructions that cause a DSI exception for this reason set ESR[DLK] (documented as DLK0 in Book E); instruction cache locking instructions that cause a DSI exception for this reason set ESR[ILK] (documented as DLK1 in Book E).

4.13.1 Exception Conditions for Cache Instructions

If multiple no-op or exception conditions arise on a cache instruction, the results are determined by the order of precedence described in Table 4-4.

Table 4-4. Special Case Handling

Operation	CT!=0	Cache Disabled	TLB Miss	User & UCLE=0	Protection Violation	CI and Cache Miss	All Available Ways Locked	WT or Cache in Write-Through Mode	Precise External Termination Error
dcbtls **dcbtstls** **dcblc**	DCUL DCUL NOP	DCUL DCUL NOP	DTLB DTLB DTLB	DLK DLK DLK	DSI DSI DSI	DCUL DCUL —	DCLO DCLO —	— — —	XTE XTE —
icbtls **icblc**	ICUL NOP	ICUL NOP	DTLB DTLB	ILK ILK	DSI DSI	ICUL —	ICLO —	_ —	XTE —
dcbz	_	ALI	DTLB	—	DSI	ALI	ALI	ALI	—
dcbf, **dcbst**, **icbi**	—	NOP	DTLB	—	DSI	—	—	—	XTE
dcbi [1]	—	NOP	DTLB	—	DSI	—	—	—	—
lwarx **stwcx.**	_ —	_ —	DTLB DTLB	_ —	DSI DSI	_ —	_ —	_ —	XTE XTE
load store	_ —	_ —	DTLB DTLB	_ —	DSI DSI	_ —	_ —	_ —	XTE XTE

Notes
- Priority decreases from left to right
- Cache operations that do not set or clear locks ignore the value of the CT field
- “dash” indicates executes normally
- “NOP” indicates treated as a no-op
- DSI = data storage interrupt; ALI = alignment interrupt; DTLB = data TLB interrupt
- DCUL, ICUL = no-op, and set L1CSR0[CUL]
- DCLO, ICLO = no-op, and set L1CSR0[CLO]
- DLK, ILK = data storage interrupt (DSI) and set ESR[DLK] or ESR[ILK]
- MC = Machine check and update DEAR
- XTE = DSI + set ESR[XTE]

[1] Privileged

4.13.2 Transfer Type Encodings for Cache Management Instructions

Transfer type encodings are used to indicate to the cache whether a normal access, atomic access, cache management control access, or MMU management control access is being requested. These attribute signals are driven with addresses when an access is requested. Table 4-5 shows the definitions of the *p_ttype[0:3]* encodings.

Table 4-5. Transfer Type Encoding

p_ttype[0:3]	Transfer Type	Instruction
0000	Normal	Normal loads / stores
0001	Atomic	**lwarx**, **stwcx.**
0010	Flush block	**dcbst**
0011	Flush and Invalidate block	**dcbf**, **icbi**
0100	Allocate and zero block	**dcbz**
0101	Invalidate block	**dcbi**
0110–0111	Reserved	**—**
1000	TLB invalidate	**tlbivax**
1001	TLB search	**tlbsx**
1010	TLB read entry	**tlbre**
1011	TLB write entry	**tlbwe**
1100	Lock clear for data	**dcblc**
1101	Lock set for data	**dcbtls**, **dcbtstls**
1110	Lock clear for instruction	**icblc**
1111	Lock set for instruction	**icbtls**

4.14 Sequential Consistency

The PowerPC architecture requires that all memory operations executed by a single processor be sequentially self-consistent. This means that all memory accesses appear to be executed in the order that is specified by the program with respect to exceptions and data dependencies. The e200z6 CPU achieves this effect by operating a single pipeline to the cache/MMU. All memory accesses are presented to the MMU in the same order that they appear in the program and therefore exceptions are determined in order.

4.15 Self-Modifying Code Requirements

The following sequence of instructions synchronizes the instruction stream.

```
dcbf
icbi
msync
isync
```

This sequence is redundant for the e200z6, but ensures that the operation is correct for other Book E–compliant processors which implement separate instruction and data caches, as well as for multiple-processor cache-coherent systems.

4.16 Page Table Control Bits

The PowerPC architecture allows certain memory characteristics to be set on a page and on a block basis. These characteristics include write through (using the W bit), cacheability (using the I bit), coherency (using the M bit), guarded memory (using the G bit), and endianness (using the E bit). Incorrect use of these bits may create situations where coherency paradoxes are observed by the processor. In particular, this can occur when the state of these bits is changed without appropriate precautions being taken (that is, flushing 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 are allowed by the Book E architecture, however others may present cache coherence paradoxes and are considered programming errors.

4.16.1 Write-Through Stores

A write-through store (WIMGE = 0b1*xxxx*) may normally hit to a valid cache line. In this case, the cache line remains in its current state, the store data is written into the cache, and the store goes out on the bus as a single-beat write.

4.16.2 Cache-Inhibited Accesses

When the cache-inhibited attribute is indicated by translation (WIMGE = b’x1xxx’) and a cache miss occurs, all accesses are performed as single-beat transactions on the system bus with a size indicator corresponding to the size of the load, store or prefetch operation. Cache inhibited status is ignored on all cache hits. For cache-inhibited misses, the store and push buffers are emptied prior to performing the miss access.

4.16.3 Memory Coherence Required

For the e200z6, the memory coherence required storage attribute (WIMGE = b’xx1xx’) is ignored.

4.16.4 Guarded Storage

For the e200z6, the guarded storage attribute (WIMGE = b’xxx1x’) is ignored except for generation of the *p_hprot[4:2]* attributes on an external access, since no out-of-order execution is supported by the hardware. (*p_hprot* signals are described in *Section 8.3, “Signal Descriptions*.”)

4.16.5 Misaligned Accesses and the Endian (E) Bit

Misaligned load or store accesses that cross page boundaries can cause data corruption if the two pages do not have the same endianness. That is, if one page is big endian and the other is little endian, the processor would not get all the bytes, or would get some bytes out of order, resulting in garbled data. To protect against data corruption, the e200z6 core takes a DSI exception and sets the byte ordering bit, ESR[BO], when this occurs.

4.17 Reservation Instructions and Cache Interactions

The e200z6 core treats **lwarx** and **stwcx.** accesses as though they were cache inhibited, regardless of page attributes. Additionally, a cache line corresponding to the address of a **lwarx** or **stwcx.** access are flushed to memory if modified, and then invalidated (even if marked as locked), prior to the **lwarx** or **stwcx.** access being issued to the bus. This allows the building of external reservation logic that properly signals a reservation failure. The bus access is treated as a single-beat transfer.

4.18 Effect of Hardware Debug on Cache Operation

Hardware debug facilities use normal CPU instructions to access register and memory contents during a debug session. This may have the unavoidable side-effect of causing the store and push buffers to be flushed. During hardware debug, the MMU page attributes are controllable by the debug firmware via settings of the OnCE control register, described in Section 10.5.5.3, “e200z6 OnCE Control Register (OCR).”

4.19 Cache Memory Access during Debug

The cache memory provides resources needed to do background accesses through the JTAG/OnCE port to read and write the cache SRAM arrays. Accesses are supported via a pair of OnCE-mapped registers. These resources are intended for use only by special debug tools and debug software.

Access to cache memory SRAM arrays using this port can occur only if the CPU is in debug mode before a read or write access is initiated.

This debug port 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, push, store, and late-write buffers 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 should be written to the cache arrays using the merging control GO command described in the next section.

The CPU must be in the debug state before a merge control command is initiated.

4.19.1 Merging Line-Fill and Late-Write Buffers into the Cache Array

To ensure that the cache array is updated with the latest information in various internal buffers, the content of these buffers must be merged back into the actual cache arrays, To merge late-write and line-fill buffer data contents into the cache arrays, the user must access and write CDACNTL[GO] to 10. This ensures that the cache arrays contain the data from the last cache write hit (late write) and the last line loaded from a read miss (line fill).

4.19.2 Cache Memory Access through JTAG/OnCE Port

Cache debug access control and data information is serially accessed through the OnCE controller using the cache debug access control and data registers, CDACNTL and CDADATA (see Table 4-6 and Table 4-7). Accesses are performed one word at a time.

For a cache write access, the user must first write CDADATA with the desired tag or data values. The second step is to write CDACNTL with desired tag or data location, parity and modified information (for data writes only), and set the CDACNTL R/W and GO bits.

Information about accessing OnCE registers is provided in Section 10.5.5.2, “e200z6 OnCE Command Register (OCMD).”

For a cache read access, the user must first access and write to CDACNTL with desired tag or data location, and set the R/W and GO bits in CDACNTL. The second step is to access and read CDADATA for the tag or data and read CDACNTL for parity (data reads only).

Completion of any operation can be determined by reading CDACNTL. Operations are indicated as complete when CDACNTL[GO] = 00. Debug firmware should poll CDACNTL to determine when an access has been completed prior to assuming validity of any other information in CDACNTL or CDADATA.

4.19.2.1 Cache Debug Access Control Register (CDACNTL)

CDACNTL, shown in Figure 4-5, 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. It includes data SRAM parity bit values, which must be supplied by the user for write accesses, and which is supplied by the cache for read accesses of the data SRAM arrays.

	0	1–3	4	5	6–12	13–15	16–19	20–28	29	30	31
Field	T/D	CWAY	—		CSET	WORD	PARITY	—	R/W	GO	
Reset	All zeros										
R/W	R/W										
Number	OCMD[RS] = 111 1010										

Figure 4-5. CDACNTL Register

Table 4-6 describes CDACNTL fields.

Table 4-6. Cache Debug Access Control Register Definition

Bits	Name	Description
0	T/D	Tag/data: 0 Data array selected 1 Tag array selected
1–3	CWAY	Cache way. Specifies the cache way to be selected
4–5	—	Reserved, should be cleared.
6–12	CSET	Cache set. Specifies the cache set to be selected
13–15	WORD	Word (data array access only). Specifies one of 8 words of selected set
16–19	PARITY	Data parity bits (data array access only). Byte parity bits. One bit per data byte (Bytes 0–3 of selected word) (cache parity checkers assume odd parity.) 16: Parity for byte 0, 17: Parity for byte 1, 18: Parity for byte 2, 19: Parity for byte 3
20–28	—	Reserved, should be cleared.
29	R/W	Read/write: 0 Selects write operation. Write the data in CDADATA to the location specified by CDACNTL. 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	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. 10 Merge valid late write and line-fill buffer data into the cache arrays 11 Reserved

4.19.2.2 Cache Debug Access Data Register (CDADATA)

The cache debug access data register (CDADATA), shown in Figure 4-6, contains SRAM data for a debug access. It is also used for tag and data SRAM read and write operations.

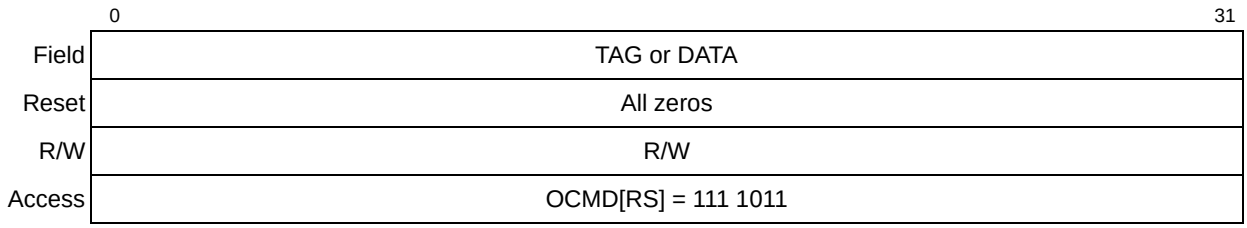


Figure 4-6. Cache Debug Access Data Register (CDADATA)

Table 4-7 describes CDADATA fields.

Table 4-7. CDADATA Field Descriptions

Bits	Name	Description
0–31	TAG	TAG array access data 0–19 Tag compare bits 20 Valid bit 21–22 Lock bits. These two bits must always have the same value, 1-Locked, 0-Unlocked. 23 Parity high bit, parity for bits 0–10 (cache parity checkers assume odd parity) 24 Parity low bit, parity for bits 11–22 (cache parity checkers assume odd parity) 25 Dirty bit 26–31 Reserved, write as zero
	DATA	DATA array access data (bytes 0–3 of the selected word) 0–7 Byte 0 8–15 Byte 1 16–23 Byte 2 24–31 Byte 3

Chapter 5
Interrupts and Exceptions

This chapter provides a general description of the PowerPC Book E interrupt and exception model and gives details of the additions and changes to that model that are implemented in the e200z6 core. This chapter identifies features defined by Book E, the Motorola Book E implementation standards (EIS), and the e200z6 implementation.

5.1 Overview

Book E defines the mechanisms by which the e200z6 core implements interrupts and exceptions. Note the following definitions:

Interrupt	Action in which the processor saves its old context and begins execution at a predetermined interrupt handler address
Exceptions	Events that, if enabled, cause the processor to take an interrupt

The PowerPC exception mechanism allows the processor to change to supervisor state for the following reasons:

- As a result of unusual conditions (exceptions) arising in the execution of instructions
- As a response to the assertion of external signals, bus errors, or various internal conditions

When an interrupt occurs, information about the processor state held in the MSR and the address at which execution should resume after the interrupt is handled are saved to a pair of save/restore registers (SRR0/SRR1 for non-critical interrupts, CSRR0/CSRR1 for critical interrupts, or DSRR0/DSRR1 for debug interrupts when the debug APU is enabled) and the processor begins executing at an address (interrupt vector) determined by the interrupt vector prefix register (IVPR) and an interrupt-specific interrupt vector offset register (IVOR*n*). 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 save/restore registers soon after the interrupt is taken. Hardware supports nesting of critical interrupts within non-critical interrupts, and debug interrupts within both critical and non-critical interrupts. The interrupt handler must save necessary state information if interrupts of a given class are re-enabled within the handler.

The following terms are used to describe the stages of exception processing:

Recognition	Exception recognition occurs when the condition that can cause an exception is identified by the processor. Recognition is also referred to as an ‘exception event.’
Taken	An interrupt is said to be taken when control of instruction execution is passed to the interrupt handler; that is, the context is saved, the instruction at the appropriate vector offset is fetched, and the interrupt handler routine begins.
Handling	Interrupt handling is performed by the software linked to the appropriate vector offset. Interrupt handling is begun in supervisor mode.

Returning from an interrupt is performed by executing **rfi**, **rfci**, or **rfdi**, which restores state information from their respective save/restore registers and returns instruction fetching to the interrupted flow.

The e200z6 supports the existence of multiple hardware register contexts. A new operating context can be selected during the process of handing off control of instruction execution to an interrupt handler. Contexts are selected by control field in the IVOR used to determine the location of an interrupt handler. See Section 2.15, “Support for Fast Context Switching,” for more information on context support.

5.2 e200z6 Interrupts

The Book E architecture specifies that interrupts can be precise or imprecise, synchronous or asynchronous, and critical or non-critical, and are described as follows:

- Asynchronous exceptions are caused by events external to the processor’s instruction execution.
- Synchronous exceptions are directly caused by instructions or by an event somehow synchronous to the program flow, such as a context switch.
- A precise interrupt architecturally guarantees that no instruction beyond the instruction causing the exception has (visibly) executed. An imprecise interrupt does not have this guarantee.
- Book E defines critical and non-critical interrupt types, and the e200z6 defines an implementation-specific debug APU that includes the debug interrupt type. Each interrupt type provides separate resources (save/restore registers and return from

interrupt instructions) that allow interrupts of one type to not interfere with the state handling of an interrupt of another type.

Table 5-1 describes how these definitions apply to the interrupts implemented by the e200z6 core.

Table 5-1. Interrupt Classifications

Interrupt Types	Synchronous/Asynchronous	Precise/Imprecise	Critical/Non-Critical/Debug
System reset (not an interrupt on the e200z6; included here for reference)	Asynchronous, non-maskable	Imprecise	—
Machine check	—	—	Critical
Critical input Watchdog timer	Asynchronous, maskable	Imprecise	Critical
External input Fixed-interval timer Decrementer	Asynchronous, maskable	Imprecise	Non-critical
Instruction-based debug	Synchronous	Precise	Critical/debug
Debug (UDE) Debug imprecise	Asynchronous	Imprecise	Critical/Debug
Data storage/alignment/TLB Instruction storage/TLB	Synchronous	Precise	Non-critical

The classifications in Table 5-1 are discussed in greater detail in Section 5.6, “Interrupt Definitions.” Table 5-2 lists interrupts implemented in the e200z6 and the exception conditions that cause them. Note that although this table lists system reset, Book E does not define system reset as an interrupt and assigns no interrupt vector to it.

Table 5-2. Exceptions and Conditions

Interrupt Type	IVOR*n*	Cause	Section/Page
System reset (not an interrupt)	None, vector to [*p_rstbase[0:19]*] || 0xFFC	• Reset by assertion of *p_reset_b* • Watchdog timer reset control • Debug reset control	—
Critical input	IVOR 0 [1]	*p_critint_b* is asserted and MSR[CE]=1	5.6.1/5-9
Machine check	IVOR 1	• *p_mcp_b* is asserted and MSR[ME] =1 • ISI, ITLB error on first instruction fetch for an exception handler and current MSR[ME] = 1 • Parity error signaled on cache access and current MSR[ME]=1 • Write bus error on buffered store or cache line push	5.6.2/5-10
Data storage	IVOR 2	• Access control • Byte ordering due to misaligned access across page boundary to pages with mismatched E bits • Cache locking exception • Precise external termination error (*p_hresp*=ERROR and precise recognition)	5.6.3/5-12

Table 5-2. Exceptions and Conditions (continued)

Interrupt Type	IVOR*n*	Cause	Section/Page
Instruction storage	IVOR 3	• Access control • Precise external termination error (*p_hresp*=ERROR and precise recognition)	5.6.4/5-13
External input	IVOR 4[1]	*p_extint_b* is asserted and MSR[EE]=1	5.6.5/5-14
Alignment	IVOR 5	• **lmw**, **stmw** not word aligned • **lwarx** or **stwcx.** not word aligned • **dcbz** with disabled cache, or to W or I storage • Misaligned SPE load and store instructions	5.6.6/5-14
Program	IVOR 6	Illegal, privileged, trap, floating-point enabled, APU enabled, unimplemented operation	5.6.7/5-15
Floating-point unavailable	IVOR 7	MSR[FP] = 0 and attempt to execute a Book E floating-point operation	5.6.8/5-16
System call	IVOR 8	Execution of the System Call (**sc**) instruction	5.6.9/5-17
APU unavailable	IVOR 9	Unused by the e200z6	5.6.10/5-17
Decrementer	IVOR 10	As specified in Book E	5.6.11/5-17
Fixed-interval timer	IVOR 11	As specified in Book E	5.6.12/5-18
Watchdog timer	IVOR 12	As specified in Book E	5.6.13/5-19
Data TLB error	IVOR 13	Data translation lookup did not match a valid TLB entry.	5.6.14/5-20
Instruction TLB error	IVOR 14	Instruction translation lookup did not match a valid TLB entry.	5.6.15/5-20
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	5.6.16/5-21
Reserved	IVOR 16–31	—	—
SPE unavailable exception	IVOR 32	See Section 5.6.18, “SPE APU Unavailable Interrupt (IVOR32).”	5.6.18/5-25
SPE data exception	IVOR 33	See Section 5.6.19, “SPE Floating-Point Data Interrupt (IVOR33).”	5.6.19/5-25
SPE round exception	IVOR 34	See Section 5.6.20, “SPE Floating-Point Round Interrupt (IVOR34).”	5.6.20/5-26

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

5.3 Exception Syndrome Register (ESR)

The ESR provides a syndrome to differentiate exceptions that can generate the same interrupt type. The e200z6 adds implementation-specific bits to the ESR.

The ESR fields are described in Table 5-3.

Table 5-3. ESR Field Descriptions

Bit(s)	Name	Description	Associated Interrupt Type
32–35	—	Reserved, should be cleared.	—
36	PIL	Illegal instruction exception	Program
37	PPR	Privileged instruction exception	Program
38	PTR	Trap exception	Program
39	FP	Floating-point operation	Alignment, data storage, data TLB, program
40	ST	Store operation	Alignment, data storage, data TLB
41	—	Reserved, should be cleared.	—
42	DLK	Data cache locking	Data storage
43	ILK	Instruction cache locking	Data storage
44	AP	Auxiliary processor operation (unused in the E200z6)	Alignment, data storage, data TLB, program
45	PUO	Unimplemented operation exception	Program
46	BO	Byte ordering exception	Data storage
47	PIE	Program imprecise exception—unused in the e200z6 (reserved, should be cleared)	—
48–55	—	Reserved, should be cleared.	—
56	SPE	SPE APU operation	SPE unavailable, SPE floating-point data exception, SPE floating-point round exception, alignment, data storage, data TLB
57–62	—	Reserved on the e200z6, should be cleared.	—
63	XTE	External termination error (precise)	Data storage, instruction storage

5.4 Machine State Register (MSR)

The MSR, shown in Figure 5-4, defines the state of the processor.

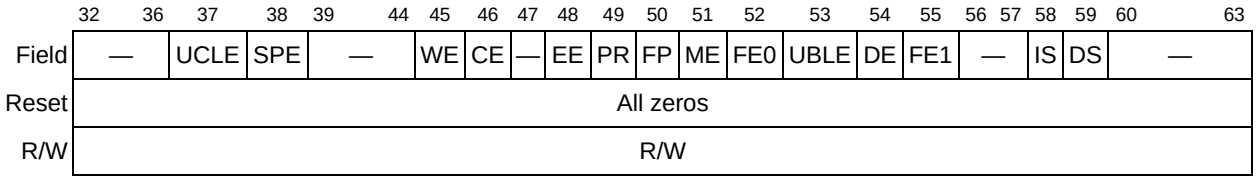


Figure 5-1. Machine State Register (MSR)

The MSR bits are described in Table 5-4.

Table 5-4. MSR Field Descriptions

Bits	Name	Description
32–36	—	Reserved, should be cleared.
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 is set in the ESR. 1 Execution of the cache lock instructions in user mode enabled
38	SPE	SPE available 0 Execution of SPE APU vector instructions is disabled; SPE Unavailable exception taken instead, and ESR[SPE] is set. 1 Execution of SPE APU vector instructions is enabled.
39–44	—	Reserved, should be cleared.
45	WE	Wait state (power management) enable. Defined as optional by Book E and implemented in the e200z6. 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 HID0[DOZE,NAP,SLEEP], described in Section 2.11.1, “Hardware Implementation-Dependent Register 0 (HID0).”
46	CE	Critical interrupt enable 0 Critical input and watchdog timer interrupts are disabled. 1 Critical input and watchdog timer interrupts are enabled.
47	—	Preserved
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.
49	PR	Problem state 0 The processor is in supervisor mode, can execute any instruction, and can access any resource (for example, GPRs, SPRs, MSR, etc.). 1 The processor is in user mode, cannot execute any privileged instruction, and cannot access any privileged resource.
50	FP	Floating-point available 0 Floating-point unit is unavailable. The processor cannot execute floating-point instructions, including floating-point loads, stores, and moves. (An FP unavailable interrupt is generated on attempted execution of floating-point instructions). 1 Floating-point unit is available. The processor can execute floating-point instructions. (Note that for the e200z6, the floating-point unit is not supported; an unimplemented operation exception is generated for attempted execution of floating-point instructions when FP is set).
51	ME	Machine check enable 0 Machine check interrupts are disabled. Checkstop mode is entered when *p_mcp_b* is recognized asserted or an ISI or ITLB exception occurs on a fetch of the first instruction of an exception handler. 1 Machine check interrupts are enabled.
52	FE0	Floating-point exception mode 0 (not used by the e200z6)
53	—	Reserved, should be cleared.
54	DE	Debug interrupt enable 0 Debug interrupt APU is disabled and the Book E–defined critical-type debug interrupt is invoked if a debug interrupt occurs. 1 Debug interrupt APU is enabled and the e200z6-defined debug APU interrupt is invoked if a debug interrupt occurs.

Table 5-4. MSR Field Descriptions (continued)

Bits	Name	Description
55	FE1	Floating-point exception mode 1 (not used by the e200z6)
56	—	Reserved, should be cleared.
57	—	Preserved, should be cleared.
58	IS	Instruction address space 0 The processor directs all instruction fetches to address space 0 (TS=0 in the relevant TLB entry). 1 The processor directs all instruction fetches to address space 1 (TS=1 in the relevant TLB entry).
59	DS	Data address space 0 The core directs all data storage accesses to address space 0 (TS=0 in the relevant TLB entry). 1 The core directs all data storage accesses to address space 1 (TS=1 in the relevant TLB entry).
60–61	—	Reserved, should be cleared.
62–63	—	Preserved, should be cleared.

5.4.1 Machine Check Syndrome Register (MCSR)

When the core complex takes a machine check interrupt, it updates the machine check syndrome register (MCSR), to differentiate among machine check conditions. The MCSR also indicates whether the source of a machine check condition is recoverable. When an MCSR bit is set, the core complex asserts *p_mcp_out* for system information.

Table 5-5 describes MCSR fields.

Table 5-5. MCSR Field Descriptions

Bits	Name	Description	Recoverable
32	MCP	Machine check input pin	Maybe
33	—	Reserved, should be cleared.	—
34	CP_PERR	Cache push parity error	Unlikely
35	CPERR	Cache parity error	Precise
36	EXCP_ERR	ISI, ITLB, or bus error on first instruction fetch for an exception handler	Precise
37–60	—	Reserved, should be cleared.	—
61	BUS_WRERR	Write bus error on buffered store or cache line push	Unlikely
62–63	—	Reserved, should be cleared.	—

5.4.1.1 Interrupt Vector Prefix Register (IVPR)

The IVPR, shown in Figure 5-2, is used during interrupt processing for determining the starting address for the software interrupt handler. The value contained in the vector offset field of the IVOR selected for a particular interrupt type is concatenated with the value in the IVPR to form an instruction address from which execution is to begin.

	32 — 47	48 — 63
Field	Vector Base	—
Reset	Undefined on *m_por* assertion, unchanged on *p_reset_b* assertion	
R/W	R/W	
SPR	SPR 63	

Figure 5-2. Interrupt Vector Prefix Register (IVPR)

IVPR fields are defined in Table 5-6.

Table 5-6. IVPR Field Descriptions

Bits	Name	Description
32–47	Vector Base	Used to define the base location of the vector table, aligned to a 64-Kbyte boundary. This field provides the high-order 16 bits of the location of all interrupt handlers. The contents of the IVOR*n* appropriate for the type of exception being processed are concatenated with the IVPR vector base to form the address of the handler in memory.
48–63	—	Reserved, should be cleared.

5.5 Interrupt Vector Offset Registers (IVOR*n*)

IVORs are used during interrupt processing for determining the starting address of a software interrupt handler. The value in the vector offset field of the IVOR assigned to the interrupt type is concatenated with the value in IVPR to form an instruction address at which execution is to begin. The e200z6 also defines the low-order bits of the IVORs (defined as zeros in Book E) as a context selector field to be used as the current context number once interrupt handling begins when multiple hardware contexts are supported (CTXCR[NUMCTX] ≠ 0). For forward compatibility, this field should be written to zero when only a single context is supported because it will not be implemented and is read as zero.

	32 — 47	48 — 59	60	61 — 63
Field	—	Vector Offset	—	CS
Reset	Unaffected			
R/W	R/W			
SPR	See Table 5-7.			

Figure 5-3. Interrupt Vector Offset Registers (IVOR)

IVOR SPR assignments are shown in Table 5-7.

Table 5-7. IVOR Assignments

IVOR Number	SPR	Interrupt Type
IVOR0	400	Critical input
IVOR1	401	Machine check
IVOR2	402	Data storage
IVOR3	403	Instruction storage
IVOR4	404	External input
IVOR5	405	Alignment
IVOR6	406	Program
IVOR7	407	Floating-point unavailable
IVOR8	408	System call
IVOR9	409	Auxiliary processor unavailable. Not used by the e200z6.
IVOR10	410	Decrementer
IVOR11	411	Fixed-interval timer interrupt
IVOR12	412	Watchdog timer interrupt
IVOR13	413	Data TLB error
IVOR14	414	Instruction TLB error
IVOR15	415	Debug
IVOR16–IVOR31	—	Reserved for future architectural use
e200z6-Specific IVORs (Defined by the EID)		
IVOR32	528	SPE APU unavailable
IVOR33	529	SPE floating-point data exception
IVOR34	530	SPE floating-point round exception

5.6 Interrupt Definitions

The following sections describes interrupts as they are implemented on the e200z6.

5.6.1 Critical Input Interrupt (IVOR0)

A critical input exception is signaled to the processor by the assertion of the critical interrupt pin (*p_critint_b*). When the e200z6 detects the exception, if critical interrupts are enabled (MSR[CE] = 1), the e200z6 takes the critical input interrupt. The *p_critint_b* input is a level-sensitive signal expected to remain asserted until the e200z6 acknowledges the interrupt. If *p_critint_b* is negated early, recognition of the interrupt request is not guaranteed. After the e200z6 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 5-8 lists register settings when a critical input interrupt is taken.

Table 5-8. Critical Input Interrupt Register Settings

Register	Setting Description
CSRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present.
CSRR1	Set to the contents of the MSR at the time of the interrupt.
MSR	UCLE 0 SPE 0 WE 0 CE 0 EE 0 PR 0 FP 0 ME — FE0 0 DE —/0 [1] FE1 0 IS 0 DS 0
ESR	Unchanged
MCSR	Unchanged
DEAR	Unchanged
Vector	IVPR[32–47] || IVOR0[48–59] || 0b0000 (autovectored) IVPR[32–47] || *p_voffset[0:11]* || 0b0000 (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 APU is enabled, MSR[DE] is not automatically cleared by a critical input interrupt, but can be configured to be cleared through HID0 (HID0[CICLRDE]). Refer to Section 2.11.1, “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 e200z6 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]* inputs 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).

5.6.2 Machine Check Interrupt (IVOR1)

The e200z6 implements the machine check exception as defined in Book E except for automatic clearing of MSR[DE] (see later paragraph). The e200z6 initiates a machine check interrupt if MSR[ME]=1 and any of the machine check sources listed in Table 5-2 is detected. As defined in Book E, the interrupt is not taken if MSR[ME] is cleared, in which case the processor generates an internal checkstop condition and enters checkstop state. When a processor is in checkstop state, instruction processing is suspended and generally cannot continue without restarting the processor. Note that other conditions may lead to the checkstop condition; the disabled machine check exception is only one of these.

The e200z6 implements the machine check syndrome register (MCSR) to record the sources of machine checks.

MSR[DE] is not automatically cleared by a machine check exception, but can be configured to be cleared or left unchanged through HID0[MCCLRDE]. See Section 2.11.1, “Hardware Implementation-Dependent Register 0 (HID0).”

5.6.2.1 Machine Check Interrupt Enabled (MSR[ME]=1)

Machine check interrupts are enabled when MSR[ME]=1. When a machine check interrupt is taken, registers are updated as shown in Table 5-9.

Table 5-9. Machine Check Interrupt Register Settings

Register	Setting Description
CSRR0	On a best-effort basis, the e200z6 sets this to the address of some instruction that was executing or about to be executing when the machine check condition occurred.
CSRR1	Set to the contents of the MSR at the time of the interrupt
MSR	UCLE 0 EE 0 DE 0 — Cleared when the debug APU is disabled. Clearing of DE is optionally supported by control in HID0 when the debug APU is enabled. SPE 0 PR 0 WE 0 FP 0 FE1 0 CE 0 ME 0 IS 0 DS 0
ESR	Unchanged
MCSR	Updated to reflect the sources of a machine check
DEAR	Unchanged unless machine check is due to a data access causing a cache parity error to be signaled; updated with data access effective address in that case
Vector	IVPR[32–47] || IVOR1[48–59] || 0b0000

The machine check input, *p_mcp_b*, can be masked by HID0[EMCP].

Most 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.

The MCSR is provided to identify the sources of a machine check and may be used to identify recoverable events.

The interrupt handler should set MSR[ME] as soon as possible to avoid entering checkstop state if another machine check condition occurs.

5.6.2.2 Checkstop State

Exception-related checkstop conditions are as follows:

- MSR[ME] = 0 and a machine check occurs

- First instruction in an interrupt handler cannot be executed due to a translation miss (ITLB), page marked no execute (ISI), or a bus error termination, and MSR[ME]=0
- Bus error termination for a buffered store or a cache line push and MSR[ME]=0
- Cache parity error condition is signaled and MSR[ME] = 0

Non-exception–related checkstop conditions are as follows:

- TCR[WRC]—Watchdog reset control bits set to checkstop on second watchdog timer overflow event

When a processor is in checkstop state, instruction processing is suspended and generally cannot resume without the processor being reset. To indicate that a checkstop condition exists, the *p_chkstop* output is asserted whenever the CPU is in checkstop state.

When a debug request is presented to the e200z6 core while in the checkstop state, *p_wakeup* is asserted, and when *m_clk* is provided to the CPU, it temporarily exits checkstop state and enters debug mode. The *p_chkstop* output is negated for the duration of the time the CPU remains in a debug session (*p_debug_b* asserted). When the debug session is exited, the CPU re-enters checkstop state. Note that the external system logic may be in an undefined state following a checkstop condition, such as having an outstanding bus transaction, or other inconsistency; thus, no guarantee can be made in general about activities performed in debug mode while a checkstop is still outstanding. Debug logic does have the capability of generating assertion of *p_resetout_b* through DBCR0.

5.6.3 Data Storage Interrupt (IVOR2)

A data storage interrupt (DSI) may occur if no higher priority exception exists and one of the following exception conditions exists:

- Read or write access control exception condition
- Byte ordering exception condition
- Cache locking exception condition
- External termination error (precise)

Access control is defined as in Book E. 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 **dcbtls**, **dcbtstls**, **dcblc**, **icbtls**, or **icblc** in user mode with MSR[UCLE] = 0. External termination errors occur when a load, cache-inhibited, or guarded store is terminated by assertion of a *p_hpresp*=ERROR termination response.

Table 5-10 lists register settings when a DSI is taken.

Table 5-10. 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 SPE 0 WE 0 CE — EE 0 PR 0 FP 0 ME — FE0 0 DE — FE1 0 IS 0 DS 0
ESR	Access: [ST], [SPE]. All other bits cleared. Byte ordering: [ST], [SPE], BO. All other bits cleared. Cache locking: (DLK, ILK), [ST]. All other bits cleared. External termination error (precise): [ST], [SPE], XTE. All other bits cleared.
MCSR	Unchanged
DEAR	For access and byte-ordering exceptions, set to the effective address of a byte within the page whose access caused the violation. Undefined on cache-locking exceptions. (The e200z6 does not update DEAR on a cache locking exception.)
Vector	IVPR[32–47] || IVOR2[48–59] || 0b0000

5.6.4 Instruction Storage Interrupt (IVOR3)

An instruction storage interrupt (ISI) occurs when no higher priority exception exists and an execute access control exception occurs. This interrupt is implemented as defined by Book E, except that the byte ordering condition does not occur in the e200z6 and the addition of precise external termination errors that occur when an instruction fetch is terminated by assertion of a *p_hpresp*=ERROR termination response.

Table 5-11 lists register settings when an ISI is taken.

Table 5-11. Instruction Storage Interrupt Register Settings

Register	Setting Description
SRR0	Set to the effective address of the excepting instruction.
SRR1	Set to the contents of the MSR at the time of the interrupt
MSR	UCLE 0 SPE 0 WE 0 CE — EE 0 PR 0 FP 0 ME — FE0 0 DE — FE1 0 IS 0 DS 0
ESR	[XTE]. All other bits cleared.
MCSR	Unchanged
DEAR	Unchanged
Vector	IVPR[32–47] || IVOR3[48–59] || 0b0000

5.6.5 External Input Interrupt (IVOR4)

An external input exception is signaled to the processor by the assertion of the external interrupt input (*p_extint_b*), a level-sensitive signal expected to remain asserted until the e200z6 acknowledges the external interrupt. If *p_extint_b* is negated early, recognition of the interrupt request is not guaranteed. When the e200z6 detects the exception, if the exception is enabled by MSR[EE], the e200z6 takes an 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 5-12 lists register settings when an external input interrupt is taken.

Table 5-12. 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 SPE 0 WE 0 CE — EE 0 PR 0 FP 0 ME — FE0 0 DE — FE1 0 IS 0 DS 0
ESR	Unchanged
MCSR	Unchanged
DEAR	Unchanged
Vector	IVPR[32–47] || IVOR4[48–59] || 0b0000 IVPR[32–47] || *p_voffset[0:11]* || 0b0000 (non-autovectored)

IVOR4 is the vector offset register used by autovectored external input interrupts to determine the interrupt handler location. The e200z6 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 an external input interrupt request is not autovectored (*p_avec_b* negated when *p_extint_b* asserted).

5.6.6 Alignment Interrupt (IVOR5)

The e200z6 implements the alignment interrupt as defined by Book E. An alignment exception is generated when any of the following occurs:

- The operand of **lmw** or **stmw** is not word-aligned.
- The operand of **lwarx** or **stwcx.** is not word-aligned.
- Execution of **dcbz** is attempted with a disabled cache.

- Execution of **dcbz** is attempted with an enabled cache and W or I =1.
- Execution is attempted of an SPE APU load or store instruction that is not properly aligned.

Table 5-13 lists register settings when an alignment interrupt is taken.

Table 5-13. Alignment Interrupt Register Settings

Register	Setting Description
SRR0	Set to the effective address of the excepting load/store instruction.
SRR1	Set to the contents of the MSR at the time of the interrupt
MSR	UCLE 0 PR 0 DE — SPE 0 FP 0 FE1 0 WE 0 ME — IS 0 CE — FE0 0 DS 0 EE 0
ESR	[ST], [SPE]. 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[32–47] || IVOR5[48–59] || 0b0000

5.6.7 Program Interrupt (IVOR6)

The e200z6 implements the program interrupt as defined by Book E. A program interrupt occurs when no higher priority exception exists and one or more of the following exception conditions defined in Book E occur:

- Illegal instruction exception
- Privileged instruction exception
- Trap exception
- Unimplemented operation exception

The e200z6 invokes an illegal instruction program exception on attempted execution of the following instructions:

- Instruction from the illegal instruction class
- **mtspr** and **mfspr** with an undefined SPR specified

The e200z6 invokes a privileged instruction program exception on attempted execution of the following instructions when MSR[PR]=1 (user mode):

- A privileged instruction
- **mtspr** and **mfspr** instructions that specify a SPRN value with SPRN[5] = 1 (even if the SPR is undefined).

The e200z6 invokes an trap exception on execution of **tw** and **twi** if the trap conditions are met and the exception is not also enabled as a debug interrupt.

The e200z6 invokes an unimplemented operation program exception on attempted execution of the instructions **lswi**, **lswx**, **stswi**, **stswx**, **mfapidi**, **mfdcr**, **mfdcrx**, **mtdcr**, **mtdcrx**, or any Book E floating-point instruction when MSR[FP]=1. All other defined or allocated instructions that are not implemented by the e200z6 cause a illegal instruction program exception.

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

Table 5-14. 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 PR 0 DE — SPE 0 FP 0 FE1 0 WE 0 ME — IS 0 CE — FE0 0 DS 0 EE 0
ESR	Illegal: PIL. All other bits cleared. Privileged: PPR. All other bits cleared. Trap: PTR. All other bits cleared. Unimplemented: PUO, [FP]. All other bits cleared.
MCSR	Unchanged
DEAR	Unchanged
Vector	IVPR[32–47] || IVOR6[48–59] || 0b0000

5.6.8 Floating-Point Unavailable Interrupt (IVOR7)

The floating-point unavailable exception is implemented as defined in Book E. 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 5-15 lists register settings when a floating-point unavailable interrupt is taken.

Table 5-15. 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

Table 5-15. Floating-Point Unavailable Interrupt Register Settings (continued)

Register	Setting Description		
MSR	UCLE 0 SPE 0 WE 0 CE — PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0
ESR	Unchanged		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	IVPR[32–47] || IVOR7[48–59] || 0b0000		

5.6.9 System Call Interrupt (IVOR8)

A system call interrupt occurs when a system call (**sc**) is executed and no higher priority exception exists. Table 5-16 lists register settings when a system call interrupt is taken.

Table 5-16. System Call Interrupt Register Settings

Register	Setting Description		
SRR0	Set to the effective address of the instruction *following* the **sc** instruction.		
SRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 SPE 0 WE 0 CE — EE 0	PR 0 FP 0 ME — FE0 0	DE — FE1 0 IS 0 DS 0
ESR	Unchanged		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	IVPR[32–47] || IVOR8[48–59] || 0b0000		

5.6.10 Auxiliary Processor Unavailable Interrupt (IVOR9)

An APU exception is defined by Book E 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 e200z6 does not use this interrupt.

5.6.11 Decrementer Interrupt (IVOR10)

The e200z6 implements the decrementer exception as described in Book E. 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 5-17 lists register settings when a decrementer interrupt is taken.

Table 5-17. Decrementer Interrupt Register Settings

Register	Setting Description
SRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present.
SRR1	Set to the contents of the MSR at the time of the interrupt
MSR	UCLE 0 SPE 0 WE 0 CE — EE 0 PR 0 FP 0 ME — FE0 0 DE — FE1 0 IS 0 DS 0
ESR	Unchanged
MCSR	Unchanged
DEAR	Unchanged
Vector	IVPR[32–47] || IVOR10[48–59] || 0b0000

5.6.12 Fixed-Interval Timer Interrupt (IVOR11)

The e200z6 implements the fixed-interval timer exception as defined in Book E. 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 fixed-interval timer exception exists (TSR[FIS]=1), and the interrupt is enabled (both TCR[FIE] and MSR[EE]=1).

The timer status register (TSR) holds the fixed-interval timer interrupt bit set by the timer facility when an exception is detected. Software must clear this bit in the interrupt handler to avoid repeated fixed-interval timer interrupts.

Table 5-18 lists register settings when a fixed-interval timer interrupt is taken.

Table 5-18. 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.

Table 5-18. Fixed-Interval Timer Interrupt Register Settings (continued)

Register	Setting Description
MSR	UCLE 0 PR 0 DE — SPE 0 FP 0 FE1 0 WE 0 ME — IS 0 CE — FE0 0 DS 0 EE 0
ESR	Unchanged
MCSR	Unchanged
DEAR	Unchanged
Vector	IVPR[32–47] || IVOR11[48–59] || 0b0000

5.6.13 Watchdog Timer Interrupt (IVOR12)

The e200z6 implements the watchdog timer interrupt as defined in Book E. The exception is triggered by the first enabled watchdog timeout.

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 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 5-19 lists register settings when a watchdog timer interrupt is taken.

Table 5-19. 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 PR 0 DE 0/— Cleared when the debug APU is disabled. Clearing DE is optionally supported by control in HID0 when the debug APU is enabled. SPE 0 FP 0 WE 0 ME — FE1 0 CE 0 FE0 0 IS 0 EE 0 DS 0
ESR	Unchanged
MCSR	Unchanged
DEAR	Unchanged
Vector	IVPR[32–47] || IVOR12[48–59] || 0b0000

MSR[DE] is not automatically cleared by a watchdog timer interrupt, but can be configured to be cleared through HID0[CICLRDE]. Refer to Section 2.11.1, “Hardware Implementation-Dependent Register 0 (HID0).”

5.6.14 Data TLB Error Interrupt (IVOR13)

A data TLB error interrupt occurs when no higher priority exception exists and a data TLB error exception exists due to a data translation lookup miss in the TLB. Table 5-20 lists register settings when a DTLB interrupt is taken.

Table 5-20. Data TLB Error Interrupt Register Settings

Register	Setting Description		
SRR0	Set to the effective address of the excepting load/store instruction.		
SRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 SPE 0 WE 0 CE — EE 0	PR 0 FP 0 ME — FE0 0	DE — FE1 0 IS 0 DS 0
ESR	[ST], [SPE]. 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[32–47] || IVOR13[48–59] || 0b0000		

5.6.15 Instruction TLB Error Interrupt (IVOR14)

An instruction TLB error interrupt occurs when no higher priority exception exists and an instruction TLB error exception exists due to an instruction translation lookup miss in the TLB. Table 5-21 lists register settings when an ITLB interrupt is taken.

Table 5-21. 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 — FE0 0	DE — FE1 0 IS 0 DS 0
ESR	Unchanged		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	IVPR[32–47] || IVOR14[48–59] || 0b0000		

5.6.16 Debug Interrupt (IVOR15)

The e200z6 implements the debug interrupt as defined in Book E with the following changes:

- When the debug APU is enabled (MSR[DE] = 1), debug is no longer a critical interrupt, but uses DSRR0 and DSRR1 for saving machine state on context switch.
- The return from debug interrupt instruction (**rfdi**) supports the debug APU save/restore registers (DSRR0 and DSRR1).
- The critical interrupt taken debug event allows critical interrupts to generate a debug event.
- The critical return debug event allows debug events to be generated for **rfci** instructions.

Multiple sources can signal a debug exception. A debug interrupt occurs when no higher priority exception exists, a debug exception is indicated in the debug status register (DBSR), and debug interrupts are enabled (DBCR0[IDM]=1 (internal debug mode) and MSR[DE]=1). Enabling debug events and other debug modes is discussed in Chapter 10, “Debug Support.”

With the debug APU enabled (see Section 2.11.1, “Hardware Implementation-Dependent Register 0 (HID0)”), the debug interrupt uses its own set of save/restore registers (DSRR0, DSRR1) to allow debugging of both critical and non-critical interrupt handlers. This capability also allows 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 through HID0[DCLREE,DCLRCE]. See Section 2.11.1, “Hardware Implementation-Dependent Register 0 (HID0).” When the debug APU is disabled, debug interrupts use CSRR0 and CSRR1 to save machine state.

NOTE

For additional details regarding the following descriptions of debug exception types, refer to Section 10.4, “Software Debug Events and Exceptions.”

Table 5-22. Debug Exceptions

Exception	Cause
Instruction address compare (IAC)	Instruction address compare events are enabled and an instruction address match occurs as defined by the debug control registers. This could either be a direct instruction address match or a selected set of instruction addresses. IAC has the highest priority of all instruction-based interrupts, even if the instruction itself encountered an ITLB error or instruction storage exception.
Branch taken (BRT)	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)	Data address compare events are enabled and a data access address match occurs as defined by the debug control registers. This could either be a direct data address match or a selected set of data addresses. The debug interrupt is taken when no higher priority exception is pending. The e200z6 does not implement the data value compare debug mode, specified in Book E. The e200z6 implementation provides IAC linked with DAC exceptions. This results in a DAC exception only if one or more IAC conditions are also met. See Chapter 10, “Debug Support,” for more details.
Trap (TRAP) debug	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 and is 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 is taken instead.
Return (RET)	Return exceptions are enabled and **rfi** is executed. Return debug exceptions are not generated for **rfci** or **rfdi**. If MSR[DE]=1 when **rfi** executes, a debug interrupt occurs if no higher priority exception exists that is enabled to cause an interrupt. CSRR0 (debug APU disabled) or DSRR0 (debug APU enabled) is set the address of the **rfi**. If MSR[DE] = 0 when **rfi** executes, a debug interrupt does not occur immediately; the event is recorded by setting DBSR[RET] and DBSR[IDE].
Critical return (CRET)	Critical return debug events are enabled and **rfci** is executed. Critical return debug exceptions are only generated for **rfci**. If MSR[DE]=1 when **rfci** executes, a debug interrupt occurs if no higher priority exception exists that is enabled to cause an interrupt. CSRR0 (debug APU disabled) or DSRR0 (debug APU enabled) is set to the address of the **rfci**. If MSR[DE] = 0 when **rfci** executes, a debug interrupt does not occur immediately, but the event is recorded by setting DBSR[CRET] and DBSR[IDE]. Note that critical return debug events should not normally be enabled unless the debug APU is enabled to avoid corrupting CSRR0 and CSRR1.
Instruction complete (ICMP)	An instruction completed while this event is enabled. A **mtmsr** or **mtdbcr0** that causes both MSR[DE] and DBCR0[IDM] to end up set, enabling precise debug mode, may cause an imprecise (delayed) debug exception to be generated due to an earlier recorded event in the DBSR.
Interrupt taken (IRPT)	A non-critical interrupt context switch is detected. This exception is imprecise and unordered with respect to the program flow. Note that an IRPT debug interrupt occurs only when detecting a non-critical interrupt on the e200z6. The value saved in CSRR0/DSRR0 is the address of the non-critical interrupt handler.
Critical interrupt taken (CIRPT)	A critical interrupt context switch is detected. This exception is imprecise and unordered with respect to program flow. Note that a CIRPT debug interrupt occurs only when detecting a critical interrupt on the e200z6. The address of the critical interrupt handler is saved in CSRR0/DSRR0. To avoid corrupting CSRR0 and CSRR1, critical interrupt taken debug events should not normally be enabled unless the debug APU is enabled.
Unconditional debug event (UDE)	The unconditional debug event signal (*p_ude*) transitions to asserted state.
Debug counter	A debug counter exception is enabled and a debug counter decrements to zero.
External debug	An external debug exception is enabled and an external debug event (*p_devt1*, *p_devt2*) transitions to the asserted state.

The DBSR provides a syndrome to differentiate among debug exceptions that can generate the same interrupt. See Chapter 10, “Debug Support.” Table 5-23 lists register settings when a debug interrupt is taken.

Table 5-23. Debug Interrupt Register Settings

Register	Setting Description
CSRR0 (MSR[DE]=0) DSRR0 (MSR[DE]=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 UDE, IRPT, CIRPT, DCNT, or DEVT type exceptions, set to the effective address of the instruction that 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 PR 0 DE 0 SPE 0 FP 0 FE1 0 WE 0 ME — IS 0 CE —/0 [1] FE0 0 DS 0 EE —/0[1]
DBSR [2]	Unconditional debug event: UDE Instruction complete debug event: ICMP Branch taken debug event: BRT Interrupt taken debug event: IRPT Critical interrupt taken debug event: CIRPT Trap instruction debug event: TRAP Instruction address compare: {IAC1, IAC2, IAC3, IAC4} Data address compare: {DAC1R, DAC1W, DAC2R, DAC2W} Return debug event: RET Critical return debug event: CRET Debug counter event: {DCNT1, DCNT2} External debug event: {DEVT1, DEVT2} and optionally, an imprecise debug event flag {IDE}
ESR	Unchanged
MCSR	Unchanged
DEAR	Unchanged
Vector	IVPR[32–47] || IVOR15[48–59] || 0b0000

1 Conditional based on HID0 control bits.

2 Note that multiple DBSR bits may be set.

5.6.17 System Reset Interrupt

The e200z6 implements the system reset interrupt as defined in Book E. The system reset exception is a non-maskable, asynchronous exception signaled to the processor through the assertion of system-defined signals.

A system reset may be initiated as follows:

- By asserting the *p_reset_b* input. *p_reset_b* must remain asserted for a period (specified in the hardware specifications) that allows internal logic to be reset. Assertion for less than the required interval causes unpredictable results.
- By asserting *m_por* during power-on reset. *m_por* must be asserted during power up and must remain asserted for a period (specified in the hardware specifications) that allows internal logic to be reset. Assertion for less than the required interval causes unpredictable results.
- By watchdog timer reset control
- By debug reset control

When a reset request occurs, the processor branches to the system reset exception vector (value on *p_rstbase[0:19]* concatenated with 0xFFC) without attempting to reach a recoverable state. If reset occurs during normal operation, all operations stop and machine state is lost. The internal state of the e200z6 after a reset is defined in Section 2.16.4, “Reset Settings.”

For reset initiated by watchdog timer or debug reset control, the e200z6 implements TSR[WRS] or DBSR[MRR] to help software determine the cause. Watchdog timer and debug reset control provide the capability to assert *p_resetout_b*. External logic may factor this signal into *p_reset_b* to cause an e200z6 reset.

Table 5-24 shows the TSR bits associated with reset status.

Table 5-24. TSR Watchdog Timer Reset Status

Bits	Name	Description
34–35	WRS	00 No action performed by watchdog timer 01 Watchdog timer second timeout caused checkstop. 10 Watchdog timer second timeout caused *p_resetout_b* to be asserted. 11 Reserved

Table 5-25 shows the DBSR bits associated with reset status.

Table 5-25. DBSR Most Recent Reset

Bits	Name	Function
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. 1*x* Reserved

Table 5-26 lists register settings when a system reset interrupt is taken.

Table 5-26. System Reset Interrupt Register Settings

Register	Setting Description
CSRR0	Undefined
CSRR1	Undefined

Table 5-26. System Reset Interrupt Register Settings (continued)

Register	Setting Description		
MSR	UCLE 0 SPE 0 WE 0 CE 0	EE 0 PR 0 FP 0 ME 0	DE 0 FE1 0 IS 0 DS 0
ESR	Cleared		
DEAR	Undefined		
Vector	[*p_rstbase[0:19]*] || 0xFFC		

5.6.18 SPE APU Unavailable Interrupt (IVOR32)

The SPE APU unavailable exception is taken if MSR[SPE] is cleared and execution of an SPE APU instruction other than an embedded scalar floating-point or **brinc** instruction is attempted. When the SPE APU unavailable exception occurs, the processor suppresses execution of the instruction causing the exception. Table 5-27 lists register settings when an SPE unavailable interrupt is taken.

Table 5-27. SPE Unavailable Interrupt Register Settings

Register	Setting Description		
SRR0	Set to the effective address of the excepting SPE instruction		
SRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 SPE 0 WE 0 CE — EE 0	PR 0 FP 0 ME — FE0 0	DE — FE1 0 IS 0 DS 0
ESR	SPE. All other bits cleared.		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	IVPR[32–47] || IVOR32[48–59] || 0b0000		

5.6.19 SPE Floating-Point Data Interrupt (IVOR33)

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

Table 5-28 lists register settings when an SPE floating-point data interrupt is taken.

Table 5-28. SPE Floating-Point Data Interrupt Register Settings

Register	Setting Description		
SRR0	Set to the effective address of the excepting SPE instruction.		
SRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 SPE 0 WE 0 CE — EE 0	PR 0 FP 0 ME — FE0 0	DE — FE1 0 IS 0 DS 0
ESR	SPE. All other bits cleared.		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	IVPR[32–47] || IVOR33[48–59] || 0b0000		

5.6.20 SPE Floating-Point Round Interrupt (IVOR34)

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

Table 5-29 lists register settings when an SPE floating-point round interrupt is taken.

Table 5-29. SPE Floating-Point Round Interrupt Register Settings

Register	Setting Description		
SRR0	Set to the effective address of the instruction following the excepting SPE instruction.		
SRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 SPE 0 WE 0 CE — EE 0	PR 0 FP 0 ME — FE0 0	DE — FE1 0 IS 0 DS 0
ESR	SPE. All other bits cleared.		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	IVPR[32–47] || IVOR34[48–59] || 0b0000		

5.7 Exception Recognition and Priorities

The following list of exception categories describes how the e200z6 handles exceptions up to the point of signaling the appropriate interrupt to occur. Also, instruction completion is defined as updating all architectural registers associated with that instruction as necessary, and then removing the instruction from the pipeline.

- Interrupts caused by asynchronous events (exceptions). These exceptions are further distinguished by whether they are maskable and recoverable.
  - — Asynchronous, 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, maskable, non-recoverable:

    Machine check interrupt

    Has priority over any other pending exception except system reset conditions; is dependent on the source of the exception. Typically non-recoverable.
  - — 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 remains pending until taken or cancelled by software.
- Synchronous, non-instruction-based interrupts. The only exception in 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 program flow.
  - — Synchronous, maskable, recoverable:

    Interrupt taken debug event

    The machine is 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, SPE floating-point data, SPE floating-point round, debug (trap, branch taken, return) 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).

5.7.1 Exception Priorities

Exceptions are prioritized as described in Table 5-30. Some exceptions may be masked or imprecise, which affects 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 most machine checks are non-recoverable.

Table 5-30. e200z6 **Exception Priorities**

Priority	Exception	Cause
Asynchronous Exceptions		
0	System reset	Assertion of *p_reset_b*, watchdog timer reset control, or debug reset control
1	Machine check	Assertion of *p_mcp_b*, cache parity error, exception on fetch of first instruction of an interrupt handler, bus error on buffered store or cache line push
2	—	—
3 [1]	1. Debug: UDE 2. Debug: DEVT1 3. Debug: DEVT2 4. Debug: DCNT1 5. Debug: DCNT2 6. Debug: 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
4[1]	Critical Input	Assertion of *p_critint_b*
5[1]	Watchdog timer	Watchdog timer first enabled time-out
6[1]	External input	Assertion of *p_extint_b*

Table 5-30. e200z6 **Exception Priorities (continued)**

Priority	Exception	Cause
7[1]	Fixed-interval timer	Posting of a fixed-interval timer exception in TSR due to programmer-specified bit transition in the time base register
8[1]	Decrementer	Posting of a decrementer exception in TSR due to programmer-specified decrementer condition
Instruction Fetch Exceptions		
9	Debug: IAC (unlinked)	Instruction address compare match for enabled IAC debug event and DBCR0[IDM] asserted
10	ITLB error	Instruction translation lookup miss in the TLB
11	Instruction storage	• Access control (no execute permission) • External termination error (precise)
Instruction Dispatch/Execution Interrupts		
12	Program: Illegal	Attempted execution of an illegal instruction
13	Program: privileged	Attempted execution of a privileged instruction in user mode
14	Floating-point unavailable	Any floating-point unavailable exception condition
	SPE Unavailable	Any SPE unavailable exception condition
15	Program: unimplemented	Attempted execution of an unimplemented instruction
16	1. Debug: BRT 2. Debug: Trap 3. Debug: RET 4. Debug: 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 require corresponding debug event enabled, MSR[DE]=1, and DBCR0[IDM]=1.
17	Program: trap	Condition specified in **tw** or **twi** instruction met and not trap debug.
	System call	Execution of the system call (**sc**) instruction.
	SPE floating-point data	NaN, infinity, or denormalized data detected as input or output, or underflow, overflow, divide by zero, or invalid operation in the SPE APU.
	SPE round	Inexact result
18	Alignment	**lmw**, **stmw, lwarx,** or **stwcx.** Not word aligned. **dcbz** with cache disabled
19	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 require corresponding debug event enabled, MSR[DE]=1, and DBCR0[IDM]=1. In this case, the debug exception is considered imprecise and DBSR[IDE] is set. Saved PC points to the load or store instruction causing the DAC event.
20	Data TLB error	Data translation lookup miss in the TLB.

Table 5-30. e200z6 **Exception Priorities (continued)**

Priority	Exception	Cause
21	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 **dcbtls**, **dcbtstls**, **dcblc**, **icbtls**, or **icblc** in user mode with MSR[UCLE] = 0. 4. External termination error (precise)
22	Alignment	**dcbz** to W=1 or I=1 storage with cache enabled
23	1. Debug: IRPT 2. Debug: 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.
Post-Instruction Execution Exceptions		
24	1. Debug: DAC/IAC linked[2] 2. Debug: 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 points to the instruction following the load or store instruction causing the DAC event.
25	Debug: ICMP	Completion of an instruction. **Note**: Exceptions requires corresponding debug event enabled, MSR[DE]=1, and DBCR0[IDM]=1.

1 These exceptions are sampled at instruction boundaries, and may actually occur after exceptions that 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 e200z6 implements the data address compare debug exceptions as post-instruction exceptions, which differs from the Book E definition. When a TEA (either a DTLB error or DSI) occurs in conjunction with an enabled DAC or linked DAC/IAC on a load or store class instruction, the debug interrupt takes priority, and the saved PC value points to the load or store class instruction, rather than to the next instruction.

5.8 Interrupt Processing

When an interrupt is taken, the processor uses SRR0/SRR1 for non-critical interrupts, CSRR0/CSRR1 for critical 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 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** executes. 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 APU is enabled and to restore those values when an **rfdi** executes.

The ESR 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 MSR is updated to preclude unrecoverable interrupts from occurring during the initial portion of the interrupt handler. Specific settings are described in Table 5-31.

For alignment, data storage, or data TLB miss interrupts, or for a machine check due to cache parity error on data access interrupts, the data exception address register (DEAR) is loaded with the address which caused the interrupt to occur.

For machine check interrupts, the MCSR is loaded with information specific to the exception type.

Instruction fetch and execution resumes, using the new MSR value, at a location specific to the exception type. The location is determined by the IVPR and an IVOR specific for each type of interrupt (see Table 5-2). A new operating context is selected using the low-order three bits of the specific IVOR selected by the type of interrupt.

Table 5-31 shows the MSR settings for different interrupt categories. Note that reserved and preserved MSR bits are unimplemented and are read as 0.

Table 5-31. MSR Setting Due to Interrupt

Bits	MSR Definition	Reset Setting	Non-Critical Interrupt	Critical Interrupt	Debug Interrupt
37	UCLE	0	0	0	0
38	SPE	0	0	0	0
45	WE	0	0	0	0
46	CE	0	—	0	—/0 [1]
48	EE	0	0	0	—/0[1]
49	PR	0	0	0	0
50	FP	0	0	0	0
51	ME	0	—	—	—
52	FE0	0	0	0	0
54	DE	0	—	—/0[1]	0
55	FE1	0	0	0	0
58	IS	0	0	0	0
59	DS	0	0	0	0

[1] Conditionally cleared based on control bits in HID0

5.8.1 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.
- A machine check exception can occur only if the machine check enable bit (MSR[ME]) is set. If MSR[ME] is cleared, the processor goes directly into checkstop state when a machine check exception condition occurs. Individual machine check exceptions can be enabled and disabled through HID0 bits.
- 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 10, “Debug Support,” for more details on individual control of debug exceptions.
- The floating-point unavailable exception can be prevented by setting MSR[FP] (although the e200z6 generates an unimplemented instruction exception instead).

5.8.2 Returning from an Interrupt Handler

The Return from Interrupt (**rfi**), Return from Critical Interrupt (**rfci**) and Return from Debug Interrupt (**rfdi**) instructions perform context synchronization by allowing previously-issued instructions to complete before returning to the interrupted process. In general, execution of **rfi**, **rfci**, or **rfdi** 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** copies SRR1 bits back into the MSR.
- The **rfci** copies CSRR1 bits back into the MSR.
- The **rfdi** copies DSRR1 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**, CSRR0 for **rfci** and DSRR0 for **rfdi**.

Note that the **rfi** may be subject to a return type debug exception and that **rfci** may be subject to a critical return type debug exception. For a complete description of context synchronization, refer to Book E.

5.9 Process Switching

The following instructions are useful for restoring proper context during process switching:

- **msync** 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.
- **isync** 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.
- **stwcx.** 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
Memory Management Unit

This chapter describes the implementation details of the e200z6 core complex MMU relative to the Book E architecture and the Motorola Book E standards.

6.1 Overview

The e200z6 memory management unit is a 32-bit PowerPC Book E–compliant implementation.

6.1.1 MMU Features

The MMU of the e200z6 core has the following feature set:

- Motorola Book E MMU architecture compliant
- 32-bit effective address translated to 32-bit real address (using a 41-bit interim virtual address)
- 32-entry fully associative translation lookaside buffer (TLB1) that supports the nine page sizes shown in Table 6-2
- One 8-bit PID register (PID0) for supporting up to 255 translation IDs at any time in the TLB
- No page table format defined; software is free to use its own page table format
- Hardware assist for TLB miss exceptions
- TLB1 managed by **tlbre**, **tlbwe**, **tlbsx**, **tlbsync**, and **tlbivax** instructions and six MMU assist (MAS) registers
- IPROT bit implemented in TLB1 prevents invalidations, protecting critical entries (so designated by having the IPROT bit set) from being invalidated.

6.1.2 TLB Entry Maintenance Features Summary

The TLB entries of the e200z6 core complex must be loaded and maintained by the system software; this includes performing any required table search operations in memory. The e200z6 provides support for maintaining TLB entries in software with the resources shown in Table 6-1. Note that many of these features are defined at the Motorola Book E level.

Table 6-1. TLB Maintenance Programming Model

Features		Description	More Information Section/Page
TLB Instructions	**tlbre**	TLB Read Entry instruction	6.4.1/6-10
	tlbwe	TLB Write Entry instruction	6.4.2/6-11
	tlbsx rA**, r**B	TLB Search for Entry instruction	6.4.3/6-11
	tlbivax rA**, r**B	TLB Invalidate Entries instruction	6.4.4/6-12
	tlbsync	TLB Synchronize Invalidations with other masters' instruction (privileged nop on the e200z6)	6.4.5/6-12
Registers	PID0	Process ID register	2.3.2/2-9
	MMUCSR0	MMU control and status register	2.14.1/2-59
	MMUCFG	MMU configuration register	2.14.2/2-60
	TLB0CFG–TLB1CFG	TLB configuration registers	2.14.3/2-61
	MAS0–MAS4, MAS6	MMU assist registers. Note that MAS5 is not implemented on the e200z6.	2.14.4/2-63
	DEAR	Data exception address register	2.7.1.5/2-21
Interrupts	Instruction TLB miss exception	Causes instruction TLB error interrupt	5.6.15/5-20
	Data TLB miss exception	Causes data TLB error interrupt	5.6.14/5-20
	Instruction permission violation exception	Causes ISI interrupt	5.6.4/5-13
	Data permission violation exception	Causes DSI interrupt	5.6.3/5-12

Other hardware assistance features for maintenance of the TLB on the e200z6 are described in Section 6.6.5.2, "MAS Register Updates."

6.2 Effective-to-Real Address Translation

This section describes the general principles that guide the PowerPC Book E definition for memory management, and further describes the structure for MMUs defined by the Motorola Book E implementation standard (EIS) and the e200z6 MMU.

Figure 6-1 illustrates the high-level translation flow, showing that because the smallest page size supported by the e200z6 core complex is 4 Kbytes, the least significant 12 bits always index within the page and are untranslated.

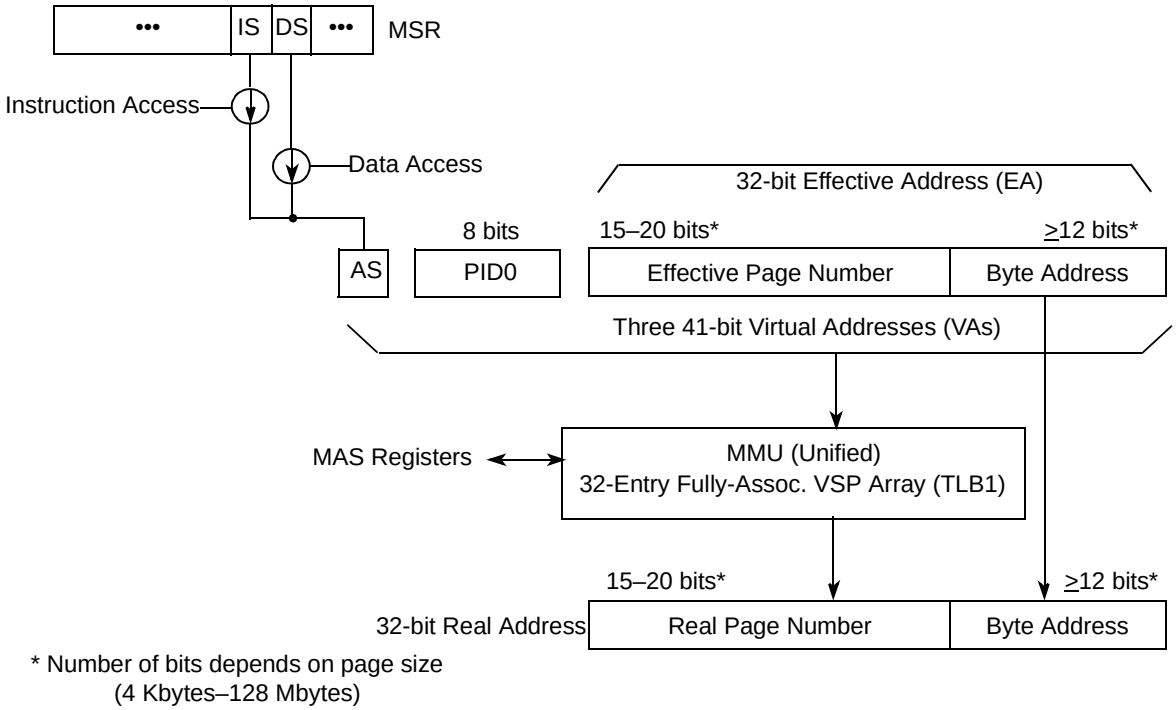


Figure 6-1. Effective-to-Real Address Translation Flow

6.2.1 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 e200z6 instruction fetch, branch, and load/store units generate 32-bit effective addresses. The MMU translates these effective addresses to 32-bit real addresses that are then used for memory accesses.

The PowerPC Book E 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 e200z6 MMU supports nine page sizes (4 Kbytes to 256 Mbytes, as defined in Table 6-2). 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 TLB. Accesses to addresses for which no TLB entry exists (a TLB miss) cause instruction or data TLB errors.

6.2.2 Address Spaces

The PowerPC Book E 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] (instruction address space bit) and MSR[DS] (data address space bit), respectively. The address space indicator (the corresponding value of either MSR[IS] or MSR[DS]) is used in addition to

the effective address generated by the processor for translation into a physical address by the TLB mechanism. Because MSR[IS] and MSR[DS] are both cleared to 0 when an interrupt occurs, an address space value of 0b0 can be used to denote interrupt-related address spaces (or possibly all system software address spaces), and an address space value of 0b1 can be used to denote non interrupt-related (or possibly all user address spaces) address spaces.

The address space associated with an instruction or data access is included as part of the virtual address in the translation process (AS).

6.2.3 Virtual Addresses and Process ID

The PowerPC Book E architecture defines that a process ID (PID) value is associated with each effective address (instruction or data) generated by the processor to construct a virtual address for each access. 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 e200z6 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.

The core complex implements a single process ID (PID) register, PID0, as an SPR shown in Section 2.14.5, “Process ID Register (PID0).” The current value in the PID register is used in the TLB look-up process and compared with the TID field in all the TLB entries. If the PID value in PID0 matches with a TLB entry in which all the other match criteria are met, that entry is used for translation.

Note that when a TID value in a TLB entry is all zeros, it always causes a match in the PID compare (effectively ignoring the values of the PID register). Thus, the operating system can set the values of all the TIDs to zero, effectively eliminating the PID value from all translation comparisons.

6.2.4 Translation Flow

The effective address, concatenated with the address space value of the corresponding MSR bit (MSR[IS] or MSR[DS]), is compared to the appropriate number of bits of the EPN field (depending on the page size) and the TS field of TLB entries. If the contents of the effective address plus the address space bit matches the EPN field and TS bit of the TLB entry, that TLB entry is a candidate for a possible translation match. In addition to a match in the EPN field and TS, a matching TLB entry must match with the current process ID of the access (in PID0), or have a TID value of 0, indicating that the entry is globally shared among all processes.

Figure 6-2 shows the translation match logic for the effective address plus its attributes, collectively called the virtual address, and how it is compared with the corresponding fields in the TLB entries.

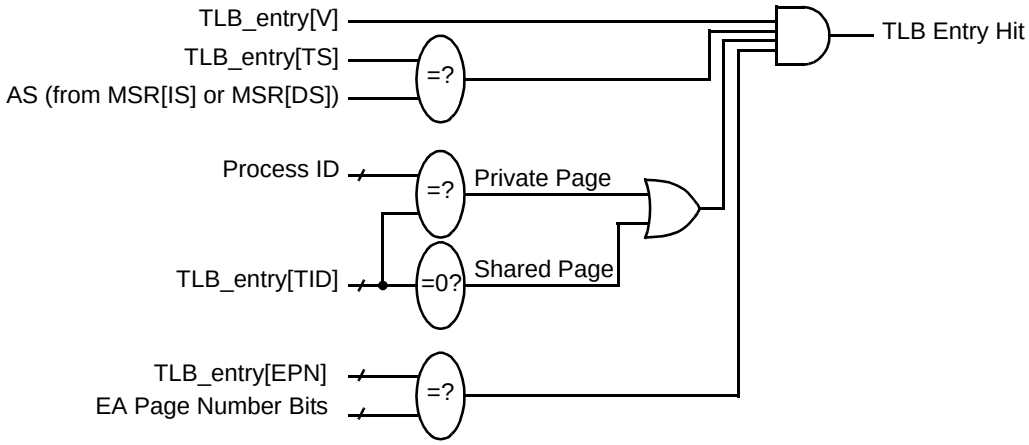


Figure 6-2. Virtual Address and TLB-Entry Compare Process

The page size defined for a TLB entry determines how many bits of the effective address are compared with the corresponding EPN field in the TLB entry as shown in Table 6-2. On a TLB hit, the corresponding bits of the real page number (RPN) field are used to form the real address.

Table 6-2. Page Size (for e200z6 Core) and EPN Field Comparison

SIZE Field	Page Size (4^{SIZE} Kbytes)	EA to EPN Comparison (Bits 32–53; 2 × SIZE)
0b0001	4 Kbytes	EA[32–51] = EPN[32–51]?
0b0010	16 Kbytes	EA[32–49] = EPN[0–49]?
0b0011	64 Kbytes	EA[32–47] = EPN[32–47]?
0b0100	256 Kbytes	EA[32–45] = EPN[32–45]?
0b0101	1 Mbyte	EA[32–43] = EPN[32–43]?
0b0110	4 Mbytes	EA[32–41] = EPN[32–41]?
0b0111	16 Mbytes	EA[32–39] = EPN[32–39]?
0b1000	64 Mbytes	EA[32–37] = EPN[32–37]?
0b1001	256 Mbytes	EA[32–35] = EPN[32–35]?

On a TLB hit, the generation of the physical address occurs as shown in Figure 6-1.

6.2.5 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 may be execute only, and data structures may be mapped as read/write/no-execute) and 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 6-3. If the access is not allowed by the access permission mechanism, the processor generates an instruction or data storage interrupt (ISI or DSI).

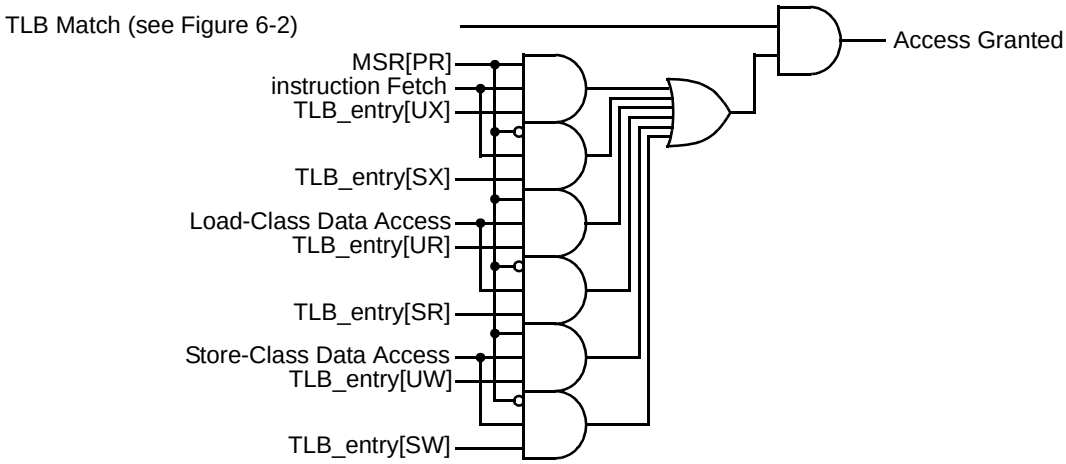


Figure 6-3. Granting of Access Permission

6.3 Translation Lookaside Buffer

The Motorola Book E 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 e200z6 MMU supports a single TLB that is fully associative and supports variable page sizes; thus it corresponds to TLB1 in the programming model.

The TLB on the e200z6 MMU (TLB1) consists of a 32-entry, fully-associative content-addressable memory (CAM) array with support for nine page sizes. To perform a lookup, the TLB is searched in parallel for a matching TLB entry. The contents of a matching 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 and TLB entries may be corrupted (an exception will not be reported).

The structure of TLB1 is shown in Figure 6-4.

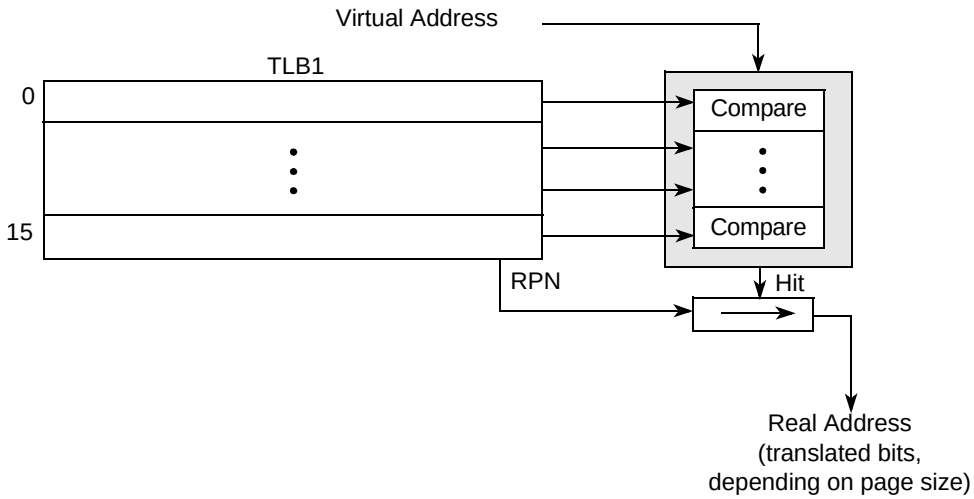


Figure 6-4. e200z6 TLB1 Organization

6.3.1 IPROT Invalidation Protection in TLB1

The IPROT bit in TLB1 is used to protect TLB entries from invalidation. TLB1 entries with IPROT set are not invalidated by a **tlbivax** instruction executed by this processor (even when the INV_ALL command is indicated), or by a flash invalidate initiated by writing to MMUCSR0[TLB1_FI]. The IPROT bit can be used to protect critical code and data such as interrupt vectors/handlers in order to guarantee that the instruction fetch of those vectors

never takes a TLB miss exception. Entries with IPROT set can only be invalidated by writing a 0 to the valid bit of the entry (by using the MAS registers and executing the **tlbwe** instruction).

Invalidation operations generated by execution of the **tlbivax** instruction are guaranteed to invalidate the entry that translates the address specified in the operand of the **tlbivax** instruction. Additional entries may also be invalidated by this operation if they are not protected with IPROT. A precise invalidation can be performed by writing a 0 to the valid bit of a TLB entry.

6.3.2 Replacement Algorithm for TLB1

The replacement algorithm for TLB1 must be implemented completely by system software. Thus, when an entry in TLB1 is to be replaced, the software can select which entry to replace and write the entry number to the MAS0[ESEL] field before executing a **tlbwe** instruction.

Alternately, the software can load the entry number of the next desired victim into MAS0[NV]. The e200z6 then automatically loads MAS0[ESEL] from MAS0[NV] on a TLB error condition as shown in Figure 6-5.

See Table 6-6 for a complete description of MAS register updates on various exception conditions.

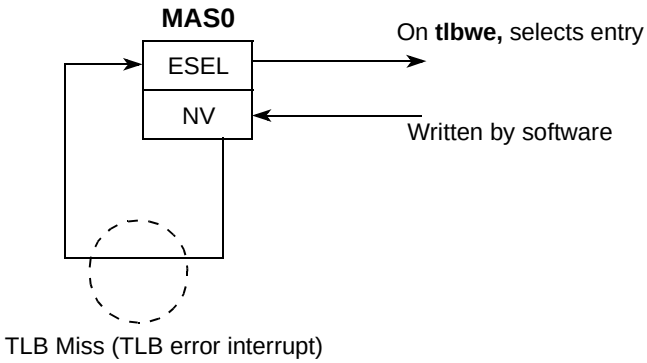


Figure 6-5. Victim Selection

6.3.3 TLB Access Time

The TLB array is checked for a translation hit in parallel with the on-chip L1 cache lookups, and no penalty on a TLB hit is incurred. If the TLB array misses, a TLB miss interrupt is reported.

6.3.4 The G Bit (of WIMGE)

The G bit provides protection from bus accesses that could be canceled due to an exception on a prior uncompleted instruction.

If G = 1 (guarded), these types of accesses must stall (if they miss in the cache) until the exception status of the instruction(s) in progress is known. If G = 0 (unguarded), these accesses may be issued to the bus regardless of the completion status of other instructions. Because the e200z6 does not make requests to the cache for load or store instructions until it is known that prior instructions will complete without exceptions, the G bit is essentially ignored. Proper operation always occurs to guarded memory.

6.3.5 TLB Entry Field Summary

Table 6-3 summarizes the fields of e200z6 TLB entries. Note that all of these fields are defined at the Motorola Book E level.

Table 6-3. TLB Entry Bit Fields for e200z6

Field	Description
V	Valid bit for entry
TS	Translation address space (compared with AS bit of the current access)
TID[0–7]	Translation ID (compared with PID0 or TIDZ (all zeros))
EPN[0–19]	Effective page number (compared with effective address)
RPN[0–19]	Real page number (translated address)
SIZE[0–3]	Encoded page size 0000 Reserved 0001 4 Kbytes 0010 16 Kbytes 0011 64 Kbytes 0100 256 Kbytes 0101 1 Mbyte 0110 4 Mbytes 0111 16 Mbytes 1000 64 Mbytes 1001 256 Mbytes All others—reserved
PERMIS[0–5]	Supervisor execute, write, and read permission bits, and user execute, write, and read permission bits.
WIMGE	Memory/cache attributes (write-through, cache-inhibit, memory coherence required, guarded, endian)
U0–U3	User attribute bits—used only by software
IPROT	Invalidation protection

6.4 Software Interface and TLB Instructions

TLB1 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 TLB1. 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, data TLB error, DSI, or ISI interrupt occurs.

On a TLB error interrupt, the MAS registers are written by hardware with the proper EA, default attributes (TID, WIMGE, permissions, and so on), 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, the hardware updates only the search PID (SPID) and search address space (SAS) fields in the MAS registers, using the contents of PID0 and the corresponding value of MSR[IS] or MSR[DS] that was used when the DSI or ISI exception was recognized. During the interrupt handler, software can issue a TLB search instruction (**tlbsx**), which uses the SPID field along with the SAS field, to determine the entry related to the DSI or ISI exception. Note that it is possible that the entry that caused the DSI or ISI interrupt no longer exists in the TLB by the time the search occurs if a TLB invalidate or replacement removes the entry between the time the exception is recognized and when the **tlbsx** is executed.

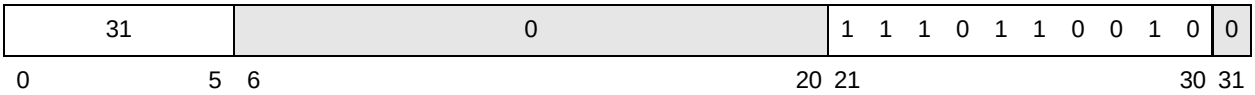
The **tlbre**, **tlbwe**, **tlbsx**, **tlbivax**, and **tlbsync** instructions are privileged.

6.4.1 TLB Read Entry Instruction (tlbre)

The TLB Read Entry instruction causes the contents 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 6-6 for details on how MAS register fields are updated.

tlbre **tlbre**

TLB Read Entry Form X



```
tlb_entry_id = MAS0(TLBSEL, ESEL)
result = MMU(tlb_entry_id)
MAS1, MAS2, MAS3 = result
```

6.4.2 TLB Write Entry Instruction (tlbwe)

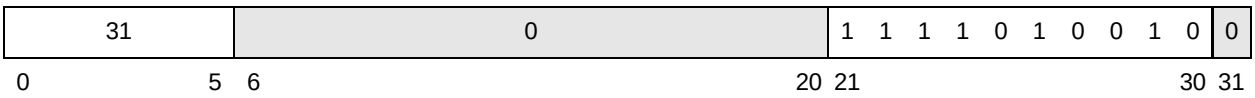
The TLB Write Entry instruction causes the contents of certain fields within the MMU assist registers MAS1, MAS2, and MAS3 to be written into a single TLB entry in the MMU. The entry written is specified by the TLBSEL and ESEL fields of the MAS0 register.

tlbwe

TLB Write Entry

tlbwe

Form X




```
tlb_entry_id = MAS0(TLBSEL, ESEL)
MMU(tlb_entry_id) = MAS1, MAS2, MAS3
```

6.4.3 TLB Search Indexed Instruction (tlbsx)

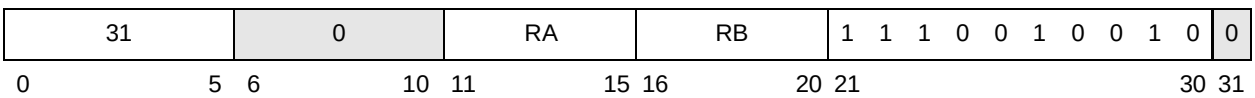
The TLB Search Indexed instruction updates the MMU assist registers conditionally based on success or failure of a lookup of the TLB. The lookup is controlled by an effective address provided by GPR[RB] as specified in the instruction encoding, as well as by the SAS and SPID search fields in MAS6. The values placed into MAS0, MAS1, MAS2, and MAS3 differ depending on a successful or unsuccessful search. See Table 6-6 for details on how MAS register fields are updated.

tlbsx

TLB Search Indexed

tlbsx

tlbsx RA,RB Form X




```
if RA!=0 then EA = GPR(RA) + GPR(RB)
else EA = GPR(RB)
ProcessID = MAS6(SPID), 8'b00000000
AS = MAS6(SAS)
VA = AS || ProcessIDs || EA
if Valid_TLB_matching_entry_exists(VA)
    then result = see  Table 6-6, column "tlbsx hit"
    else result = see  Table 6-6, column "tlbsx miss"
MAS0, MAS1, MAS2, MAS3 = result
```

6.4.4 TLB Invalidate (tlbivax) Instruction

The TLB invalidate operation is performed whenever a TLB Invalidate Virtual Address Indexed (**tlbivax**) instruction is executed. This instruction invalidates TLB entries which correspond to the virtual address calculated by this instruction. The address is detailed in Table 6-4. No other information except for that shown in Table 6-4 is used for the invalidation (the 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 one to ensure that 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 one).

The bits of EA used to perform the **tlbivax** invalidation of TLB1 are bits 0–19.

Table 6-4. tlbivax EA Bit Definitions

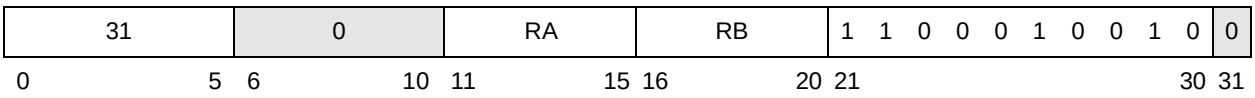
Bits	Field
0–19	EA[0–19]
20–27	Reserved [1]
28	TLBSEL (1 = TLB1) Should be set to '1' for future compatibility.
29	INV_ALL
30–31	Reserved [1]

[1] These bits should be zero for future compatibility. They are ignored.

tlbivax tlbivax

TLB Invalidate Virtual Address Indexed

tlbivax RA,RB Form X




```
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
    InvalidateTLB(VA)
```

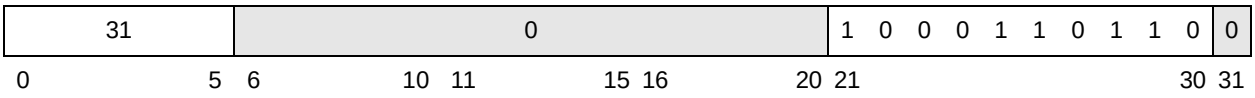
6.4.5 TLB Synchronize Instruction (tlbsync)

The TLB Synchronize instruction is treated as a privileged no-op by the e200z6.

tlbsync tlbsync

TLB Synchronize

tlbsync RA,RB Form X



6.5 TLB Operations

This section describes how the software (with some hardware assistance) maintains TLB1.

6.5.1 Translation Reload

The TLB reload function is performed in software with some hardware assistance. This hardware assistance consists of:

- 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.

6.5.2 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 TLB1, 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 are updated with the data from the selected TLB entry. See Section 6.4.1, “TLB Read Entry Instruction (tlbre),” for more information.

6.5.3 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 TLB1, the TLBSEL field in MAS0 must be set to 01, and the ESEL bits in MAS0 must be set to point to the desired entry. When the **tlbwe** instruction is executed, the TLB entry

information stored in MAS1–MAS3 is written into the selected TLB entry. See Section 6.4.2, “TLB Write Entry Instruction (**tlbwe**),” for more information.

6.5.4 Searching the TLB

TLB1 can be searched using the **tlbsx** instruction by first writing the necessary information into MAS6. The **tlbsx** instruction searches using EPN[0–19] 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. See Section 6.4.3, “TLB Search Instruction (**tlbsx**),” for more information.

6.5.5 TLB Coherency Control

The e200z6 core provides the ability to invalidate a TLB entry as described in the Book E PowerPC architecture. The **tlbivax** instruction invalidates local TLB entries only. No broadcast is performed, as no hardware-based coherency support is provided.

The **tlbivax** instruction invalidates by effective address only. This means that only the TLB entry’s EPN bits are used to determine if the TLB entry should be invalidated. It is therefore possible for a single **tlbivax** instruction to invalidate multiple TLB entries, since the AS and TID fields of the entries are ignored.

6.5.6 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[0–19] 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[0–19], 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 through **mtspr** before executing **tlbwe**. If the defaults are not applicable to the TLB entry being loaded, the TLB miss exception handler must update MAS0–MAS2 before performing the TLB write.

See Table 6-6 for more details on the automatic updates to the MAS registers on exceptions.

6.5.7 TLB Load on Reset

During reset, all TLB entries except entry 0 are automatically invalidated by the hardware. TLB entry 0 is also loaded with the default values in Table 6-5.

Table 6-5. 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[0–19]	value of *p_rstbase[0:19]*	Page address present on *p_rstbase[0:19]*. See Chapter 8, “External Core Complex Interfaces,” for more information.
RPN[0–19]	value of *p_rstbase[0:19]*	Page address present on *p_rstbase[0:19]*. See Chapter 8, “External Core Complex Interfaces,” for more information.
SIZE[0–3]	0001	4KB page size
SX/SW/SR	111	Full supervisor mode access allowed
UX/UW/UR	111	Full user mode access allowed
WIMG	0100	Cache inhibited, non-coherent
E	value of *p_rst_endmode*	Value present on *p_rst_endmode*. See Chapter 8, “External Core Complex Interfaces,” for more information.
U0–U3	0000	User bits
IPROT	1	Page is protected from invalidation

6.6 MMU Configuration and Control Registers

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. Additionally, there are a number of MMU control and assist registers summarized in Section 2.14.4, “MMU Assist Registers (MAS0–MAS4, MAS6).”

6.6.1 MMU Configuration Register (MMUCFG)

The MMUCFG register provides configuration information for the MMU supplied with this version of the e200z6 CPU core. A description of the MMUCFG register can be found in Section 2.14.2, “MMU Configuration Register (MMUCFG).”

6.6.2 TLB0 and TLB1 Configuration Registers

The TLB0CFG and TLB1CFG registers provide configuration information for the MMU TLBs supplied with this version of the e200z6 CPU core. A description of these registers can be found in Section 2.14.3, “TLB Configuration Registers (TLBnCFG).”

6.6.3 DEAR Register

The data exception address register (DEAR) is loaded with the effective address of the data access which results in an alignment, data TLB miss, or DSI exception. A description of DEAR can be found in Section 2.7.1.5, “Data Exception Address Register (DEAR).”

6.6.4 MMU Control and Status Register 0 (MMUCSR0)

The MMU control and status register 0 (MMUCSR0) is a 32-bit register that controls the state of the MMU. The MMUCSR0 register is shown in Section 2.14.1, “MMU Control and Status Register 0 (MMUCSR0).”

6.6.5 MMU Assist Registers (MAS)

The e200z6 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 e200z6 does not implement the MAS5 register, present in other Motorola Book E designs, because the **tlbsx** instruction only searches based on a single SPID value.

The MAS registers are shown in detail in Section 2.14.4, “MMU Assist Registers (MAS0–MAS4, MAS6).”

6.6.5.1 MAS Registers Summary

The fields of the MAS registers are summarized in Figure 6-6.

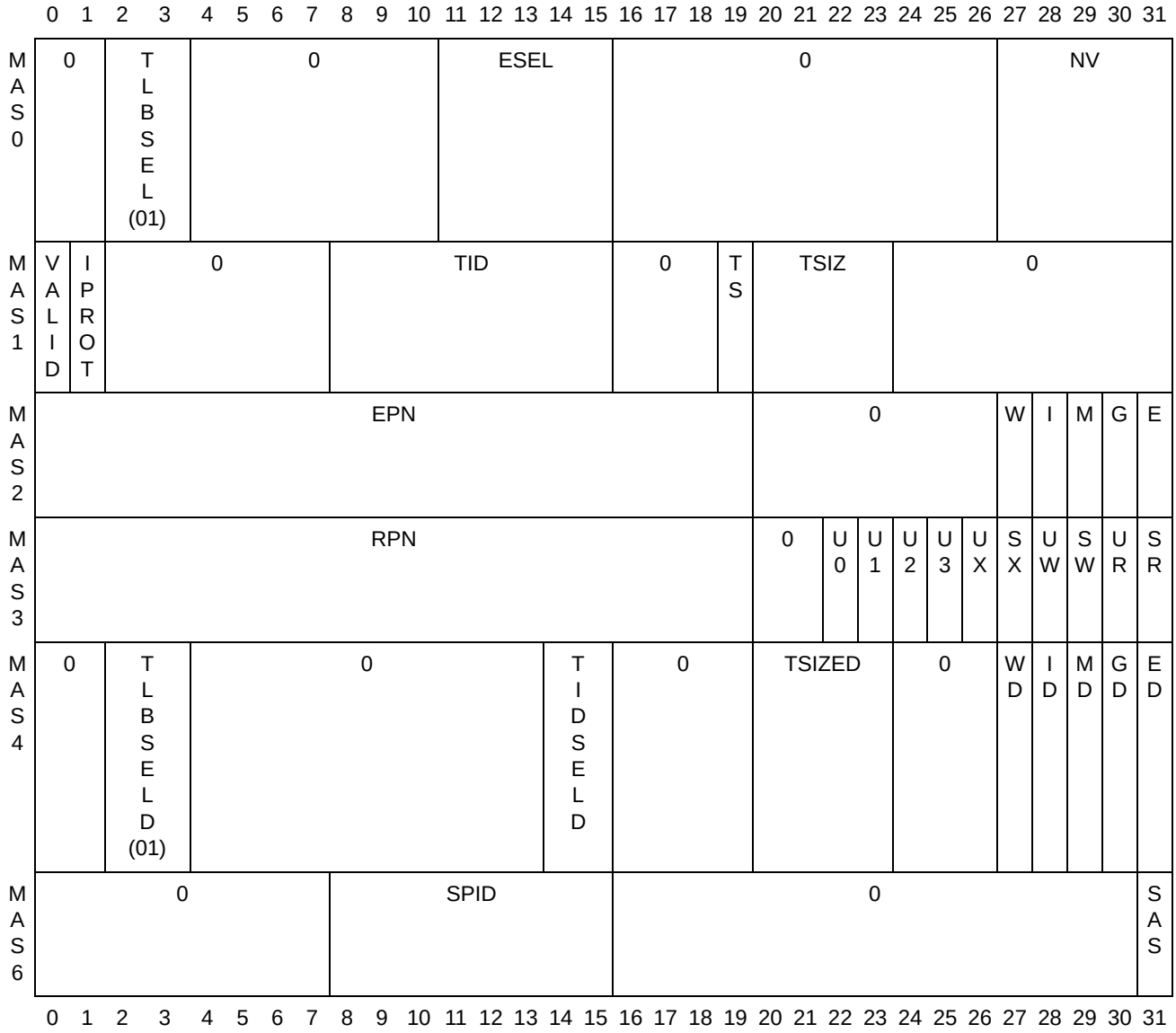


Figure 6-6. MMU Assist Registers Summary

6.6.5.2 MAS Register Updates

Table 6-6 details the updates to each MAS register field for each update type.

Table 6-6. MMU Assist Register Field Updates

Bit/Field	MAS Affected	Instr/Data TLB Error	tlbsx hit	tlbsx miss	tlbre	tlbwe	ISI/DSI
TLBSEL	0	TLBSELD	'01'	TLBSELD	NC [1]	NC	NC
ESEL	0	NV	Matched entry	NV	NC	NC	NC
NV	0	NC	NC	NC	NC	NC	NC

Table 6-6. MMU Assist Register Field Updates (continued)

Bit/Field	MAS Affected	Instr/Data TLB Error	tlbsx hit	tlbsx miss	tlbre	tlbwe	ISI/DSI
VALID	1	1	1	0	V(array)	NC	NC
IPROT	1	0	Matched IPROT	0	IPROT(array)	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[0–3]	1	TSIZED	TSIZE(array)	TSIZED	TSIZE(array)	NC	NC
EPN[0–19]	2	I/D EPN	EPN(array)	**tlbsx** EPN	EPN(array)	NC	NC
WIMGE	2	Default values	WIMGE(array)	Default values	WIMGE(array)	NC	NC
RPN[0–19]	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[0–3]	4	NC	NC	NC	NC	NC	NC
Default WIMGE	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

[1] NC—no change

6.7 Effect of Hardware Debug on MMU Operation

Hardware debug facilities use normal CPU instructions to access register and memory contents during a debug session. If desired during a debug session, the debug firmware may disable the translation process and may substitute default values for the access protection (UX, UR, UW, SX, SR, SW) bits and values obtained from the OnCE control register for page attribute (W, I, M, G, E) bits normally provided by a matching TLB entry. In addition, no address translation is performed, and instead, a 1:1 mapping of effective to real addresses is performed. When disabled during the debug session, no TLB miss or TLB access protection-related DSI conditions 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) remains in effect. Refer to Section 10.5.5.3, “e200z6 OnCE Control Register (OCR),” for more details on controlling MMU operation during debug sessions.

Chapter 7
Instruction Pipeline and Execution Timing

This section describes the e200z6 instruction pipeline and instruction timing information.

7.1 Overview of Operation

Figure 7-1 is a block diagram of the e200z6 core.

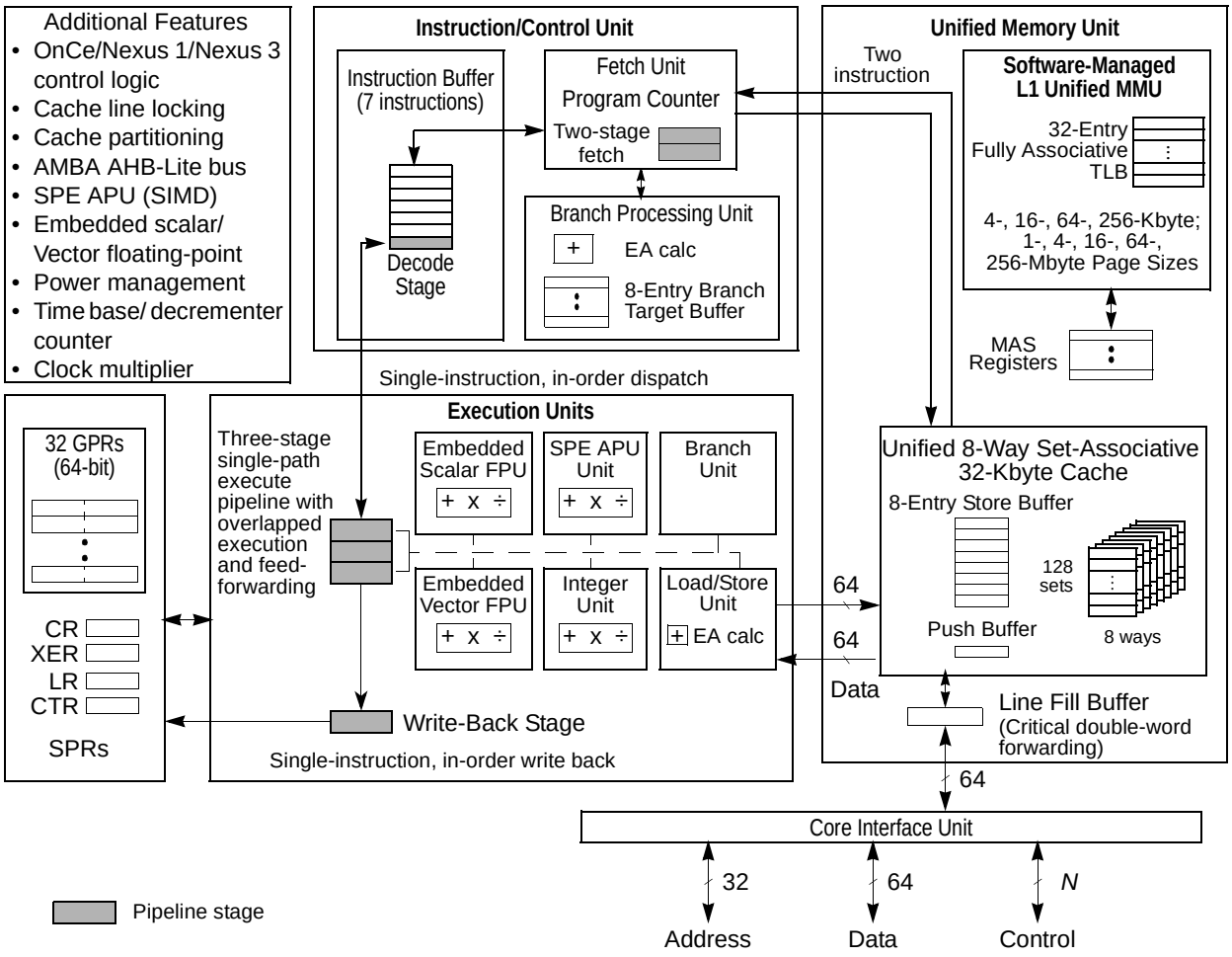


Figure 7-1. e200z6 Block Diagram

The e200z6 core contains the following major subsystems:

- Instruction unit—Fetches instructions from memory into the instruction buffer. Includes logic that predicts branches.
- Control unit—Coordinates the instruction fetch, branch, instruction decode, instruction dispatch, and completion units and the exception handling logic
- Decode unit—Decodes each instruction and generates information needed by the branch unit and the execution units. Branch target instructions are written into the branch target refetch buffers, while sequentially fetched instructions are written into the instruction buffer.
- Execution units:
  - Integer unit—Executes all computational and logical instructions
  - Load/store unit (LSU)—Executes loads, stores, and other memory access instructions
  - The e200z6 implements the following optional units:
    - SPE APU unit—Executes all SPE logical and computational instructions. Note that SPE vector load and store instructions are executed by the LSU.
    - Embedded vector and scalar single-precision floating-point units—Execute all logical and computational floating-point instructions defined by the Motorola Book E architecture. Note that the e200z6 does not implement the Book E–defined floating-point instructions.
  - Branch unit—Resolves branch prediction and updates the CTR and LR, where required.

  The executions are described in detail in Section 7.2.3, “Execute Stages.”
- Core interface—Provides the interface between the core and the system integrated logic.

The e200z6 core dispatches a single instruction each cycle to the three-stage execute pipeline. All instructions must pass through each of the three execute stages, although most computational instructions have a single-cycle latency and can feed forward their results on the next clock cycle, even through the instruction still occupies a position in the execute pipeline.

Source operands for each instruction are provided from the general-purpose registers (GPRs) or by the operand feed-forwarding mechanism. Data or resource hazards may create stall conditions that cause instruction dispatch to be stalled for one or more cycles until the hazard is eliminated.

The execution units write the results of a finished instruction onto the proper result bus and into the destination registers. The write-back (or completion) logic retires an instruction when the instruction has vacated the execution stages and written back results to the architected registers. Up to two results can be simultaneously written for a single

instruction (for example, load with update instructions (**lwhu** and **lwzu**) or a Load Multiple Word (**lmw**) instruction.

7.1.1 Instruction Unit

The instruction unit controls the flow of instructions from the cache to the instruction buffer and decode unit. A seven-entry instruction buffer and a two-entry branch target instruction prefetch buffer allow the instruction unit to fetch instructions and decouple memory from the execution pipeline.

- Control logic coordinates the instruction fetch unit, branch unit, instruction decode unit, instruction dispatch unit, completion unit, and exception handling logic.
- The branch processing unit predicts conditional branches and provides branch target addresses for instruction fetches.
- The instruction decode unit includes the seven-entry instruction buffer, into which instructions are fetched. A single instruction per processor clock cycle can be decoded and dispatched from the bottom entry of this buffer. The major functions of the decode logic are as follows:
  - — Opcode decoding to determine the instruction class and resource requirements for the decoded instruction
  - — Source and destination register dependency checking
  - — Execution unit assignment
  - — Determination of any decode serialization that would inhibit subsequent instructions from decoding and dispatching. Serialization is described in Section 7.5, “Instruction Serialization.”

  The decode latency is a single processor clock cycle. Although it takes one clock cycle to decode an instruction, an instruction cannot dispatch for any of the following reasons:
  - — The previous instruction is either dispatch or prefetch serializing
  - — Space is not available in the execution unit (for example, if a divide instruction is executing)
  - — A multiple-cycle instruction (such as a multiply) with a following data dependency is present
- The exception handling unit includes logic to handle exceptions, interrupts, and traps.

7.2 Instruction Pipeline

The seven-stage processor pipeline is shown in Figure 7-2.

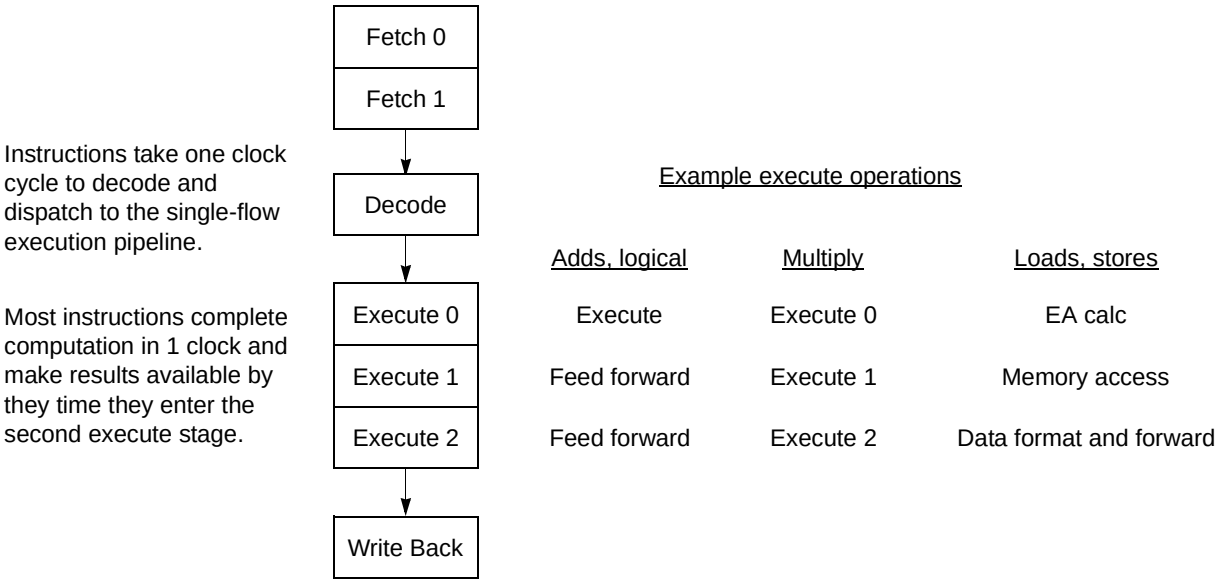


Figure 7-2. Seven-Stage Instruction Pipeline

The instruction pipeline consists of the following stages:

- Two instruction fetch stages. As many as two instructions are fetched per clock cycle and placed in the lowest available entries in the seven-entry instruction buffer. An instruction can be fetched directly into the bottom entry, in which instruction decoding takes place. The decode stage is as follows.
- Instruction decode/dispatch stage. Each instruction takes 1 cycle to decode and is dispatched at the end of the decode stage. Instructions are dispatched in order when
- A three-stage execution pipeline that includes feed-forwarding, which allows dependent instructions to continue through the pipeline.

  All instructions, including branch instructions, pass through all three stages of the execute pipeline in order and in single-file. Any combination of up to three of the six execute units can participate in the three-stage pipeline per clock cycle. The execution units are as follows:

  — Integer unit—Executes all integer logical and computational instructions

  — Load/store unit (LSU)—Executes all load and store instructions and all instructions that affect memory, such as cache and TLB management instructions. These include the 64-bit load and store instructions that are defined by the SPE APU and by the embedded vector floating-point APU.

  — Signal processing engine (SPE) APU unit—Executes all logical and computational instructions defined by the Motorola Book E SPE APU instruction set, except vector loads and stores, which are handled by the LSU.

  — Embedded vector single-precision floating-point unit—executes all logical and computational instructions defined by the Motorola Book E embedded vector

single-precision floating-point APU, except for the vector load and store instructions that are shared by the SPE APU.

— Embedded scalar single-precision floating-point unit—executes all logical and computational instructions defined by the Motorola Book E embedded scalar single-precision floating-point APU.

— Branch unit—executes branch instructions. All branch instructions, including unconditional branch instructions, pass through all three execute pipeline stages.

Single-cycle instructions (that are not synchronizing and do not have special serialization requirements) finish executing after the first stage and make results available immediately so subsequent dependent instructions can enter the three-stage pipeline without waiting for the write-back stage.

Instructions pass through the instruction pipeline in order and in single file. At a given time, any combination of up to three execute units can be in use, although parallel execution is not supported per individual stage in the three-stage execute pipeline.

- A result write back stage, in which results are committed to architected registers (such as GPRs) and instructions are deallocated from the instruction pipeline.

Instructions pass single-file through the pipeline, decoding, executing, and writing back in order with a maximum throughput of one instruction per processor clock cycle.

Instructions pass through the seven pipeline stages as shown in Figure 7-3.

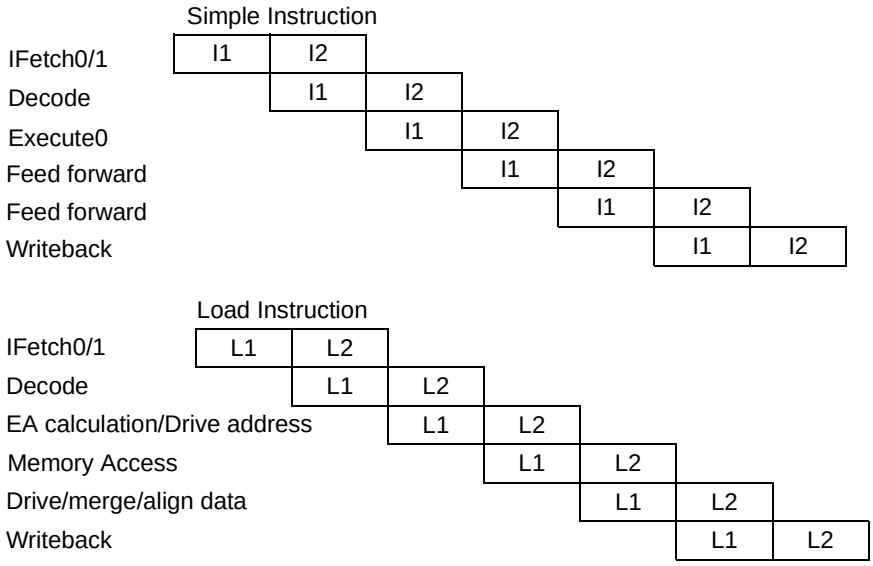


Figure 7-3. Pipeline

7.2.1 Fetch Stages

The fetch pipeline stages retrieve instructions from the memory system and determine where the next instruction fetch is performed. Up to two instructions every cycle are sent from memory to the instruction buffer.

7.2.1.1 Instruction Buffer

The e200z6 contains a seven-entry instruction buffer. The bottom entry, referred to as the instruction register (IR), is where an instruction resides for the decode stage.

Instruction fetches request a 64-bit double-word and the 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 (single-instruction) fetch is performed to load the instruction buffer. This 32-bit fetch may be immediately followed by a 64-bit fetch to fill slots 0 and 1 in case the branch is resolved to be taken.

In normal sequential execution, instructions are loaded into the IR from slot 0, and as a pair of slots are emptied, they are refilled (for maximum fetch rate of two instructions every two cycles). Whenever a pair of slots is empty, a 64-bit fetch is initiated, which fills the lowest empty slot pairs beginning with slot 0.

If the instruction buffer empties, dispatch stalls until the buffer is refilled. The first returned instruction is forwarded directly to the IR. Open cycles on the memory bus are used to keep the buffer full when possible, but bus priority is given to data loads and stores.

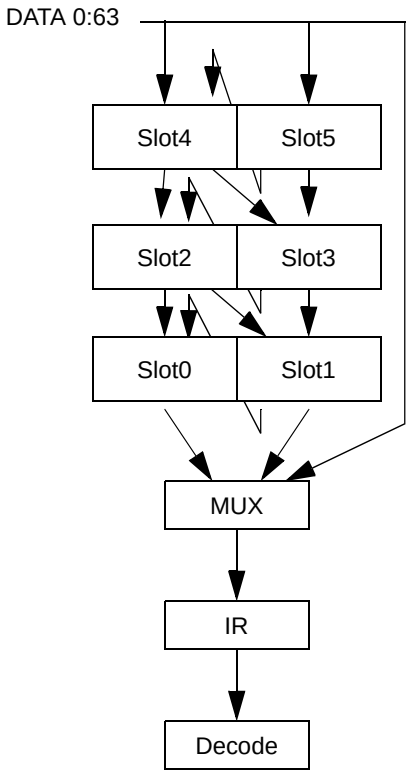


Figure 7-4. e200z6 Instruction Buffer

7.2.1.2 Branch Target Buffer (BTB)

To resolve branch instructions and improve the accuracy of branch predictions, the e200z6 implements a dynamic branch prediction mechanism using an eight-entry branch target buffer (BTB), a fully associative address cache of branch target addresses. The BTB on the e200z6 is purposefully small to reduce cost and power. It is expected to accelerate the execution of loops with some potential change of flow within the loop body.

A BTB entry is allocated whenever a branch resolves as taken and the BTB is enabled. Branches that have not been allocated are always predicted as not taken. BTB entries are allocated on taken branches using a FIFO replacement algorithm.

Each BTB entry holds 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.

A branch is predicted as taken on a hit in the BTB with a counter value of strongly or weakly taken. In this case the target address 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 mispredicted branch, the instruction fetch stream returns to the sequential 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 counter value is updated. 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 e200z6 does not implement the static branch prediction that is defined by the PowerPC architecture. The prediction bit in the BO operand is ignored.

Dynamic branch prediction is enabled by setting BUCSR[BPEN]. Clearing BUCSR[BPEN] disables dynamic branch prediction, in which case the e200z6 predicts every branch as not taken. Additional control is available in HID0[BPRED] to control whether forward or backward branches (or both) are candidates for entry into the BTB, and thus for branch prediction. After a branch is in the BTB, HID0[BPRED] has no further effect on that branch entry.

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 e200z6 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 BUCSR[BBFI].

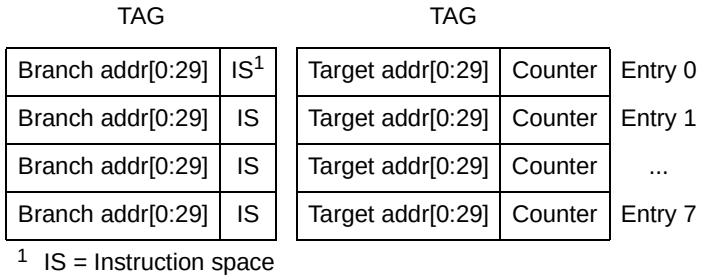


Figure 7-5. e200z6 Branch Target Buffer

The valid bit in each BTB entry is zero (invalid) at reset. When a branch instruction first enters the instruction pipeline, it is not allocated in the BTB and so by default is predicted as not taken. It is not allocated in the BTB until it is taken; the initial prediction is weakly taken, as shown in the example in Figure 7-6.

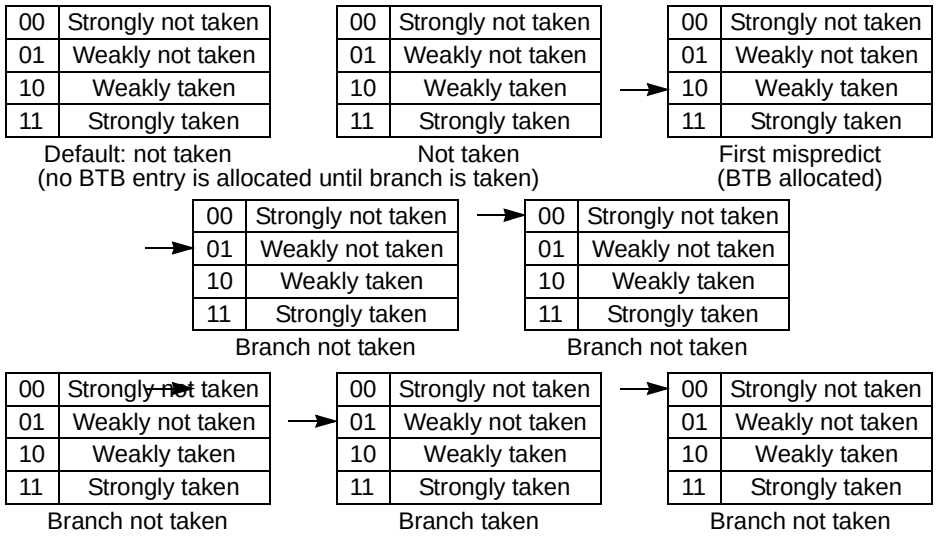


Figure 7-6. Updating Branch History

7.2.2 Decode Stage

The decode pipeline stage decodes instructions and checks for dependencies.

7.2.3 Execute Stages

Most instructions occupy only one of the three stages in the execute pipeline in any given clock cycle. For example, the following instruction sequence passes through the pipeline as shown in Figure 7-7:

```
ldx r1,r2,r3
add r5,r6,r7
mulli r8,r5,r6 /r5 depends on the results of add)
```

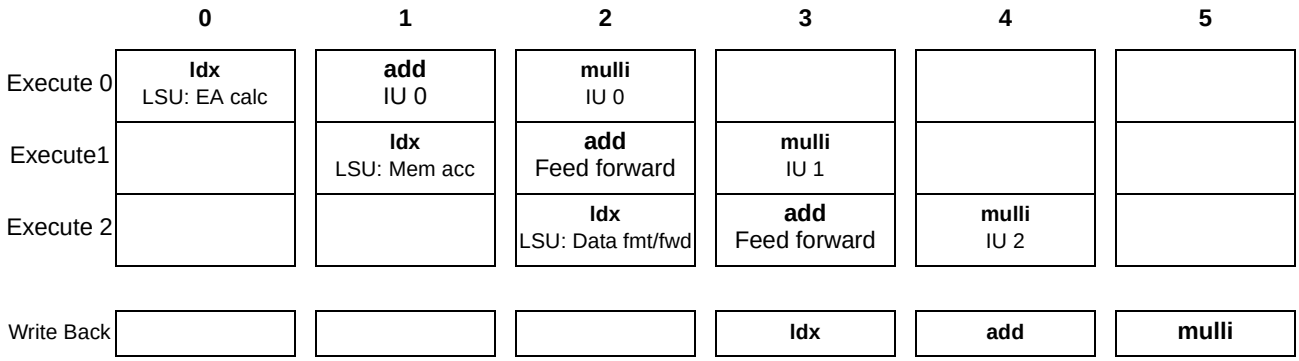


Figure 7-7. Pipelining—Execute and Write Back Stages

The execution of this sequence is described as follows:

- Clock cycle 0: The **ldx** instruction enters execute stage 0, during which the LSU calculates the effective address of the load.

- Clock cycle 1: The **ldx** instruction enters execute stage 1, during which the LSU initiates the memory access.

  The single-cycle **add** instruction is executed by the integer unit.
- Clock cycle 2: The **ldx** instruction enters execute stage 2, during which the LSU handles data formatting and forwarding.

  The results of the **add** instruction (in **r**5) are made available to **mulli**.

  The 3-cycle **mulli**, which is executed by the IU, enters execute stage 0.
- Clock cycle 3: The **ldx** instruction writes back its results to the destination GPR (**r**1) and is deallocated from the pipeline.

  The results of the **add** instruction remain available to any subsequent dependent instructions.

  The **mulli** instruction enters execute stage 1.
- Clock cycle 4: The **add** instruction writes back its results to its destination GPR (**r**5).

  mulli enters execute stage 2.
- Clock cycle 5: The **mulli** instruction writes back its results to its destination GPR (**r**8).

The execution pipeline is closely coupled with registers required for execution. Some of these registers are specified explicitly by the instruction (GPRs, condition register (CR), link register (LR), count register (CTR)). Others, such as the integer exception register (XER) and the SPE/embedded floating-point status and control register (SPEFSCR), and the CR are accessed implicitly, typically to record conditions or arithmetic exceptions.

Instructions are dispatched when all register resources are allocated and data dependencies are resolved. Most instructions perform their calculations in a single clock cycle, as shown in the latency tables in Section 7.7, “Instruction Timings.” Results are made available by the next clock cycle, so even though architected registers (such as GPRs) are not updated until instructions pass through the three-stage execution pipeline, dependent instructions do not have to wait for the instruction to pass through the entire three-stage execution pipeline. Therefore, a sequence of instructions with single-cycle latency can have a throughput of one instruction per cycle, even though each instruction must spend a cycle in each stage of the three-stage pipeline.

Note that although the core has multiple execution units for integer, SPE, embedded floating-point, branch, and load/store instructions, parallel instruction issue is not supported.

7.2.3.1 Integer Execution Unit

The integer execution unit executes integer arithmetic and logical instructions. Adds, subtracts, compares, count leading zeros, shifts, and rotates execute in a single cycle.

Multiply instructions have a 3-cycle latency with a maximum throughput of one per cycle. A sequence of floating-point instructions has a one-instruction-per-cycle throughput as long as they have no data dependencies.

Divide instructions have a variable latency of 6–16 cycles depending on the operand data. The worst-case integer divide takes 16 cycles. During execution of the divide, the integer execution pipeline is unavailable for additional instructions (blocking divide).

7.2.3.2 SPE Execution Unit

The SPE execution unit accesses the entire 64 bits of the GPRs to execute all SPE computational and logical instructions. Instruction latency and throughput for these instructions are similar to those for integer instructions: single-cycle latencies for most computations except for multiplication and division. Latency and throughput for these instructions are listed in Section 7.7.1, “SPE and Embedded Floating-Point APU Instruction Timing.”

SPE load/store operations are handled by the LSU, and have the same latencies and throughput as normal loads and stores.

7.2.3.3 Embedded Floating-Point Execution Units

The floating-point units execute all floating-point computational and comparison instructions defined by the embedded floating-point APUs. Instruction latency and throughput for these instructions are similar to those for integer instructions: single-cycle latencies for most computations except for multiplication and division. Latency and throughput as for these instructions are listed in Section 7.7.1, “SPE and Embedded Floating-Point APU Instruction Timing.”

7.2.3.4 Load/Store Unit (LSU)

The LSU 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 3-cycle latency. Stores also execute with a maximum throughput of one per cycle and 3-cycle latency. Store data can be fed forward from an immediately preceding load with no stall.

The LSU also executes all TLB and cache instructions.

7.2.3.5 Branch Execution Unit

Although branch prediction and redirection is handled in the fetch stages, branch execution consists of resolving the prediction and updating any registers (such as the LR or CTR). In certain cases, the branch unit predicts branches and supplies a speculative instruction stream to the instruction buffer.

7.3 Pipeline Drawings

Simple integer instructions, such as addition and logical instructions, complete execution in the execute 0 stage of the pipeline. Load and store instructions use all three execute stages, but may be dispatched one per clock assuming no data dependencies exist. Multiply instructions require three execute stages, but may also be pipelined unless there are data dependency stalls. 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 operands of a subsequent 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 all three execute stages into the end of the decode stage for a subsequent data-dependent instruction.

7.3.1 Pipeline Operation for Instructions with Single-Cycle Latency

Sequences of single-cycle execution instructions follow the flow in Figure 7-8. Instructions are dispatched and completed in program order. Most arithmetic and logical instructions fall into this category.

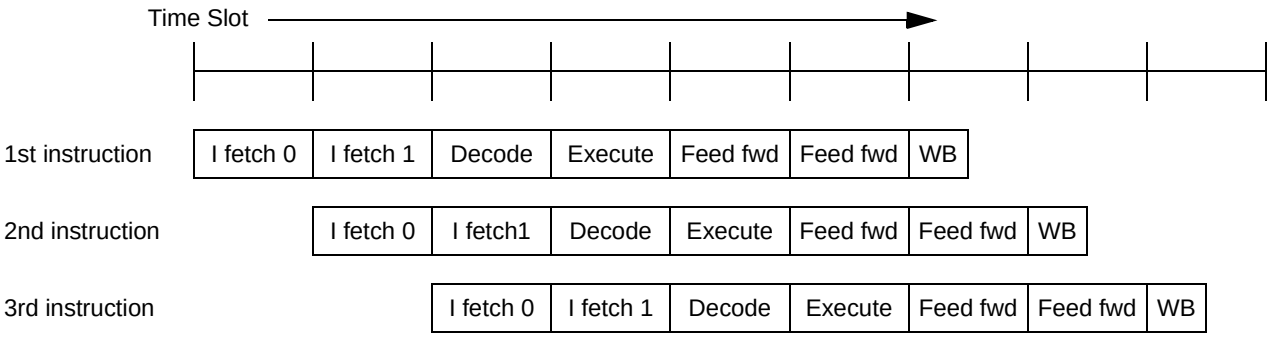


Figure 7-8. Basic Pipeline Flow, Single-Cycle Instructions

7.3.2 Basic Load and Store Instruction Pipeline Operation

Load and store instructions require a minimum of three cycles in the execute stages. For load and store instructions, the effective address is calculated in the Ex0/EA calculation stage, and memory is accessed in the Ex1/Mem0 and Ex2/Mem1 stages.

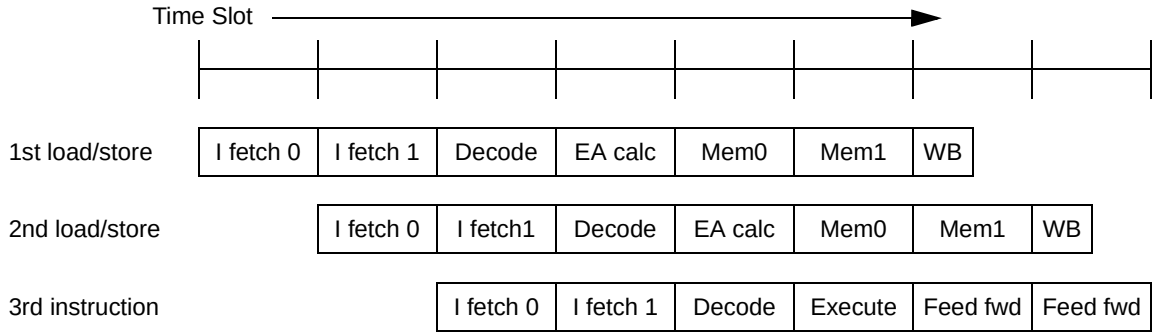


Figure 7-9. Basic Pipeline Flow, Load and Store Instructions

7.3.3 Change-of-Flow Instruction Pipeline Operation

Simple change-of-flow instructions require 3 cycles to refill the pipeline with the target instruction for taken branches and branch and link instructions with no prediction.

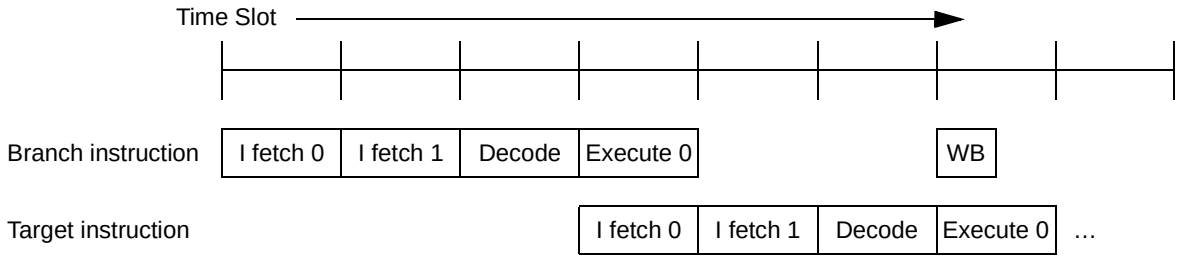


Figure 7-10. Basic Pipeline Flow, Branch Instructions

In some situations, the 3-cycle timing for branch-type instructions can be reduced by performing the target fetch speculatively while the branch instruction is still being fetched into the instruction buffer. The branch target address is obtained from the BTB. The resulting branch timing reduces to a single clock when the target fetch is initiated early enough and the branch is taken.

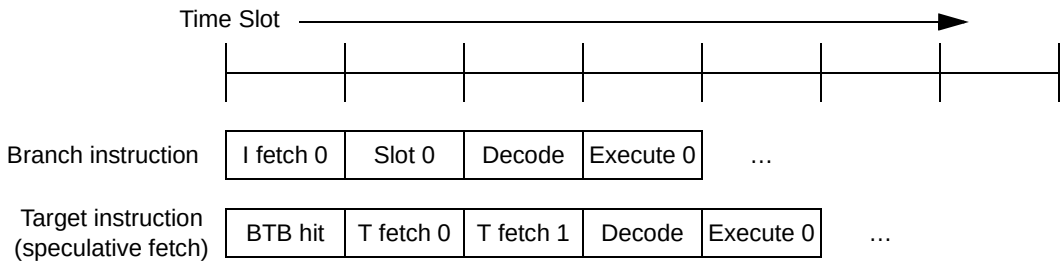


Figure 7-11. Basic Pipeline Flow, Branch Speculation

7.3.4 Basic Multiple-Cycle Instruction Pipeline Operation

The divide and load and store multiple instructions require multiple cycles in the execute stage.

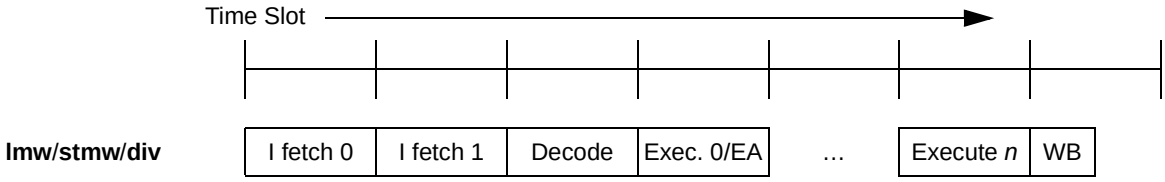


Figure 7-12. Basic Pipeline Flow, Multiple-Cycle Instructions

Most multiple-cycle instructions can be pipelined so that, for a series of multiple-cycle instructions, the effective execution time, or throughput, is smaller than the overall number of clocks spent in execution. This pipelining is allowed as long as no data dependencies exist between the instructions. Instructions must complete and write back results in order. To meet this requirement, a single-cycle instruction that follows a multiple-cycle instruction must wait for completion of the multiple-cycle instruction before it can write back. Result feed-forward paths are provided so that execution may continue prior to result write back.

7.3.5 Additional Examples of Instruction Pipeline Operation for Load and Store

Figure 7-13 shows an example of pipelining two non–data-dependent load or store instructions with a following non-data dependent single-cycle instruction. While the first load or store begins accessing memory in the MEM0 stage, the next load or store can be calculating a new effective address in the EX0/EA stage. The **add** in this example does not stall if no data dependency exists, but flows through two additional stages of feed-forwarding logic to complete in proper order.

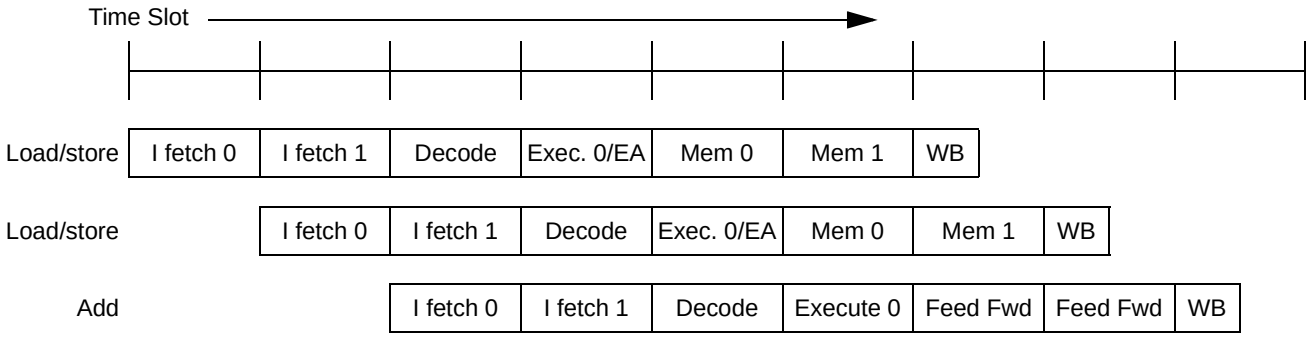


Figure 7-13. Pipelined Load/Store Instructions

For memory access instructions, wait-states may occur. This causes a subsequent memory access instruction to stall since the following memory access may not be initiated, as shown in Figure 7-14. Here, the first load/store instruction incurs a wait-state on the bus interface, causing succeeding instructions to stall.

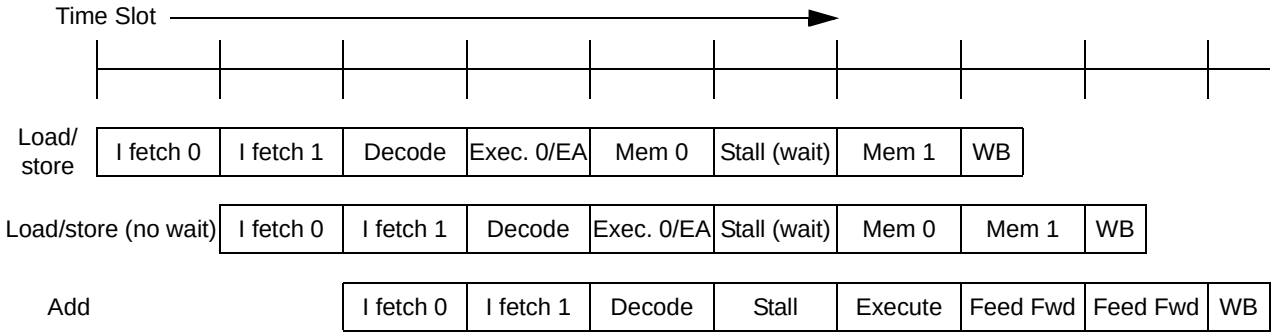


Figure 7-14. Pipelined Load/Store Instructions with Wait-State

Figure 7-15 shows an example of pipelining two non-data-dependent load or store instructions with a following load target data-dependent single-cycle instruction. While the first load or store begins accessing memory in the MEM0 stage, the next load can be calculating a new effective address in the EX0/EA stage. The **add** in this example stalls for 2 cycles because a data dependency exists on the target register of the second load.

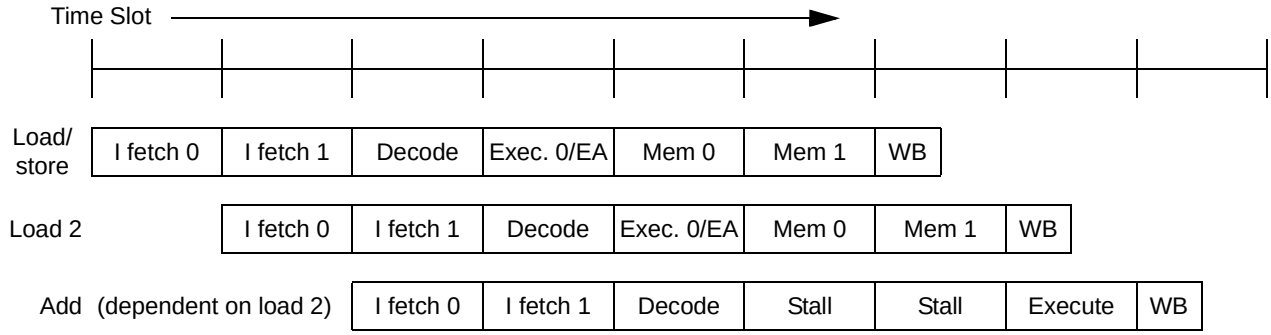


Figure 7-15. Pipelined Load Instructions with Load Target Data Dependency

Figure 7-16 shows an example of pipelining two non-data-dependent load or store instructions with a following single-cycle instruction that has a data dependency on the base register of a preceding load instruction with update. While the first load or store begins accessing memory in the MEM0 stage, the next load with update can be calculating a new effective address in the EX0/EA stage. Following the EA calculation, the updated base register value can be fed forward to subsequent instructions. The **add** in this example does not stall, even though a data dependency exists on the updated base register of the load.

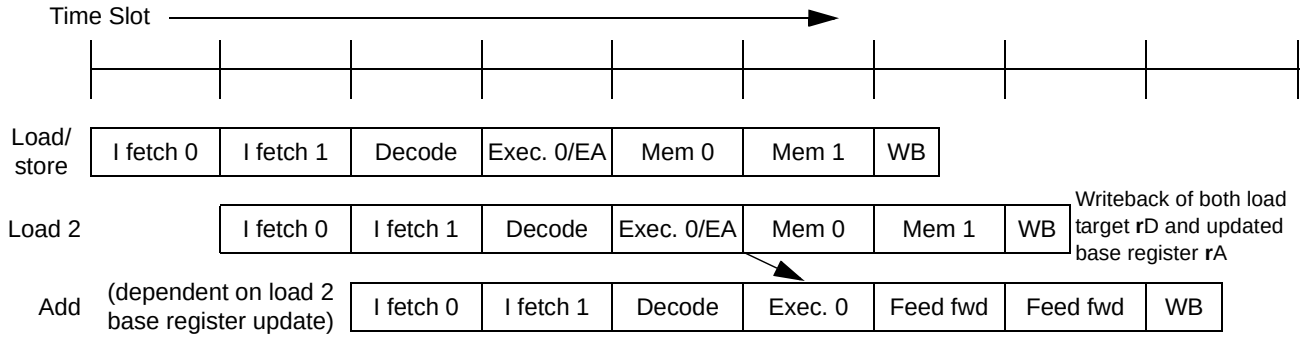


Figure 7-16. Pipelined Instructions with Base Register Update Data Dependency

7.3.6 Move to/from SPR Instruction Pipeline Operation

Most **mtspr** and **mfspr** instructions are treated like single-cycle instructions in the pipeline, and do not cause stalls. Exceptions are for the MSR, the debug SPRs, the optional embedded floating-point APUs, and Cache/MMU SPRs that do cause stalls. Figure 7-17 through Figure 7-19 show examples of **mtspr** and **mfspr** instruction timing.

Figure 7-17 applies to the debug SPRs and to the SPEFSCR. These instructions do not begin execution until all previous instructions have finished their execute stages. In addition, execution of subsequent instructions stalls until **mfspr** and **mtspr** complete.

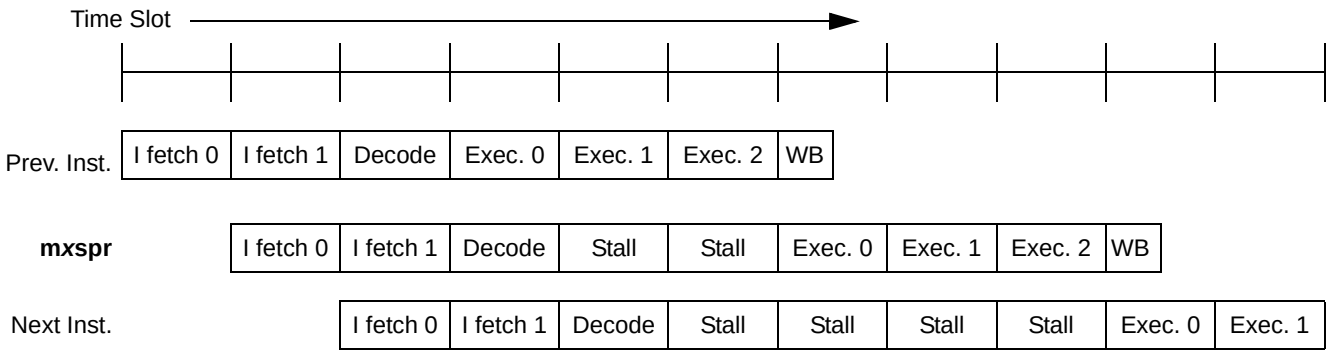


Figure 7-17. mtspr, mfspr Instruction Execution - (1)

Figure 7-18 applies to **mtmsr**, **wrtee**, and **wrteei**. Execution of subsequent instructions is stalled until these instructions write back.

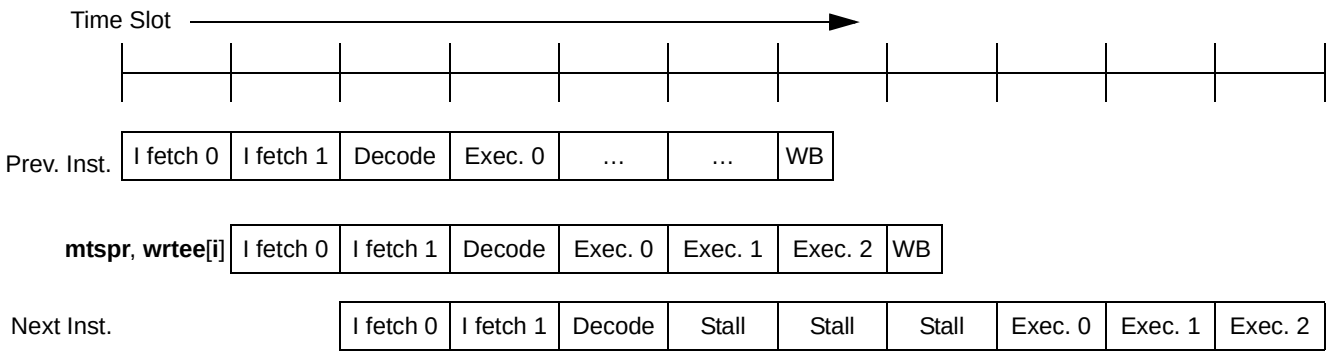


Figure 7-18. mtmsr, wrtee, and wrteei Execution

Access to cache and MMU SPRs stalls until all outstanding bus accesses have completed and the cache and MMU are idle (p_cmbusy negated) to allow an access window where no translations or cache cycles are required. Figure 7-19 shows an example where an outstanding bus access causes **mtspr**/**mfspr** execution to be delayed until the bus becomes idle. Other situations such as a cache line fill may cause the cache to be busy even when the processor interface is idle (*p_tbusy*[*0*]*_b* is negated). In these cases, execution stalls until the cache and MMU are idle, as signaled by negation of *p_cmbusy.* Processor access requests are held off during execution of a cache/MMU SPR instruction. A subsequent access request may be generated the cycle following the last execute stage (WB cycle). This same protocol applies to cache and MMU instructions (for example, **dcbz**, **dcbf**, **tlbre**, and **tlbwe**).

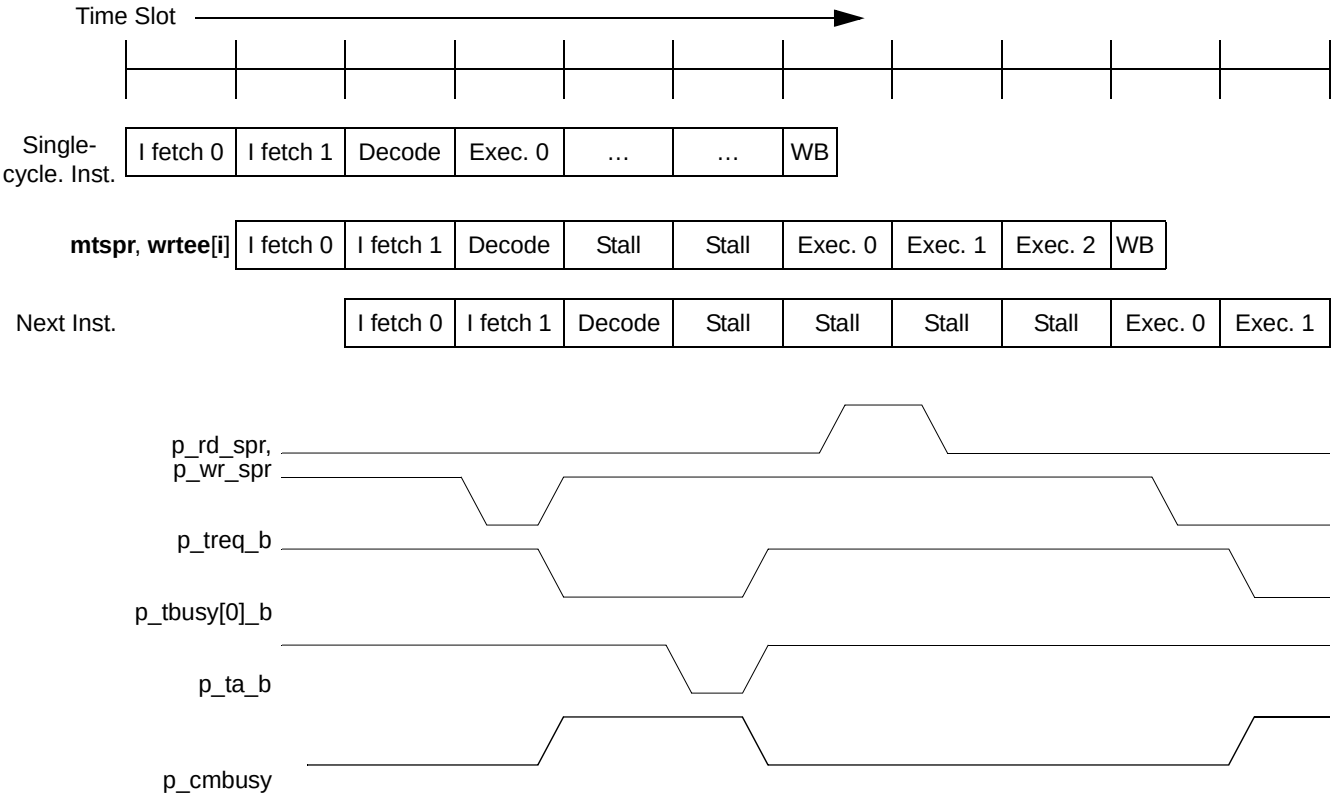


Figure 7-19. Cache/MMU mtspr, mfspr, and MMU Instruction Execution

7.4 Control Hazards

Several internal control hazards exist in the e200z6 that can cause certain instruction sequences to incur one or more stall cycles. These include the following cases:

- **mfspr** preceded by an **mtspr**—issue stalls until the **mtspr** completes

7.5 Instruction Serialization

There are three types of serialization required by the core:

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

7.5.1 Completion Serialization

A completion serialized instruction is held in the decode stage until all prior instructions have completed. The instruction finishes executing when it is next to complete in program order. Results from these instructions are not available for or forwarded to subsequent

instructions until the instruction completes. The following instructions are completion serialized:

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

7.5.2 Dispatch Serialization

A dispatch-serialized instruction prevents the next instruction from decoding until all instructions up to and including the dispatch-serialized instruction completes. The **isync**, **mbar**, **msync**, **rfi**, **rfci**, **rfdi**, and **sc** instructions are dispatch serialized. (Note that all of these instructions, except **mbar**, are also completion serialized.)

7.5.3 Refetch Serialization

Refetch serialized instructions inhibit dispatching of subsequent instructions and force a pipeline refill to refetch subsequent instructions after completion. These include the following instructions (which are also serialized at completion and dispatch):

- The context synchronizing instruction, **isync**.
- The **rfi**, **rfci**, **rfdi**, and **sc** instructions.

7.6 Interrupt Recognition and Exception Processing

Figure 7-20 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 sequence of single-cycle instructions. The handler is present in the cache and proceeds with no bubbles.

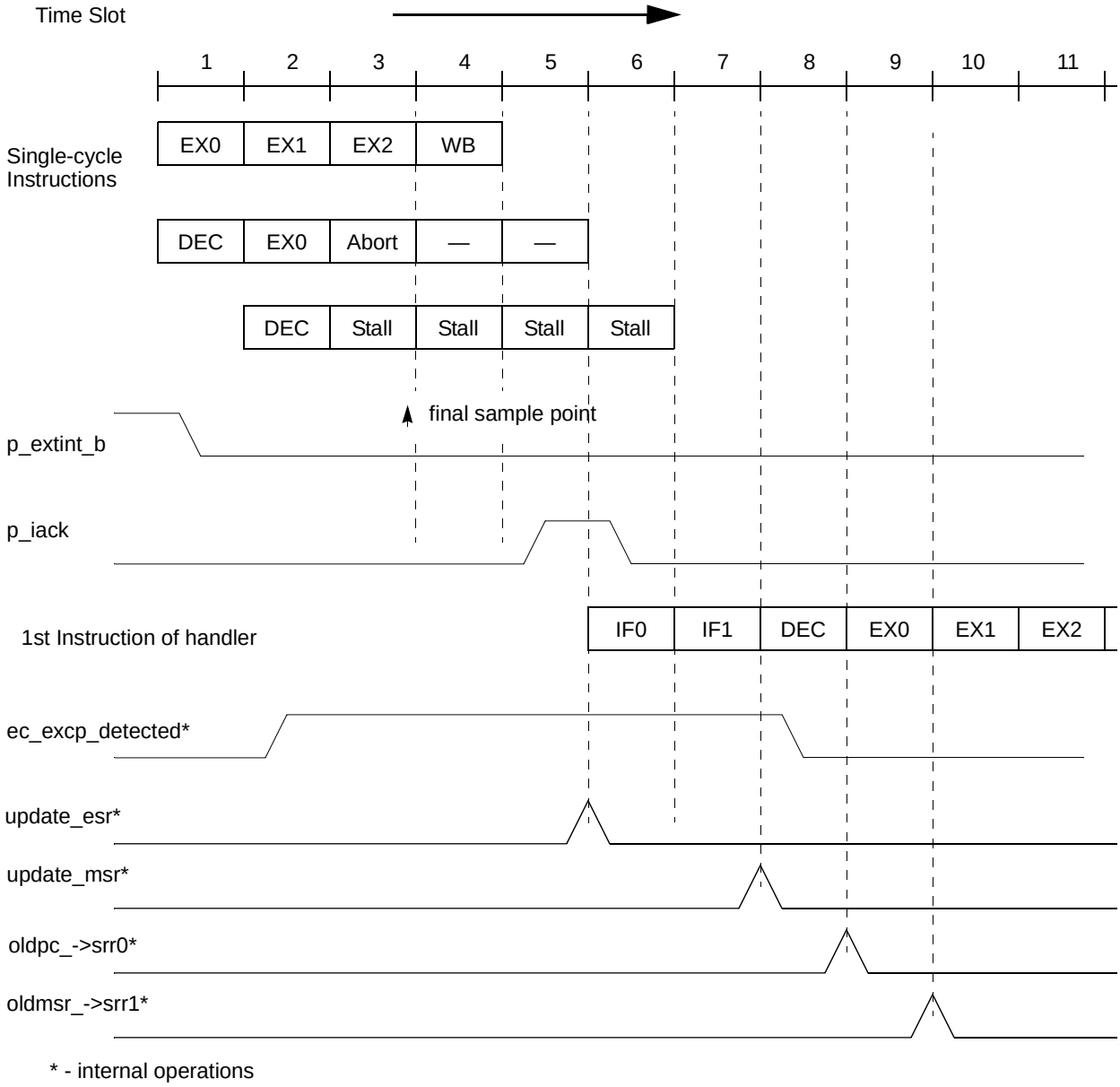


Figure 7-20. Interrupt Recognition and Exception Processing Timing

Figure 7-21 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 load or store instruction. The fetch for the handler is delayed until completion of the load or store, regardless of the number of wait states.

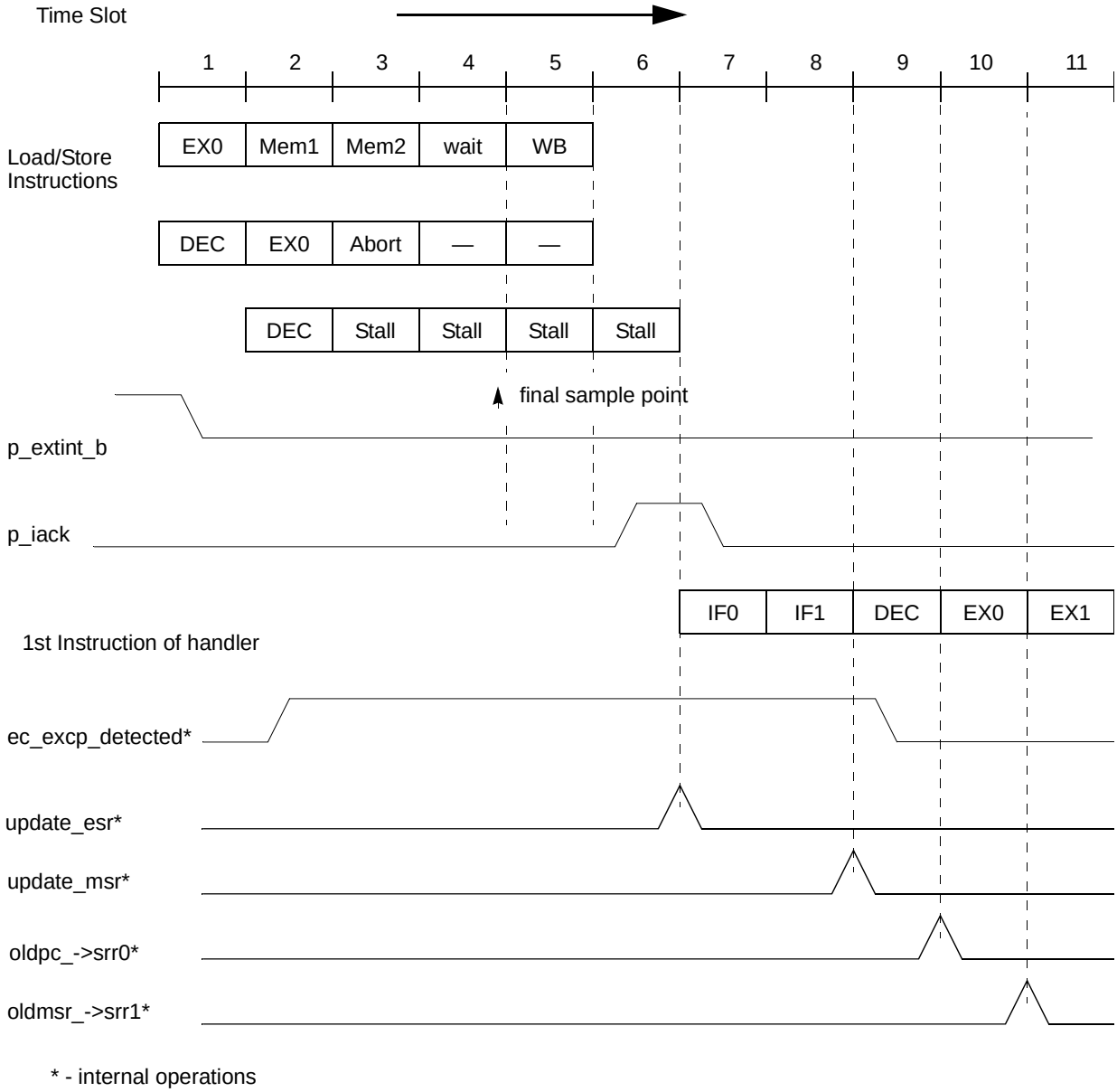


Figure 7-21. Interrupt Recognition and Handler Instruction Execution—Load/Store in Progress

Figure 7-22 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 multiple-cycle interruptible instructions. The handler is present in the cache and proceeds with no bubbles.

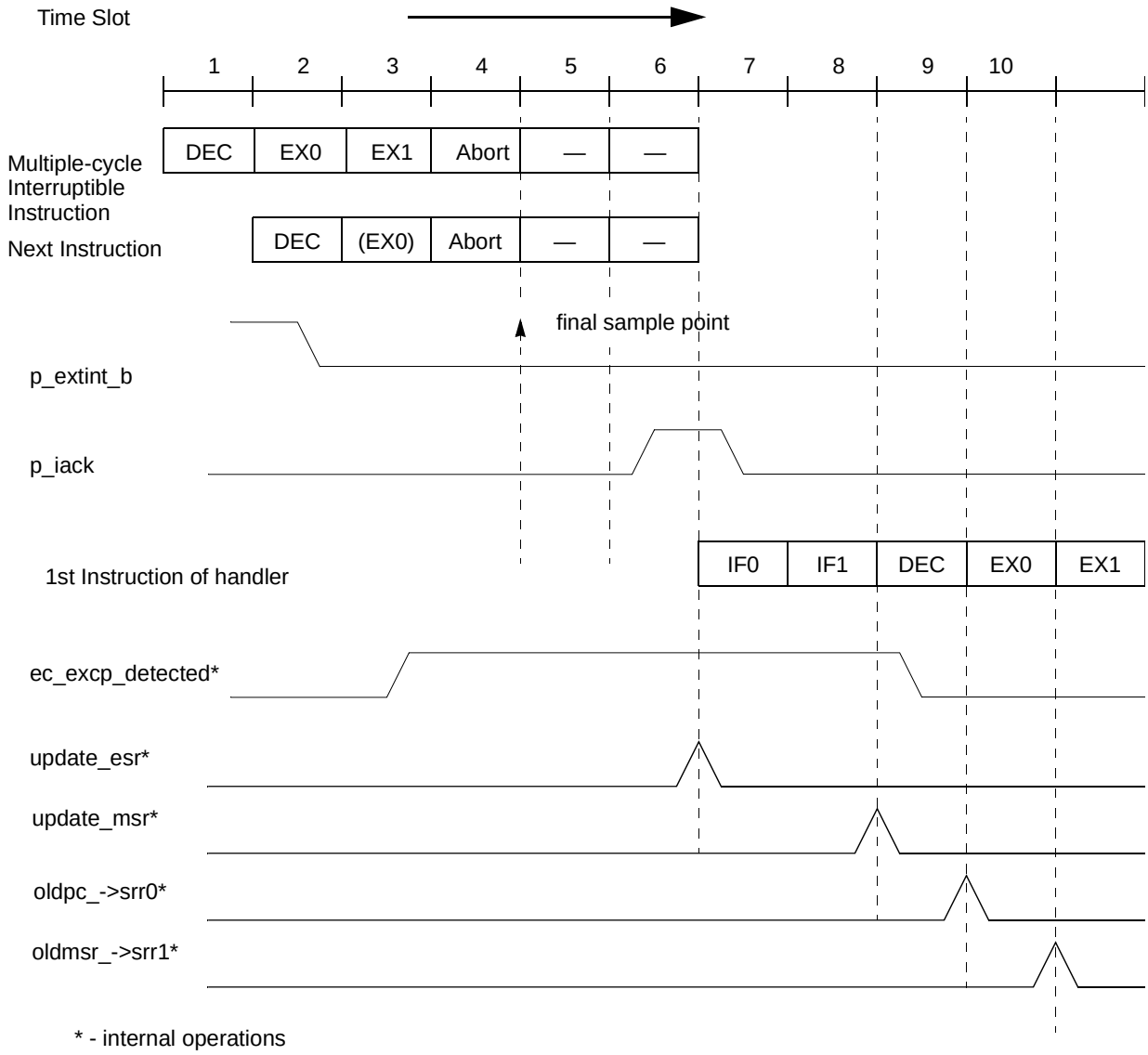


Figure 7-22. Interrupt Recognition and Handler Instruction Execution—Multiple-Cycle Instruction Abort

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.

7.7 Instruction Timings

Table 7-1 shows how timing is similar among related instructions; it does not show latencies for all supported instructions. 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. Section 7.7.1, “SPE and Embedded Floating-Point APU Instruction Timing,” describes timing for SPE instructions.

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. Cycle times marked with a ‘&’ require variable number of additional cycles due to serialization.

Table 7-1. Instruction Cycle Counts

Instruction	Latency	Throughput	Notes
Integer: add, sub, shift, rotate, logical, **cntlzw**	1	1	
Integer: compare	1	1	
Integer multiply	3	1	
Integer divide	6–16	6–16	Timing depends on operand size
Branch	3/1	3/1	Correct branch look ahead allows single-cycle execution
CR logical	1	1	
Loads (non-multiple)	3	1	
Load multiple	3 + n/2 (max)	1 + n/2 (max)	Actual timing depends on *n* and address alignment.
Stores (non-multiple)	3	1	
Store multiple	3 + n/2 (max)	1 + n/2 (max)	Actual timing depends on *n* and address alignment.
mtmsr, **wrtee**, **wrteei**	4&	4	
mcrf	1	1	
mfspr, **mtspr**	3&	3&	Applies to debug SPRs, optional unit SPRs
mfspr, **mfmsr**	1	1	Applies to internal, non debug SPRs
mfcr, **mtcr**	1	1	
rfi, **rfci**, **rfdi**	4	—	
sc	4	—	
tw, **twi**	4	—	Trap taken timing

7.7.1 SPE and Embedded Floating-Point APU Instruction Timing

This section shows latency and throughput for SPE and embedded floating-point instructions. Pipelined instructions are shown with cycles of total latency and throughput cycles. Divide instructions are not pipelined and block other instructions from executing during execution.

Instruction pipelining is affected by the possibility of a floating-point instruction generating an exception. A load or store class instruction that follows an embedded floating-point

instruction stalls until it can be ensured that no previous instruction can generate a floating-point exception. This is determined by which floating-point exception enable bits are set (FINVE, FOVFE, FUNFE, FDBZE, and FINXE) in the SPEFSCR and at what point in the FPU pipeline an exception can be guaranteed not to occur. Invalid input operands are detected in the first stage of the pipeline, while underflow, overflow, and inexactness are determined later in the pipeline. Best overall performance occurs when either floating-point exceptions are disabled, or when load and store class instructions are scheduled such that previous floating-point instructions have already resolved the possibility of exceptional results.

7.7.1.1 SPE Integer Simple Instruction Timing

Instruction timing for SPE integer simple instructions is shown in Table 7-2. The table is sorted by opcode. These instructions are dispatched as a pair of operations. Note that here, latency is defined as the time required for results to be made available to subsequent instructions. Instructions with single-cycle latency finish calculations in the first of the three execution pipeline stages, and although they make their results available to subsequent instructions, they must pass through remaining two execution pipeline stages.

Table 7-2. Timing for SPE Integer Simple Instructions

Instruction	Latency	Throughput	Comments
brinc	1	1	
evabs	1	1	
evaddiw	1	1	
evaddw	1	1	
evand	1	1	
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	

Table 7-2. Timing for SPE Integer Simple Instructions (continued)

Instruction	Latency	Throughput	Comments
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	
evsubifw	1	1	
evxor	1	1	

7.7.1.2 SPE Load and Store Instruction Timing

Instruction timing for SPE load and store instructions is shown in Table 7-2. The table is sorted by opcode. Actual timing depends on alignment; the table indicates timing for aligned operands.

Table 7-3. SPE Load and Store Instruction Timing

Instruction	Latency	Throughput	Comments
evldd	3	1	
evlddx	3	1	
evldh	3	1	
evldhx	3	1	
evldw	3	1	
evldwx	3	1	
evlhhesplat	3	1	
evlhhesplatx	3	1	
evlhhossplat	3	1	
evlhhossplatx	3	1	
evlhhousplat	3	1	
evlhhousplatx	3	1	
evlwhe	3	1	
evlwhex	3	1	
evlwhos	3	1	
evlwhosx	3	1	
evlwhou	3	1	
evlwhoux	3	1	
evlwhsplat	3	1	
evlwhsplatx	3	1	
evlwwsplat	3	1	
evlwwsplatx	3	1	
evstdd	3	1	
evstddx	3	1	
evstdh	3	1	
evstdhx	3	1	
evstdw	3	1	
evstdwx	3	1	

Table 7-3. SPE Load and Store Instruction Timing (continued)

Instruction	Latency	Throughput	Comments
evstwhe	3	1	
evstwhex	3	1	
evstwho	3	1	
evstwhox	3	1	
evstwwe	3	1	
evstwwex	3	1	
evstwwo	3	1	
evstwwox	3	1	

7.7.1.3 SPE Complex Integer Instruction Timing

Instruction timing for SPE complex integer instructions is shown in Table 7-4. The table is sorted by opcode. For the divide instructions, the number of stall cycles is (latency) for following instructions.

Table 7-4. SPE Complex Integer Instruction Timing

Instruction	Latency	Throughput	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	3	1	
evmhegsmfan	3	1	
evmhegsmiaa	3	1	
evmhegsmian	3	1	
evmhegumiaa	3	1	
evmhegumian	3	1	
evmhesmf	3	1	
evmhesmfa	3	1	
evmhesmfaaw	3	1	
evmhesmfanw	3	1	
evmhesmi	3	1	
evmhesmia	3	1	

Table 7-4. SPE Complex Integer Instruction Timing (continued)

Instruction	Latency	Throughput	Comments
evmhesmiaaw	3	1	
evmhesmianw	3	1	
evmhessf	3	1	
evmhessfa	3	1	
evmhessfaaw	3	1	
evmhessfanw	3	1	
evmhessiaaw	3	1	
evmhessianw	3	1	
evmheumi	3	1	
evmheumia	3	1	
evmheumiaaw	3	1	
evmheumianw	3	1	
evmheusiaaw	3	1	
evmheusianw	3	1	
evmhogsmfaa	3	1	
evmhogsmfan	3	1	
evmhogsmiaa	3	1	
evmhogsmian	3	1	
evmhogumiaa	3	1	
evmhogumian	3	1	
evmhosmf	3	1	
evmhosmfa	3	1	
evmhosmfaaw	3	1	
evmhosmfanw	3	1	
evmhosmi	3	1	
evmhosmia	3	1	
evmhosmiaaw	3	1	
evmhosmianw	3	1	
evmhossf	3	1	
evmhossfa	3	1	
evmhossfaaw	3	1	
evmhossfanw	3	1	
evmhossiaaw	3	1	

Table 7-4. SPE Complex Integer Instruction Timing (continued)

Instruction	Latency	Throughput	Comments
evmhossianw	3	1	
evmhoumi	3	1	
evmhoumia	3	1	
evmhoumiaaw	3	1	
evmhoumianw	3	1	
evmhousiaaw	3	1	
evmhousianw	3	1	
evmra	1	1	
evmwhsmf	3	1	
evmwhsmfa	3	1	
evmwhsmi	3	1	
evmwhsmia	3	1	
evmwhssf	3	1	
evmwhssfa	3	1	
evmwhumi	3	1	
evmwhumia	3	1	
evmwlsmf	3	1	
evmwlsmfa	3	1	
evmwlsmfaaw	3	1	
evmwlsmfanw	3	1	
evmwlsmiaaw	3	1	
evmwlsmianw	3	1	
evmwlssf	3	1	
evmwlssfa	3	1	
evmwlssfaaw	3	1	
evmwlssfanw	3	1	
evmwlssiaaw	3	1	
evmwlssianw	3	1	
evmwlumi	3	1	
evmwlumia	3	1	
evmwlumiaaw	3	1	
evmwlumianw	3	1	
evmwlusiaaw	3	1	

Table 7-4. SPE Complex Integer Instruction Timing (continued)

Instruction	Latency	Throughput	Comments
evmwlusianw	3	1	
evmwsmf	3	1	
evmwsmfa	3	1	
evmwsmfaa	3	1	
evmwsmfan	3	1	
evmwsmi	3	1	
evmwsmia	3	1	
evmwsmiaa	3	1	
evmwsmian	3	1	
evmwssf	3	1	
evmwssfa	3	1	
evmwssfaa	3	1	
evmwssfan	3	1	
evmwumi	3	1	
evmwumia	3	1	
evmwumiaa	3	1	
evmwumian	3	1	
evsubfsmiaaw	1	1	
evsubfssiaaw	1	1	
evsubfumiaaw	1	1	
evsubfusiaaw	1	1	

[1] Timing is data dependent

7.7.1.4 SPE Vector Floating-Point Instruction Timing

Instruction timing for SPE vector floating-point instructions is shown in Table 7-2. The table is sorted by opcode. The number of stall cycles for **evfsdiv** is (latency) cycles.

Table 7-5. SPE Vector Floating-Point Instruction Timing

Instruction	Latency	Throughput	Comments
evfsabs	3	1	
evfsadd	3	1	
evfscfsf	3	1	
evfscfsi	3	1	
evfscfuf	3	1	
evfscfui	3	1	
evfscmpeq	3	1	
evfscmpgt	3	1	
evfscmplt	3	1	
evfsctsf	3	1	
evfsctsi	3	1	
evfsctsiz	3	1	
evfsctuf	3	1	
evfsctui	3	1	
evfsctuiz	3	1	
evfsdiv	12	12	Blocking, no pipelining with next instruction
evfsmul	3	1	
evfsnabs	3	1	
evfsneg	3	1	
evfssub	3	1	
evfststeq	3	1	
evfststgt	3	1	
evfststlt	3	1	

7.7.1.5 Embedded Scalar Floating-Point Instruction Timing

Instruction timing for SPE scalar floating-point instructions is shown in Table 7-6. The table is sorted by opcode.

Table 7-6. Scalar SPE Floating-Point Instruction Timing

Instruction	Latency	Throughput	Comments
efsabs	3	1	
efsadd	3	1	
efscfsf	3	1	
efscfsi	3	1	
efscfuf	3	1	
efscfui	3	1	
efscmpeq	3	1	
efscmpgt	3	1	
efscmplt	3	1	
efsctsf	3	1	
efsctsi	3	1	
efsctsiz	3	1	
efsctuf	3	1	
efsctui	3	1	
efsctuiz	3	1	
efsdiv	12	12	Blocking, no execution overlap with next instruction
efsmul	3	1	
efsnabs	3	1	
efsneg	3	1	
efssub	3	1	
efststeq	3	1	
efststgt	3	1	
efststlt	3	1	

Detailed timing for each instruction mnemonic along with serialization requirements is shown in Table 7-7.

Table 7-7. 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	None
bc[l][a]	3	None
bcctr[l]	3	None
bclr[l]	3	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][.]	6–16 [1]	None
divwu[o][.]	6–16[1]	None
eqv[.]	1	None
extsb[.]	1	None

Table 7-7. Instruction Timing by Mnemonic (continued)

Mnemonic	Latency	Serialization
extsh[.]	1	None
isel	1	None
isync	3 [2]	Refetch
lbz	3 [3]	None
lbzu	3[3]	None
lbzux	3[3]	None
lbzx	3[3]	None
lha	3[3]	None
lhau	3[3]	None
lhaux	3[3]	None
lhax	3[3]	None
lhbrx	3[3]	None
lhz	3[3]	None
lhzu	3[3]	None
lhzux	3[3]	None
lhzx	3[3]	None
lmw	3 +(n/2)	None
lwarx	3	None
lwbrx	3[3]	None
lwz	3[3]	None
lwzu	3[3]	None
lwzux	3[3]	None
lwzx	3[3]	None
mbar	1[2]	Dispatch, completion
mcrf	1	None
mcrxr	1	Completion
mfcr	1	None
mfmsr	1	None
mfspr (debug)	3[2]	Completion
mfspr (except debug)	1	Completion
msync	1[2]	Dispatch, completion
mtcrf	2	None
mtmsr	4[2]	Completion

Table 7-7. Instruction Timing by Mnemonic (continued)

Mnemonic	Latency	Serialization
mtspr (debug)	3[2]	Completion
mtspr (except debug)	1	Completion
mulhw[.]	3	None
mulhwu[.]	3	None
mulli	3	None
mullw[o][.]	3	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	4	Dispatch, refetch, completion
rfdi	4	Dispatch, refetch, completion
rfi	4	Dispatch, refetch, completion
rlwimi[.]	1	None
rlwinm[.]	1	None
rlwnm[.]	1	None
sc	4	Dispatch, refetch
slw[.]	1	None
sraw[.]	1	None
srawi[.]	1	None
srw[.]	1	None
stb	3[3]	None
stbu	3[3]	None
stbux	3[3]	None
stbx	3[3]	None
sth	3[3]	None
sthbrx	3[3]	None
sthu	3[3]	None
sthux	3[3]	None

Table 7-7. Instruction Timing by Mnemonic (continued)

Mnemonic	Latency	Serialization
sthx	3[3]	None
stmw	3 + (n/2)	None
stw	3[3]	None
stwbrx	3[3]	None
stwcx.	3	None
stwu	3[3]	None
stwux	3[3]	None
stwx	3[3]	None
subf[o][.]	1	None
subfc[o][.]	1	None
subfe[o][.]	1	None
subfic	1	None
subfme[o][.]	1	None
subfze[o][.]	1	None
tw	4	None
twi	4	None
wrtee	4	Completion
wrteei	4	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

7.8 Effects of Operand Placement on Performance

The placement (location and alignment) of operands in memory affects relative performance of memory accesses, and in some cases, affects it significantly. Table 7-8 indicates the effects for the e200z6 core.

In Table 7-8, optimal means that one effective address (EA) calculation occurs during the memory operation. Good means that multiple EA calculations occur during a 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 7-8. Performance Effects of Operand Placement

Operand		Boundary Crossing*		
Size	**Byte Alignment**	**None**	**Cache Line**	**Protection Boundary**
4 byte	4	Optimal: One EA calculation	—	—
	<4	Good: Multiple EA calculations; may cause multiple bus transfers.		
2 byte	2	Optimal: One EA calculation	—	—
	<2	Good: Multiple EA calculations; may cause multiple bus transfers.		
1 byte	1	Optimal: One EA calculation	—	—
lmw, **stmw**	4	Good: Multiple EA calculations; may cause multiple bus transfers.		
	<4	Poor: Alignment interrupt occurs.		

Chapter 8
External Core Complex Interfaces

This chapter describes the external interfaces of the e200z6 core complex. Signal descriptions as well as data transfer protocols are documented in the following subsections.

Section 8.4, “Internal Signals,” describes a number of internal signals that are not directly accessible to users, but they are mentioned in various chapters in this manual and aid in understanding the behavior of the e200z6 core.

8.1 Overview

The external interfaces encompass the following:

- 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
- A test interface

The memory interface that the BIU supports is based on the AMBA AHB-Lite subset of the AMBA 2.0 AHB, with V6 AMBA extensions. (Ref. documents ARM IHI 0011A, ARM DVI 0044A, and ARM PR022-GENC-001011 0.4). Sideband signals, described in this chapter, support additional control functions. A 64-bit data bus is implemented. The pipelined memory interface supports read and write transfers of 8, 16, 24, 32, and 64 bits, misaligned transfers, burst transfers of four double words, and true big- and little-endian operating modes.

NOTE

The AMBA AHB bit and byte ordering reflect a natural little-endian ordering that AMBA documentation uses. The e200z6 BIU automatically performs byte lane conversions to support big-endian transfers. Memories and peripheral devices/interfaces should be wired according to byte lane addresses defined in Table 8-6.

Single-beat and misaligned transfers are supported for cache-inhibited read and write cycles, and write-buffer writes. Burst transfers (double-word–aligned) of 4 double words are supported for cache line-fill and copyback operations.

Misaligned accesses are supported with one or more transfers to the core interface. If an access is misaligned but is 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 that corresponds 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 e200z6 BIU performs a pair of transfers beginning at the effective address, requesting the original data size (either half word or word) for the first transfer, along with appropriate byte enables. For the second transfer, the address is incremented to the next 64-bit boundary, and the size and byte enable signals are driven to correspond to the number of remaining bytes to be transferred.

8.2 Signal Index

This section contains an index of the e200z6 signals.

The following prefixes are used for e200z6 signal mnemonics:

- *'m_'* denotes master clock and reset signals.
- *'p_'* denotes processor or core-related signals.
- *'j_'* denotes JTAG mode signals.
- *'jd_'* denotes JTAG and debug mode signals.
- *'ipt_'* denotes scan and test mode signals.
- *'nex_'* denotes Nexus3 signals.

NOTE

The "*_b*" suffix denotes an active low signal. Signals without the active-low suffix are active high.

Figure 8-1 groups core bus and control signals by function.

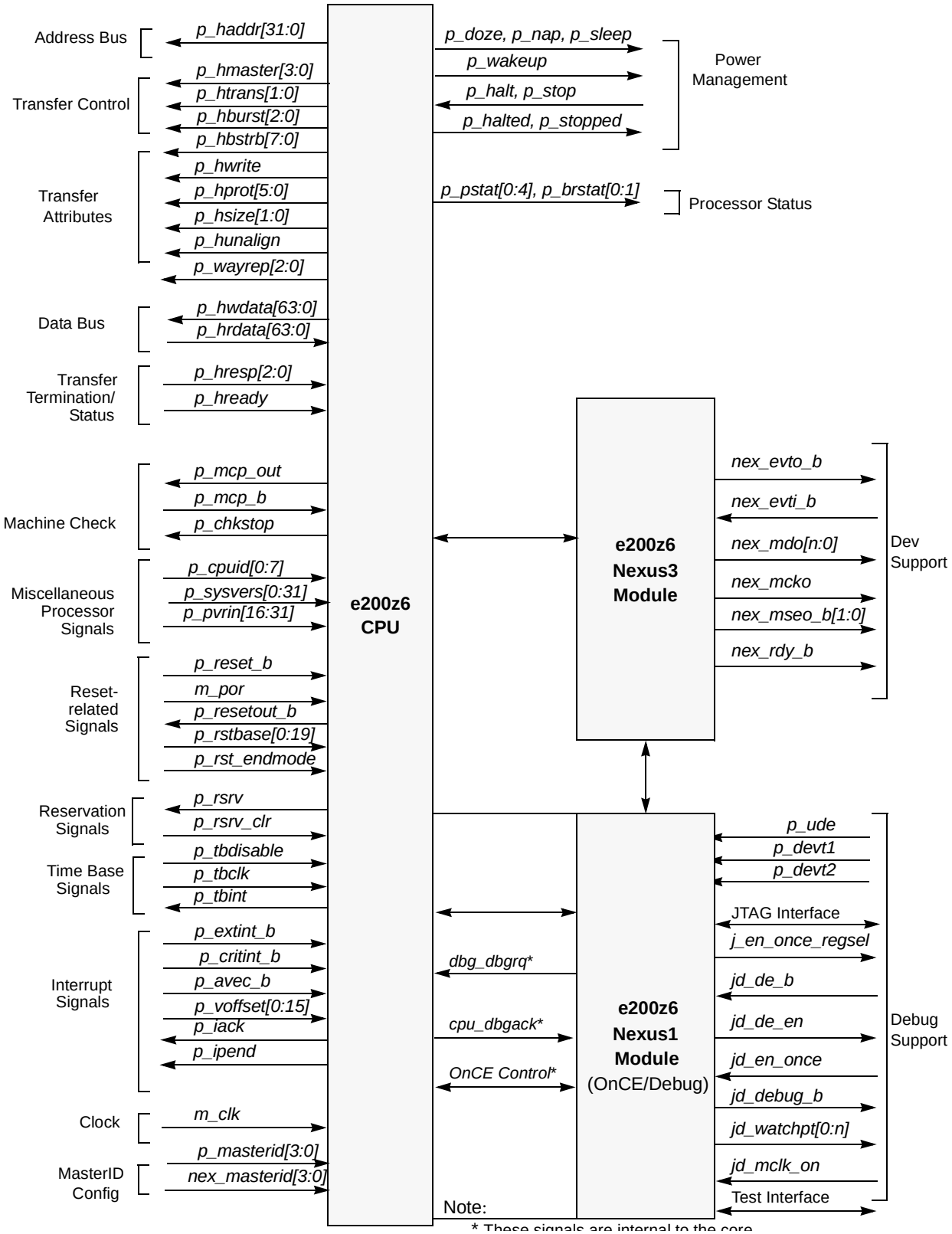


Figure 8-1. e200z6 Signal Groups

Table 8-1 shows e200z6 signal function and type, signal definition, and reset value. Signals are presented in functional groups.

Table 8-1. Interface Signal Definitions

Signal Name	I/O	Reset	Definition
Clock and Signals Related to Reset			
m_clk	I		Global system clock
m_por	I		Power-on reset
p_reset_b	I		Processor reset input
p_resetout_b	O		Processor reset output
p_rstbase[0:19]	I		Reset exception handler base address
p_rst_endmode	I		Reset endian mode select
Memory Interface Signals			
p_hmaster[3:0]	O	—	Master ID
p_haddr[31:0]	O	—	Address bus
p_hwrite	O	0	Write signal
p_hprot[5:0]	O	—	Protection codes
p_htrans[1:0]	O	—	Transfer type
p_hburst[2:0]	O	—	Burst type
p_hsize[1:0]	O	—	Transfer size
p_hunalign	O	—	Indicates that current data access is a misaligned access
p_hbstrb[7:0]	O	0	Byte strobes
p_hrdata[63:0]	I		Read data bus
p_hwdata[63:0]	O	—	Write data bus
p_hready	I		Transfer ready
p_hresp[2:0]	I		Transfer response
p_wayrep[2:0]	O		Way replacement. Indicates the cache way being replaced by a burst read line fill.
Master ID Configuration Signals			
p_masterid[3:0]	I	—	CPU master ID configuration
nex_masterid[3:0]	I	—	Nexus3 master ID configuration
Interrupt Interface Signals			
p_extint_b	I		External input interrupt request
p_critint_b	I		Critical input interrupt request
p_avec_b	I		Autovector request. Use internal interrupt vector offset.
p_voffset[0:15]	I		Interrupt vector offset for vectored interrupts

Table 8-1. Interface Signal Definitions (continued)

Signal Name	I/O	Reset	Definition
p_iack	O	0	Interrupt acknowledge. Indicates an interrupt is being acknowledged.
p_ipend	O	0	Interrupt pending. Indicates an interrupt is pending internally.
p_mcp_b	I		Machine check input request
Time Base Signals			
p_tbint	O	0	Time base interrupt
p_tbdisable	I	—	Time base disable input
p_tbclk	I	—	Time base clock input
Misc. CPU Signals			
p_cpuid[0:7]	I		CPU ID input
p_sysvers[0:31]	I		System version inputs (for SVR)
p_pvrin[16:31]	I		Inputs for PVR
p_pid0[0:7]	O	0	PID0[24:31] outputs
p_pid0_updt	O	0	PID0 update status
CPU Reservation Signals			
p_rsrv	O	0	Reservation status
p_rsrv_clr	I		Clear reservation flag
CPU State Signals			
p_pstat[0:4]	O	0	Processor status
p_brstat[0:1]	O	0	Branch prediction status
p_mcp_out	O	0	Machine check occurred
p_chkstop	O	0	Checkstop occurred
p_doze	O	0	Low-power doze mode of operation
p_nap	O	0	Low-power nap mode of operation
p_sleep	O	0	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
CPU Debug Event Signals			
p_ude	I		Unconditional debug event
p_devt1	I		Debug event 1 input

Table 8-1. Interface Signal Definitions (continued)

Signal Name	I/O	Reset	Definition
p_devt2	I		Debug event 2 input
Debug/Emulation Support Signals (Nexus 1/OnCE)			
jd_en_once	I		Enable full OnCE operation
jd_debug_b	O	1	Processor 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		System clock controller actively toggling *m_clk*
jd_watchpt[0:7]	O	0	Address watchpoint occurred
Development Support Signals (Nexus 3)			
nex_mcko	O		Nexus3 clock output
nex_rdy_b	O		Nexus3 ready output
nex_evto_b	O		Nexus3 event-out output
nex_evti_b	I		Nexus3 event-in input
nex_mdo[n:0]	O		Nexus3 message data output
nex_mseo_b[1:0]	O		Nexus3 message start/end output
JTAG-Related Signals			
j_trst_b	I		JTAG test reset from pad
j_tclk	I		JTAG test clock from pad
j_tms	I		JTAG test mode select from pad
j_tdi	I		JTAG test data input from pad
j_tdo	O	0	JTAG test data out to master controller or pad
j_tdo_en	O	0	Enables TDO output buffer
j_tst_log_rst	O	0	Test-logic-reset state of JTAG controller
j_capture_ir	O	0	Capture_IR state of JTAG controller
j_update_ir	O	0	Update_IR state of JTAG controller
j_shift_ir	O	0	Shift_IR state of JTAG controller
j_capture_dr	O	0	Parallel test data register load state of JTAG controller
j_shift_dr	O	0	TAP controller 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

Table 8-1. Interface Signal Definitions (continued)

Signal Name	I/O	Reset	Definition
j_nexus_regsel	O	0	External Nexus register select
j_lsrl_regsel	O	0	External LSRL register select
j_gp_regsel[0:11]	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

8.3 Signal Descriptions

Table 8-2 describes the e200z6 processor clock, *m_clk*.

Table 8-2. Processor Clock Signal Description

Signal	I/O	Signal Description
m_clk	I	e200z6 processor clock. The synchronous clock source for the e200z6 processor core. Because the e200z6 is designed for static operation, *m_clk* can be gated off to lower power dissipation (for example, during low-power stopped states).

Table 8-3 describes signals that are related to reset. The e200z6 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 are 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. It is also possible and desirable to assert *j_trst_b* and clear the JTAG/OnCE logic without affecting the processor state.

The synchronization logic between the processor and debug module requires an assertion of either *j_trst_b* or *m_por* during initial processor power-on reset to ensure proper operation. If the pin associated with *j_trst_b* is designed with a pull-up resistor and left floating, assertion of *m_por* is required during the initial power-on processor reset. Similarly, for those systems that 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. When a power-up reset is achieved, the two resets can be asserted independently.

A reset output signal, *p_resetout_b*, is also provided.

A set of input signals (*p_rstbase[0:19]*, *p_rst_endmode*) are provided to relocate the reset exception handler to allow for flexible placement of boot code and to select the default endian mode of the CPU out of reset.

Table 8-3. Descriptions of Signals Related to Reset

Signal	I/O	Signal Description	
m_por	I	Power-on reset. Serves the following purposes: • *m_por* is ORed with *j_trst_b* and the resulting signal clears the JTAG TAP controller and associated registers as well as the OnCE state machine. This signal is an asynchronous clear with a short assertion time requirement. • *m_por* is ORed with the *p_reset_b* function, and the resulting signal clears certain CPU registers. This is an asynchronous clear with a short assertion time requirement. Reset values for other registers are listed in Section 2.16.4, “Reset Settings.”	
		State Meaning	Asserted—Power-on reset is requested. Negated—Power-on reset is not requested.
p_reset_b	I	Reset. Treated as an asynchronous input and is sampled by the clock control logic in the e200z6 debug module.	
		State Meaning	Asserted—Reset is requested. Negated—Reset is not requested.
p_resetout_b	O	Reset out. Conditionally asserted by either the watchdog timer (Section 2.9.1, “Timer Control Register (TCR)”) or debug control logic. *p_resetout_b* is not asserted by *p_reset_b*.	
p_rstbase[0:19]	I	Reset base. Allows system integrators to specify or relocate the base address of the reset exception handler.	
		State Meaning	Forms the upper 20 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 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[0:19]]* || 0xFFC.
		Timing	Must remain stable in a window beginning 2 clocks before the negation of reset and extending into the cycle in which the reset vector fetch is initiated.
p_rst_endmode	I	Reset endian mode. Used by the MMU during reset to form the E bit of the default TLB entry 0 for translation of the reset vector fetch.	
		State Meaning	High—Causes the resultant entry E bit to be set, indicating a little-endian page. Low—causes the resultant entry E bit to be cleared, indicating a big-endian page.
j_trst_b	I	JTAG/OnCE reset (IEEE 1149.1 JTAG specification $\overline{\text{TRST}}$).	
		State Meaning	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 and the OnCE state machine.

Table 8-4 describes signals for the address and data buses. These outputs provide the address for a bus transfer. According to the AHB definition, *p_haddr31* is the msb and *p_haddr0* is the lsb.

Table 8-4. Descriptions of Signals for the Address and Data Buses

Signal	I/O	Signal Description
p_haddr[31:0]	O	Address bus. Provides the address for a bus transfer. According to the AHB definition, *p_haddr[31]* is the msb and *p_haddr[0]* is the lsb.
p_hrdata[63:0]	I	Read data bus. Provides data to the e200z6 on read transfers. The read data bus can transfer 8, 16, 24, 32, or 64 bits of data per transfer. According to the AHB definition, *p_hrdata63* is the msb and *p_hrdata0* is the lsb. Memory Byte Address — Wired to *p_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
p_hwdata[63:0]	O	Write data bus. Transfers data from the e200z6 on write transfers. The write data bus can transfer 8, 16, 24, 32, or 64 bits of data per bus transfer. According to the AHB definition, *p_hwdata[63]* is the msb and *p_hwdata[0]* is the lsb. Memory Byte Address — Wired to *p_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

Table 8-5 describes transfer attribute signals, which provide additional information about the bus transfer cycle. Attributes are driven with the address at the start of a transfer.

Table 8-5. Descriptions of Transfer Attribute Signals

Signal	I/O	Signal Description
p_htrans[1:0]	O	Transfer type. The processor drives these signals to indicate the current transfer type, as follows: *p_htrans1* — *p_htrans0* — Access type 0 — 0 — IDLE—No data transfer is required. Slaves must terminate IDLE transfers with a zero wait-state OKAY response and ignore the (non-existent) transfer. 0 — 1 — BUSY—(The e200z6 does not use the BUSY encoding and does not present this type of transfer to a bus slave.) Master is busy, burst transfer continues. 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 was incremented by the size of the data transferred (optionally wrapped). If the *p_htrans[1:0]* encoding is not IDLE or BUSY, a transfer is being requested.

Table 8-5. Descriptions of Transfer Attribute Signals (continued)

Signal	I/O	Signal Description
p_hwrite	O	Write. Defines the data transfer direction for the current bus cycle. **State Meaning**: Asserted—The current bus cycle is a write. Negated—The current bus cycle is a read.
p_hsize[1:0]	O	Transfer size. For misaligned transfers, size may exceed the requested size to ensure that all asserted byte strobes are within the container defined by *p_hsize[1:0]*. Table 8-7 and Table 8-8 show *p_hsize* encodings for aligned and misaligned transfers. *p_hsize[1:0]* — Transfer Size 00 — Byte 01 — Half word (2 bytes) 10 — Word (4 bytes) 11 — Double word (8 bytes)
p_hburst[2:0]]	O	Burst type. The e200z6 uses only SINGLE and WRAP4 burst types. *p_hburst[2:0]* — Burst Type 000 — SINGLE—No burst, single beat only 001 — INCR—Incrementing burst of unspecified length. Not used by the e200z6. 010 — WRAP4—4-beat wrapping burst 011 — INCR4—4-beat incrementing burst. Not used by the e200z6. 100 — WRAP8—8-beat wrapping burst. Not used by the e200z6. 101 — INCR8—8-beat incrementing burst. Not used by the e200z6. 110 — WRAP16 —16-beat wrapping burst. Not used by the e200z6. 111 — INCR16—16-beat incrementing burst. Not used by the e200z6.

Table 8-5. Descriptions of Transfer Attribute Signals (continued)

<table>
<tr><th>Signal</th><th>I/O</th><th>Signal Description</th></tr>
<tr><td>p_hprot[5:0]</td><td>O</td><td>Protection control. The e200z6 drives the p_hprot[5:0] signals to indicate the type of access for the current bus cycle. p_hprot[0] indicates instruction/data, p_hprot[1] indicates user/supervisor. p_hprot[5] indicates whether the access is exclusive (that is, for a lwarx or stwcx.). p_hprot[4:2] (allocate, cacheable, bufferable) indicate particular cache attributes for the access. The following table shows the definitions of the p_hprot[5:0] signals.
<table>
<tr><th>p_hprot5</th><th>p_hprot4</th><th>p_hprot3</th><th>p_hprot2</th><th>p_hprot1</th><th>p_hprot0</th><th>Transfer Type</th></tr>
<tr><td>—</td><td>—</td><td>—</td><td>—</td><td>—</td><td>0</td><td>Instruction access</td></tr>
<tr><td>—</td><td>—</td><td>—</td><td>—</td><td>—</td><td>1</td><td>Data access</td></tr>
<tr><td>—</td><td>—</td><td>—</td><td>—</td><td>0</td><td>—</td><td>User mode access</td></tr>
<tr><td>—</td><td>—</td><td>—</td><td>—</td><td>1</td><td>—</td><td>Supervisor mode access</td></tr>
<tr><td>—</td><td>0</td><td>0</td><td>0</td><td>—</td><td>—</td><td>Cache-inhibited</td></tr>
<tr><td>—</td><td>0</td><td>0</td><td>1</td><td>—</td><td>—</td><td>Guarded, not cache-inhibited</td></tr>
<tr><td>—</td><td>0</td><td>1</td><td>0</td><td>—</td><td>—</td><td>Reserved</td></tr>
<tr><td>—</td><td>0</td><td>1</td><td>1</td><td>—</td><td>—</td><td>Reserved</td></tr>
<tr><td>—</td><td>1</td><td>0</td><td>0</td><td>—</td><td>—</td><td>Reserved</td></tr>
<tr><td>—</td><td>1</td><td>0</td><td>1</td><td>—</td><td>—</td><td>Reserved</td></tr>
<tr><td>—</td><td>1</td><td>1</td><td>0</td><td>—</td><td>—</td><td>Cacheable, writethrough</td></tr>
<tr><td>—</td><td>1</td><td>1</td><td>1</td><td>—</td><td>—</td><td>Cacheable, writeback</td></tr>
<tr><td>0</td><td>—</td><td>—</td><td>—</td><td>—</td><td>—</td><td>Not exclusive</td></tr>
<tr><td>1</td><td>—</td><td>—</td><td>—</td><td>—</td><td>—</td><td>Exclusive access</td></tr>
</table>
The e200z6 maps Book E storage attributes to the AHB hprot signals as described in the following. For buffered stores, p_hprot[1] is driven with the user/supervisor mode attribute associated with the store at the time it was buffered. For cache line pushes/copybacks, p_hprot[1] indicates supervisor access. In both of these cases, p_hprot0 indicates a data access.
<table>
<tr><th>TLB[I]</th><th>TLB[G]</th><th>TLB[W]||!L1CSR0[CWM]</th><th>p_hprot[4:2]</th><th>Transfer Type</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>111</td><td>Cacheable, writeback</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>110</td><td>Cacheable, writethrough</td></tr>
<tr><td>0</td><td>1</td><td>—</td><td>001</td><td>Guarded, not cache-inhibited</td></tr>
<tr><td>1</td><td>—</td><td>—</td><td>000</td><td>Cache-inhibited</td></tr>
<tr><td>—</td><td>—</td><td>—</td><td>001</td><td>Buffered store, page marked guarded</td></tr>
<tr><td>—</td><td>—</td><td>—</td><td>110</td><td>Buffered store and page marked writethrough or L1CSR0[CWM]=0, and non-guarded</td></tr>
<tr><td>—</td><td>—</td><td>—</td><td>111</td><td>Buffered store and page marked copyback and L1CSR0[CWM]=1, and non-guarded</td></tr>
<tr><td>—</td><td>—</td><td>—</td><td>111</td><td>Dirty line push</td></tr>
</table></td></tr>
<tr><td>p_wayrep[2:0]</td><td>O</td><td>Cache way replacement.
<table>
<tr><td>State Meaning</td><td>Driven valid during cache line fills to indicate which way of the cache is being replaced. These signals are undefined on all other transfer types.</td></tr>
<tr><td>Timing</td><td>Driven valid with address and attribute timing; remain valid for all beats of the burst read.</td></tr>
</table></td></tr>
</table>

Table 8-6 describes signals for byte lane specification. Read transactions transfer from 1–8 bytes of data on the *p_hrdata[63:0]* bus. The lanes involved in the transfer are determined by the starting byte number specified by the lower address bits with the transfer size and byte strobes. Byte lane addressing is shown big-endian (left to right) regardless of the core's endian mode. The byte in memory corresponding to address 0 is connected to B0 (*p_h{r,w}data[7:0]*) and the byte corresponding to address 7 is connected to B7 (*p_h{r,w}data[63:56]*). The CPU internally permutes read data as required for the endian

mode of the current access. Assertion of *p_hunalign* indicates misaligned transfers and that byte strobes do not correspond exactly to size and low-order address bits.

Table 8-6. Descriptions of Signals for Byte Lane Specification

Signal	I/O	Signal Description	
p_hunalign	O	Unaligned access. Indicates whether the current access is misaligned.	
		State Meaning	Asserted—The current data access is misaligned. Indicates whether 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. Asserted only for misaligned data accesses; all instruction accesses are aligned. Negated—No misaligned data access is occurring.
		Timing	The timing of this signal is approximately the same as address timing.
p_hbstrb[7:0]	O	Byte strobes. Indicate the bytes selected for 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 correspond to the bytes that size and low-order address signals define. The relationships of byte addresses to the byte strobe signals are as follows.	

Memory byte address	Wired to p_h{r,w}data bits	Corresponding byte strobe signal
000	7:0	*p_hbstrb[0]*
001	15:8	*p_hbstrb[1]*
010	23:16	*p_hbstrb[2]*
011	31:24	*p_hbstrb[3]*
100	39:32	*p_hbstrb[4]*
101	47:40	*p_hbstrb[5]*
110	55:48	*p_hbstrb[6]*
111	63:56	*p_hbstrb[7]*

Table 8-7 lists all data transfer permutations. Note that misaligned data requests that cross a 64-bit boundary are broken into two bus transactions, and the address value and size encoding for the first transfer are 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 e200z6 performs the proper byte routing internally based on endianness.

Table 8-7. Byte Strobe Assertion for Transfers

Program Size and Byte Offset	A(2:0)	HSIZE [1:0]	Data Bus Byte Strobes [1]								HUNALIGN
			B0	B1	B2	B3	B4	B5	B6	B7	
Byte @000	0 0 0	0 0	X	—	—	—	—	—	—	—	0
Byte @001	0 0 1	0 0	—	X	—	—	—	—	—	—	0
Byte @010	0 1 0	0 0	—	—	X	—	—	—	—	—	0
Byte @011	0 1 1	0 0	—	—	—	X	—	—	—	—	0
Byte @100	1 0 0	0 0	—	—	—	—	X	—	—	—	0
Byte @101	1 0 1	0 0	—	—	—	—	—	X	—	—	0
Byte @110	1 1 0	0 0	—	—	—	—	—	—	X	—	0
Byte @111	1 1 1	0 0	—	—	—	—	—	—	—	X	0

Table 8-7. Byte Strobe Assertion for Transfers (continued)

Program Size and Byte Offset	A(2:0)	HSIZE [1:0]	Data Bus Byte Strobes [1]								HUNALIGN
			B0	B1	B2	B3	B4	B5	B6	B7	
Half @000	0 0 0	0 1	X	X	—	—	—	—	—	—	0
Half @001	0 0 1	1 0 [2]	—	X	X	—	—	—	—	—	1
Half @010	0 1 0	0 1	—	—	X	X	—	—	—	—	0
Half @011	0 1 1	1 1 [2]	—	—	—	X	X	—	—	—	1
Half @100	1 0 0	0 1	—	—	—	—	X	X	—	—	0
Half @101	1 0 1	1 0 [2]	—	—	—	—	—	X	X	—	1
Half @110	1 1 0	0 1	—	—	—	—	—	—	X	X	0
Half @111 (Two bus transfers)	1 1 1 0 0 0	0 1 [3] 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 [2]	—	X	X	X	X	—	—	—	1
Word @010	0 1 0	1 1 [2]	—	—	X	X	X	X	—	—	1
Word @011	0 1 1	1 1 [2]	—	—	—	X	X	X	X	—	1
Word @100	1 0 0	1 0	—	—	—	—	X	X	X	X	0
Word @101 (Two bus transfers)	1 0 1 0 0 0	1 0 0 0	— X	— —	— —	— —	— —	X —	X —	X —	1 0
Word @110 (Two bus transfers)	1 1 0 0 0 0	1 0 [3] 0 1	— X	— X	— —	— —	— —	— —	X —	X —	1 0
Word @111 (Two bus transfers)	1 1 1 0 0 0	1 0 [3] 1 0	— X	— X	— X	— —	— —	— —	— —	X —	1 1
Double word	0 0 0	1 1	X	X	X	X	X	X	X	X	0

[1] X indicates byte lanes involved in the transfer. Other lanes contain driven but unused data.

[2] These misaligned transfers drive size according to the size of the power of two aligned containers in which the byte strobes are asserted.

[3] These misaligned cases drive request size according to the size specified by the load or store instruction.

Table 8-8 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. The core breaks misaligned accesses that cross a double-word boundary into a pair of accesses. Double-word transfers are always double-word–aligned.

Table 8-8. Big-and Little-Endian Storage (64-bit GPR contains ‘A B C D E F G H’.)

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 [1]	—	G	H	—	—	—	—	—	—	—	—	—	—	—	—	—
B. E. Half @0010	0 0 1 0	0 1	—	—	G	H	—	—	—	—	—	—	—	—	—	—	—	—
B. E. Half @0011	0 0 1 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 [1]	—	—	—	—	—	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 [1]	—	—	—	—	—	—	—	—	—	G	H	—	—	—	—	—
B. E. Half @1010	1 0 1 0	0 1	—	—	—	—	—	—	—	—	—	—	G	H	—	—	—	—
B. E. Half @1011	1 0 1 1	1 1 [1]	—	—	—	—	—	—	—	—	—	—	—	G	H	—	—	—
B. E. Half @1100	1 1 0 0	0 1	—	—	—	—	—	—	—	—	—	—	—	—	G	H	—	—
B. E. Half @1101	1 1 0 1	1 0 [1]	—	—	—	—	—	—	—	—	—	—	—	—	—	G	H	—
B. E. Half @1110	1 1 1 0	0 1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	G	H

Table 8-8. Big-and Little-Endian Storage (64-bit GPR contains ‘A B C D E F G H’.)

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 @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 [1]	—	H	G	—	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Half @0010	0 0 1 0	0 1	—	—	H	G	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Half @0011	0 0 1 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 [1]	—	—	—	—	—	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 [1]	—	—	—	—	—	—	—	—	—	H	G	—	—	—	—	—
L. E. Half @1010	1 0 1 0	0 1	—	—	—	—	—	—	—	—	—	—	H	G	—	—	—	—
L. E. Half @1011	1 0 1 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 [1]	—	—	—	—	—	—	—	—	—	—	—	—	—	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 [1]	—	E	F	G	H	—	—	—	—	—	—	—	—	—	—	—
B. E. Word @0010	0 0 1 0	1 1 [1]	—	—	E	F	G	H	—	—	—	—	—	—	—	—	—	—
B. E. Word @0011	0 0 1 1	1 1 [1]	—	—	—	E	F	G	H	—	—	—	—	—	—	—	—	—
B. E. Word @0100	0 1 0 0	1 0	—	—	—	—	E	F	G	H	—	—	—	—	—	—	—	—
B. E. Word @0101	0 1 0 1	1 0	—	—	—	—	—	E	F	G	—	—	—	—	—	—	—	—
	1 0 0 0	0 0	—	—	—	—	—	—	—	—	H	—	—	—	—	—	—	—
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	—	—	—	—	—

Table 8-8. Big-and Little-Endian Storage (64-bit GPR contains ‘A B C D E F G H’.)

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 @1000	1 0 0 0	1 0	—	—	—	—	—	—	—	—	E	F	G	H	—	—	—	—
B. E. Word @1001	1 0 0 1	1 1[1]	—	—	—	—	—	—	—	—	—	E	F	G	H	—	—	—
B. E. Word @1010	1 0 1 0	1 1[1]	—	—	—	—	—	—	—	—	—	—	E	F	G	H	—	—
B. E. Word @1011	1 0 1 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[1]	—	H	G	F	E	—	—	—	—	—	—	—	—	—	—	—
L. E. Word @0010	0 0 1 0	1 1[1]	—	—	H	G	F	E	—	—	—	—	—	—	—	—	—	—
L. E. Word @0011	0 0 1 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	—	—	—	—	—
L. E. Word @1000	1 0 0 0	1 0	—	—	—	—	—	—	—	—	H	G	F	E	—	—	—	—
L. E. Word @1001	1 0 0 1	1 1[1]	—	—	—	—	—	—	—	—	—	H	G	F	E	—	—	—
L. E. Word @1010	1 0 1 0	1 1[1]	—	—	—	—	—	—	—	—	—	—	H	G	F	E	—	—
L. E. Word @1011	1 0 1 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	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—

Table 8-8. Big-and Little-Endian Storage (64-bit GPR contains ‘A B C D E F G H’.)

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 @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	- 0 0 0	1 1	A	B	C	D	E	F	G	H	—	—	—	—	—	—	—	—
L.E. Double word	- 0 0 0	1 1	H	G	F	E	D	C	B	A	—	—	—	—	—	—	—	—

[1] These misaligned transfers drive size according to the size of the power of two aligned containers in which the byte strobes are asserted.

Table 8-9 describes the transfer control signals.

Table 8-9. Descriptions of Signals for Transfer Control Signals

Signal	I/O	Signal Description	
p_hready	I	Transfer ready. Indicates whether a requested transfer operation has completed. An external device asserts *p_hready* to terminate the transfer. *p_hresp[2:0]* indicate the transfer status.	
		State Meaning	Asserted—A requested transfer operation has completed. An external device asserts *p_hready* to terminate the transfer. Negated—A requested transfer operation has not completed.
p_hresp[2:0]	I	Transfer response. Indicate status of a terminating transfer. *p_hresp[2:0]* Response Type 000 OKAY—Transfer terminated normally. 001 ERROR—Transfer terminated abnormally. See note for assertion. 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). See note for assertion. (Signaled to the CPU using the *p_xfail_b* internal signal. See Table 8-26.) 101–111 Reserved	
		Timing	Assertion—ERROR and XFAIL are required to be 2-cycle responses that must be signaled one cycle before assertion of *p_hready* and must remain unchanged during the cycle *p_hready* is asserted. The XFAIL response is signaled to the CPU using the *p_xfail_b* internal signal.

Table 8-10 describes the master ID configuration signals. These inputs drive the *p_hmaster[3:0]* outputs when a bus cycle is active.

Table 8-10. Descriptions of Master ID Configuration Signals

Signal	I/O	Signal Description
p_masterid[3:0]	I	CPU master. Configures the master ID for the CPU. Driven on *p_hmaster[3:0]* for a CPU-initiated bus cycle.
nex_masterid[3:0]	I	Nexus3 master. Configure the master ID for the Nexus3 unit. Driven on *p_hmaster[3:0]* for a Nexus3-initiated bus cycle.

Table 8-11 describes interrupt control signals. Interrupt request inputs (*p_extint_b*, *p_critint_b*, and *p_mcp_b*) to the core are level-sensitive. The interrupt controller must keep the interrupt request and any *p_voffset* or *p_avec_b* inputs (as appropriate) asserted until the interrupt is serviced to guarantee that the core recognizes the request. On the other hand, when a request is generated, the core may still not recognize the interrupt request, even if it is removed later. Requests must be held stable to avoid spurious responses.

Table 8-11. Descriptions of Interrupt Signals

Signal	I/O	Signal Description	
p_extint_b	I	External input interrupt request. Provides the external input interrupt request to the core. *p_extint_b* is masked by MSR[EE].	
		State Meaning	Asserted—An external input interrupt request has been signalled. Negated—An external input interrupt request has not been signalled.
		Timing	Not internally synchronized by the core. It must meet setup and hold time constraints relative to *m_clk* when the core clock is running. Assertion—Level-sensitive, must remain asserted to be guaranteed recognition.
p_critint_b	I	Critical input interrupt request. Critical input interrupt request to the core. Masked by MSR[CE].	
		State Meaning	Asserted—Critical input interrupt is being requested. Negated—No critical input interrupt is requested.
		Timing	Not internally synchronized by the e200z6 core. Must meet setup and hold times relative to *m_clk* when the core clock is running. See Section 8.5.3, “Interrupt Interface.” Assertion—Level-sensitive, must remain asserted to be guaranteed to be recognized.
p_ipend	I	Interrupt pending. Indicates whether a *p_extint_b* or *p_critint_b* interrupt request or an enabled timer facility interrupt was recognized internally by the core, is enabled by the appropriate bit in the MSR, and is asserted combinationally from the interrupt request inputs. *p_ipend* can signal other bus masters or a bus arbiter that an interrupt is pending. External power management logic can use *p_ipend* to control operation of the core and other logic or may use *p_wakeup* similarly. Higher priority exceptions may delay handling of the interrupt.	
		State Meaning	Asserted—A *p_extint_b* or *p_critint_b* interrupt request or an enabled timer facility interrupt (watchdog, fixed-Interval, or decrementer) was recognized internally by the core. Assertion of *p_ipend* does not mean that exception processing for the interrupt has begun. Negated—A *p_extint_b* or *p_critint_b* interrupt request or an enabled timer facility interrupt has not been recognized.

Table 8-11. Descriptions of Interrupt Signals (continued)

Signal	I/O	Signal Description	
p_avec_b	I	Autovector. Determines how a vector is chosen for critical and external interrupt signals.	
		State Meaning	Asserted—Asserted with either the *p_extint_b* or *p_critint_b* interrupt request to request use of the IVOR4 or IVOR0 for obtaining an exception vector offset. Negated—If negated when a *p_extint_b* or *p_critint_b* interrupt is requested, an external vector offset and context selector is taken from *p_voffset[0:15]*.
		Timing	Must be driven to a valid state during each clock cycle that either *p_extint_b* or *p_critint_b* is asserted. Assertion—Level-sensitive, must remain asserted to have guaranteed recognition.
p_voffset[0:15]	I	Interrupt vector offset. Vector offset and context selector used when processing begins for an incoming interrupt request. Ignored if multiple hardware contexts are not implemented.	
		State Meaning	Correspond to IVOR *n*[16–31]. *p_voffset[0:11]* are used in forming the exception handler address; *p_voffset[12:15]* are used to select a new operating context when multiple hardware contexts are implemented.
		Timing	Sampled with the *p_extint_b* and *p_critint_b* interrupt request inputs; must be driven to a valid value when either signal is asserted unless *p_avec_b* is also asserted. If *p_avec_b* is asserted, these inputs are not used. Assertion—Level-sensitive; must remain asserted to guarantee correct recognition. Must be asserted concurrently with *p_extint_b* and *p_critint_b* when used.
p_iack	O	Interrupt vector acknowledge. Interrupt vector acknowledge indicator to allow external interrupt controllers to be informed when a critical input or external input interrupt is being processed.	
		State Meaning	Asserted—An interrupt vector is being acknowledged. Negated—An interrupt vector is not being acknowledged.
		Timing	Assertion—Asserted after the cycle in which *p_avec_b* and *p_voffset[0:15]* are sampled in preparation for exception processing. See Figure 8-25 and Figure 8-26 for timing diagrams.
p_mcp_b	I	Machine check. Machine check interrupt request to the e200z6 core. Masked by HID0[EMCP].	
		State Meaning	Asserted—A machine check interrupt is being requested. Negated—A machine check interrupt is not being requested.
		Timing	Because this signal is not internally synchronized by the e200z6 core, it must meet setup and hold time constraints to *m_clk* when the e200z6 core clock is running. *p_mcp_b* is not sampled while the core is in the halted or stopped power management states. Assertion—*p_mcp_b* is sampled on two consecutive *m_clk* periods to detect a transition from the negated to the asserted state. It is internally qualified with this transition, but must remain asserted to be guaranteed to be recognized**.**

Table 8-12 describes the timer facility signals, which are associated with the time base, watchdog, fixed-interval, and decrementer facilities.

Table 8-12. Descriptions of Timer Facility Signals

Signal	I/O	Signal Description	
p_tbdisable	I	Timer disable. Used to disable the internal time base and decrementer counters. Used to freeze the state of the time base and decrementer during low power or debug operation.	
		State Meaning	Asserted—Time base and decrementer updates are frozen. Negated—Time base and decrementer updates are unaffected.
		Timing	Not internally synchronized by the e200z6 core; must meet setup and hold time constraints relative to *m_clk* when the e200z6 core clock is running, as well as to *p_tbclk* when selected as an alternate time base clock source.
p_tbclk	I	Timer external clock. Used as an alternate clock source for the time base and decrementer counters. Selection of this clock is made using HID0[SEL_TBCLK] (see Section 2.11.1, “Hardware Implementation-Dependent Register 0 (HID0)”).	
		Timing	Must be synchronous to the *m_clk* input and cannot exceed 50% of the *m_clk* frequency. Must be driven such that it changes state on the falling edge of *m_clk*.
p_tbint	O	Timer interrupt status. Indicates whether an internal timer facility unit is requesting an interrupt (TSR[WIS]=1 and TCR[WIE]=1, or TSR[DIS]=1 and TCR[DIE]=1, or TSR[FIS]=1 and TCR[FIE]=1). May be used to exit low power operation or for other system purposes.	
		State Meaning	Asserted—An internal timer facility unit is generating an interrupt request Negated—An internal timer facility unit is not generating an interrupt request

Table 8-13 describes the processor reservation signals associated with **lwarx** and **stwcx.**.

Table 8-13. Descriptions of Processor Reservation Signals

Signal	I/O	Signal Description	
p_rsrv	O	CPU reservation status. Indicates whether a reservation was established by the execution of a **lwarx**.	
		State Meaning	Asserted—A reservation was established by successful execution of a **lwarx**. Remains asserted until the reservation is cleared. Negated—No reservation is in effect.
		Timing	Assertion—Remains asserted until the reservation is cleared.
p_rsrv_clr	I	CPU reservation clear. Used to clear a reservation. External logic may use this signal to implement reservation management policies outside the scope of the CPU. *p_xfail_b* indicates success/failure of a **stwcx.** as part of bus transfer termination using the XFAIL *p_hresp[2:0]* encoding.	
		State Meaning	Asserted—Signals that a reservation should be cleared. Asserted independently of any bus transfer.
		Timing	Assertion—Asserted independently of any bus transfer.

Table 8-14 describes miscellaneous processor signals.

Table 8-14. Descriptions of Miscellaneous Processor Signals

Signal	I/O	Signal Description	
p_cpuid[0:7]	I	CPU ID. Reflected in the PIR. See Section 2.3.2, “Processor ID Register (PIR).”	
		Timing	Intended to remain in a static condition and are not internally synchronized.
p_pid0[0:7]	O	PID0 outputs. Reflected to PID0[56–63]. See Section 2.14.5, “Process ID Register (PID0).”	

Table 8-14. Descriptions of Miscellaneous Processor Signals (continued)

Signal	I/O	Signal Description	
p_pid0_updt	O	PID0 update. Indicates that PID0 is being updated by an **mtspr**.	
		State Meaning	Asserted—PID0 is being updated by an **mtspr**. Negated—PID0 is not being updated by an **mtspr**.
		Timing	Assertion—asserts during the clock cycle the *p_pid0[0:7]* outputs are changing**.**
p_sysvers[0:31]	I	System version. e200z6 core version number reflected in the SVR. See Section 2.3.4, “System Version Register (SVR).”	
		Timing	Intended to remain in a static condition and not internally synchronized.
p_pvrin[16:31]	I	Processor version. Provide a portion of the version number for a particular e200z6 CPU. Reflected in the processor version register. See Section 2.3.3, “Processor Version Register (PVR).”	
		Timing	Intended to remain in a static condition and are not internally synchronized.

8.3.1 Processor State Signals

Table 8-15 describes the processor state signals.

Table 8-15. Descriptions of Processor State Signals

Signal	I/O	Signal Description
p_pstat[0:4]	O	Processor status. Indicate the internal execution unit status. *p_pstat[0:4]* Internal Processor Status 00000 Execution stalled 00001 Execute exception 00010 Instruction squashed 00011–00111 Reserved 01000 Processor in halted state 01001 Processor in stopped state 01010 Processor in debug mode (As reflected on the *cpu_dbgack* internal state signal) 01011 Processor in checkstop state 01100–01111 Reserved 10000 Complete instruction (Except **rfi**, **rfci**, **rfdi**, **lmw**, **stmw**, **lwarx**, **stwcx.**, **isync**, **isel**, **evsel**, and change-of-flow Instructions) 10001 Complete **lmw**, or **stmw** 10010 Complete **isync** 10011 Complete **lwarx** or **stwcx.** 10100 Complete **evsel** with condition false for both elements 10101 Complete **evsel** with condition false for high element and true for low element 10110 Complete **evsel** with condition true for high element and false for low element 10111 Complete **evsel** with condition true for both elements 11000 Complete branch instruction **bc**, **bcl**, **bca**, **bcla**, **b**, **bl**, **ba**, **bla** resolved as not taken 11001 Complete branch instruction **bc**, **bcl**, **bca**, **bcla**, **b**, **bl**, **ba**, **bla** resolved as taken 11010 Complete **bclr**, **bclrl**, **bcctr**, **bcctrl** resolved as not taken 11011 Complete **bclr**, **bclrl**, **bcctr**, **bcctrl** resolved as taken 11100 Complete **isel** with condition false 11101 Complete **isel** with condition true 11110 Reserved 11111 Complete **rfi**, **rfci**, or **rfdi**
		Timing Synchronous with *m_clk*, so the indicated status may not apply to a current bus transfer.
p_brstat[0:1]	O	Branch prediction status. Indicates the status of a branch prediction prefetch. Such prefetches are performed for branch target buffer (BTB) hits with predict taken status to accelerate branches. *p_s1stat[0:1]* S1 prefetch status 0x Default (no branch predicted taken prefetch) 10 Branch predicted taken prefetch resolved as not taken 11 Branch predicted taken prefetch resolved as taken
		Timing Synchronous with *m_clk*, so the indicated status may not apply to a current bus transfer.
p_mcp_out	O	Processor machine check. Indicates whether a machine check condition has caused a syndrome bit to be set in the machine check syndrome register (MCSR).
		State Meaning Asserted—A machine check condition caused an MCSR bit to be set. Negated—No machine check condition exists that would set an MCSR bit.
p_chkstop	O	Processor checkstop. Asserted by the processor when a checkstop condition has occurred and the CPU has entered the checkstop state.
		State Meaning Asserted—The processor has indicated a checkstop condition. Negated—The processor has not indicated a checkstop condition.

Table 8-16 describes power management and other external control logic functions.

Table 8-16. Descriptions of Power Management Control Signals

Signal	I/O	Signal Description	
p_halt	I	Processor halt request. Used to request that the processor enter the halted state.	
		State Meaning	Asserted—Requests the processor to enter halted state. Negated—No request is being made for the processor to enter halted state.
p_halted	O	Processor halted. The active-high *p_halted* output signal indicates that the processor entered the halted state.	
		State Meaning	Asserted—The processor is in halted state. Negated—The processor is not in halted state.
p_stop	I	Processor stop request. The active-high *p_stop* input signal requests that the processor enter the stopped state.	
		State Meaning	Asserted—Requests the processor to enter stopped state. Negated—No request is being made for the processor to enter stopped state.
p_stopped	O	Processor stopped. The active-high *p_stopped* output signal indicates that the processor entered the stopped state.	
		State Meaning	Asserted—The processor is in stopped state. Negated—The processor is not in stopped state.
p_doze *p_nap* *p_sleep*	O	Low-power mode. Asserted by the processor to reflect the settings of HID0[DOZE,NAP,SLEEP] when MSR[WE] is set. The e200z6 core can be placed in a low-power state by forcing *m_clk* 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. 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 e200z6 core and peripherals in a low-power consumption state. *p_wakeup* can be monitored to determine when to end the low-power condition.	
		State Meaning	Asserted—MSR[WE] and the respective HID0 bit are both set. Negated—MSR[WE] and the respective HID0 bit are not both set.
		Timing	Assertion—May assert for 1 or more clock cycles.
p_wakeup	O	Wake up. Used by external logic to remove the e200z6 core and system logic from a low-power state. It can also indicate to the system clock controller that *m_clk* should be re-enabled for debug purposes. *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.	
		State Meaning	Asserted—Asserts whenever one of the following occurs: • A valid pending interrupt is detected by the core. • A request to enter debug mode is made by setting the OCR[DR] or via the assertion of *jd_de_b* or *p_ude*. • The processor is in a debug session and *jd_debug_b* is asserted. • A request to enable *m_clk* has been made by setting OCR[WKUP].
		Timing	See Section 8.5.2, “Power Management.” This signal is asynchronous to the system clock and should be synchronized to the system clock domain to avoid hazards.

Table 8-17 describes signal debug events to the e200z6 core.

Table 8-17. Descriptions of Debug Events Signals

Signal	I/O	Signal Description	
p_ude	I	Unconditional debug event. Used to request an unconditional debug event.	
		State Meaning	Asserted—An unconditional debug event has been requested. Only a transition from negated to asserted state of *p_ude* causes an event to occur. However, the level on this signal causes assertion of *p_wakeup*. Negated—No unconditional debug event has been requested.
		Timing	Not internally synchronized by the e200z6 core, and must meet setup and hold time constraints relative to *m_clk* when the core clock is running. Assertion—Level-sensitive and must be held asserted until acknowledged by software, or, when external debug mode is enabled, by assertion of *jd_debug_b* to be guaranteed recognition. Only a transition from negated to asserted state of *p_ude* causes an event to occur. However, the level on this signal causes assertion of *p_wakeup*.
p_devt1	I	External debug event 1. Used to request an external debug event. If the e200z6 core clock is disabled, this signal is not recognized. In addition, only a transition from negated to asserted state of *p_devt1* causes an event to occur. It is intended to signal e200z6-related events generated while the CPU is active.	
		State Meaning	Asserted—An external debug event is requested. Only a transition from negated to asserted state of *p_devt1* causes an event to occur. It is intended to signal e200z6-related events generated while the CPU is active. Negated—No external debug event is requested.
		Timing	Not internally synchronized by the e200z6 core, and must meet setup and hold time constraints relative to *m_clk* when the e200z6 core clock is running.
p_devt2	I	External debug event 2. Used to request an external debug event. If the e200z6 core clock is disabled, this signal is not recognized. In addition, only a transition from negated to asserted state of *p_devt2* causes an event to occur. It is intended to signal e200z6-related events generated while the CPU is active.	
		State Meaning	Asserted—An external debug event is requested. Only a transition from negated to asserted state of *p_devt2* causes an event to occur. Negated—No external debug event is requested.
		Timing	Not internally synchronized by the e200z6 core, and must meet setup and hold time constraints relative to *m_clk* when the e200z6 core clock is running.

Table 8-18 lists debug/emulation (Nexus 1/ OnCE) support signals. These signals assist in implementing an on-chip emulation capability with a controller external to the e200z6 core.

Table 8-18. e200z6 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

Table 8-19 describes debug/emulation (Nexus 1/ OnCE) support signals.

Table 8-19. Descriptions of Debug/Emulation (Nexus 1/ OnCE) Support Signals

Signal	I/O	Signal Description	
jd_en_once	I	OnCE enable. Enables the OnCE controller to allow certain instructions and operations to be executed. Other systems should tie this signal asserted to enable full OnCE operation. *j_en_once_regsel* and *j_key_in* are provided to assist external logic performing security checks.	
		State Meaning	Asserted—Enables the full OnCE command set, as well as operation of control signals and OnCE control register functions. Negated—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, OCR functions and the operation of *jd_de_b* are disabled. Secure systems may leave this signal negated until a security check is performed.
		Timing	Must change state only during the test-logic-reset, run-test/idle, or update_dr TAP states. A new value takes effect after one additional *j_tclk* cycle of synchronization.
jd_debug_b	O	Debug session. A debug session includes single-step operations (Go+NoExit OnCE commands). This signal is provided to inform system resources that access is occurring for debug purposes, thus allowing certain resource side effects to be frozen or otherwise controlled. Examples may include FIFO state change control and control of side-effects of register or memory accesses. See Section 10.5.4, “OnCE Interface Signals.”.	
		State Meaning	Asserted—Asserted when the processor enters debug mode. It remains asserted for the duration of a debug session. that is, during OnCE single-step executions.
jd_de_b	I	Debug request. Normally the input from the top-level DE_b open-drain bidirectional I/O cell. See Section 10.5.4, “OnCE Interface Signals.”	
		State Meaning	Asserted—A debug request is pending. Negated—No debug request is pending.
		Timing	Assertion—Not internally synchronized by the e200z6 core and 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 *jd_en_once* 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).
jd_de_en	O	DE_b active high output enable. Enable for the top-level DE_b open-drain bidirectional I/O cell. See Section 10.5.4, “OnCE Interface Signals.”	
		State Meaning	Asserted—the top-level DE_b open-drain bidirectional I/O cell is enabled. Negated—the top-level DE_b open-drain bidirectional I/O cell is disabled.
		Timing	Assertion—Asserted for three *j_tclk* periods upon processor entry into debug mode.
jd_mclk_on	I	Processor clock on. Driven by system-level clock control logic to indicate the *m_clk* input state	
		State Meaning	Asserted—The processor’s *m_clk* input is active. Negated—The processor’s *m_clk* input is not active.
		Timing	Assertion—Synchronized to *j_tclk* and provided as an OSR status bit.
jd_watchpoint [0:7]	O	Watchpoint events. Indicate whether a watchpoint occurred. Each debug address compare function (IAC1–IAC4, DAC1–DAC2), and debug counter event (DCNT1–DCNT2) is capable of triggering a watchpoint output.	
		State Meaning	Asserted—A watchpoint occurred Negated—No watchpoint occurred

Table 8-20 lists interface signals that assist in implementing a real-time development tool capability with a controller that is external to the e200z6 core. These signals are described in Section 11.11, “Nexus3 Pin Interface.”

Table 8-20. e200z6 Development Support (Nexus3) Signals

Signal	Type	Description
nex_mcko	O	Nexus3 clock output
nex_rdy_b	O	Nexus3 ready output
nex_evto_b	O	Nexus3 event-out output
nex_evti_b	I	Nexus3 event-in input
nex_mdo[n:0]	O	Nexus3 message data output
nex_mseo_b[1:0]	O	Nexus3 message start/end output

Table 8-21 lists the primary JTAG interface signals. These signals are usually connected directly to device pins (except for *j_tdo*, which needs tri-state and edge support logic), unless JTAG TAP controllers are concatenated.

Table 8-21. 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	O	Enables TDO output buffer. *j_tdo_en* is asserted when the TAP controller is in the shift_dr or shift_ir state.

Table 8-22 describes JTAG interface signals.

Table 8-22. Descriptions of JTAG Interface Signals

Signal	I/O	Signal Description
j_tdi	I	JTAG/OnCE serial input. Provides data and commands to the OnCE controller. Data is latched on the rising edge of *j_tclk*. Data is shifted into the OnCE serial port lsb first.
j_tclk	I	JTAG/OnCE serial clock. Supplies the serial clock to the OnCE control block. The serial clock provides pulses required to shift data and commands into and out of the OnCE serial port. (Data is clocked into the OnCE on the rising edge and is clocked out of the OnCE serial port on the rising edge.) The debug serial clock frequency must not exceed 50% of the processor clock frequency.

Table 8-22. Descriptions of JTAG Interface Signals (continued)

Signal	I/O	Signal Description
j_tdo	O	JTAG/OnCE serial output. Serial data is read from the OnCE block through *j_tdo*.
		State Meaning — Data is shifted out the OnCE serial port lsb first.
		Timing — When data is clocked out of the OnCE serial port, *j_tdo* changes on the rising edge of *j_tclk*. *j_tdo* is always driven. An external system-level TDO pin may be three-stateable and should be actively driven in the shift-IR and shift-DR controller states. *j_tdo_en* indicates when an external TDO pin should be enabled, and is asserted during the shift-IR and shift-DR controller states. In addition, for IEEE1149 compliance, the system-level pin should change state on the falling edge of TCLK.
j_tms	I	JTAG/OnCE test mode select. Used to cycle through states in the OnCE debug controller. Toggling *j_tms* while clocking with *j_tclk* controls transitions through the TAP state controller.
j_trst_b	I	JTAG/OnCE test reset. Resets the OnCE controller externally by placing it in the test-logic-reset state. The following information details additional signals that can support external JTAG data registers using the e200z6 TAP controller. Signal Name / Type / Description *j_tst_log_rst* O Indicates the TAP controller is in the Test-Logic-Reset state *j_rti* O JTAG controller run-test/idle state *j_capture_ir* O Indicates the TAP controller is in the capture IR state *j_shift_ir* O Indicates the TAP controller is in shift IR state *j_update_ir* O Indicates the TAP controller is in update IR state *j_capture_dr* O Indicates the TAP controller is in the capture DR state *j_shift_dr* O Indicates the TAP controller is in shift DR state *j_update_gp_reg* O Updates JTAG controller general-purpose data register *j_gp_regsel[0:11]* O General-purpose external JTAG register select *j_en_once_regsel* O External Enable OnCE register select *j_key_in* I Serial data from external key logic *j_nexus_regsel* O External Nexus register select *j_lsrl_regsel* O External LSRL register select *j_serial_data* I Serial data from external JTAG register(s)
j_tst_log_rst	O	Test-logic-reset. Indicates whether the TAP controller is in test-logic-reset state.
		State Meaning — Asserted—The TAP controller is in test-logic-reset state. Negated—The TAP controller is not in test-logic-reset state.
j_rti	O	Run-test/idle. Indicates whether the TAP controller is in the run-test/idle state.
		State Meaning — Asserted—The TAP controller is in run-test/idle state. Negated—The TAP controller is not in run-test/idle state.
j_capture_ir	O	Capture IR. Indicates whether the TAP controller is in the Capture_IR state.
		State Meaning — Asserted—The TAP controller is in Capture_IR state. Negated—The TAP controller is not in Capture_IR state.
j_shift_ir	O	Shift IR. Indicates whether the TAP controller is in the Shift_IR state.
		State Meaning — Asserted—The TAP controller is in Shift_IR state. Negated—The TAP controller is not in Shift_IR state.
j_update_ir	O	Update IR. Indicates the TAP controller is in the Update_IR state.
		State Meaning — Asserted—The TAP controller is in Update_IR state. Negated—The TAP controller is not in Update_IR state.

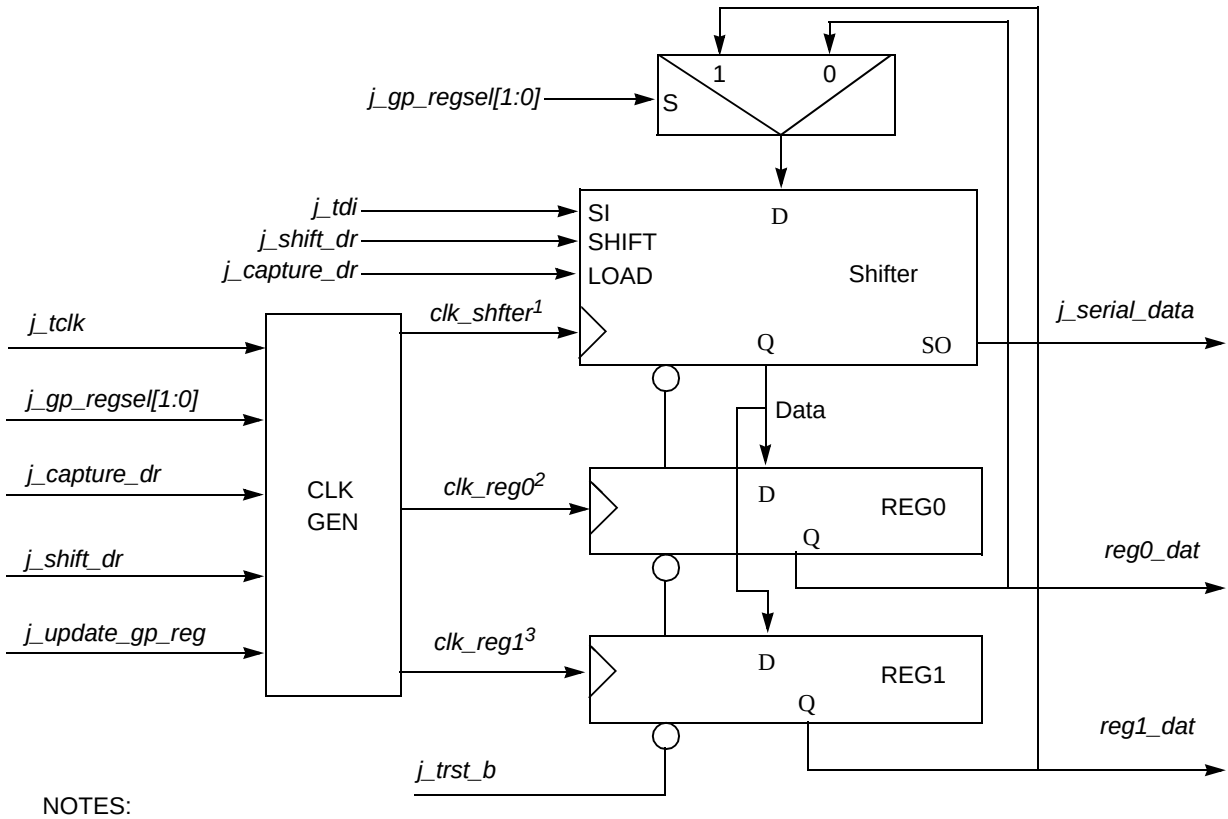
Table 8-22. Descriptions of JTAG Interface Signals (continued)

Signal	I/O	Signal Description
j_capture_dr	O	Capture DR. Indicates whether the TAP controller is in the Capture_DR state.
		State Meaning Asserted—The TAP controller is in Capture_DR state. Negated—The TAP controller is not in Capture_DR state.
j_shift_dr	O	Shift DR. Indicates whether the TAP controller is in the Shift_DR state.
		State Meaning Asserted—The TAP controller is in Shift_DR state. Negated—The TAP controller is not in Shift_DR state.
j_update_gp_reg	O	Update DR. Indicates whether the TAP controller is in the Update_DR state.
		State Meaning Asserted—The TAP controller is in the Update_DR state, and OCMD[R/W] is low (write command). *j_gp_regsel[0:11]* should be monitored to see which register, if any, needs updating. Negated—The TAP controller is not in the Update_DR state.
j_gp_regsel	O	Register select. Decoded from the OCMD[RS]. They are used to specify which external general-purpose JTAG register to access using the e200z6 TAP controller. Signal Name — Type — RS *j_gp_regsel[0]* O 0x70 *j_gp_regsel[1]* O 0x71 *j_gp_regsel[2]* O 0x72 *j_gp_regsel[3]* O 0x73 *j_gp_regsel[4]* O 0x74 *j_gp_regsel[5]* O 0x75 *j_gp_regsel[6]* O 0x76 *j_gp_regsel[7]* O 0x77 *j_gp_regsel[8]* O 0x78 *j_gp_regsel[9]* O 0x79 *j_gp_regsel[10]* O 0x7A *j_gp_regsel[11]* O 0x7B
j_en_once_regsel	O	Enable once register select. This control signal can be used by external security logic to help control *jd_enable_once*. The external enable_OnCE register should be muxed onto the *j_serial_data* input. During the Shift_DR state, *j_serial_data* is supplied to *j_tdo*.
		State Meaning Asserted—A decode of OCMD[RS] indicates an external enable_OnCE register is selected (0b1111110 encoding) for access using the e200z6 TAP controller.
j_nexus_regsel	O	External Nexus register select.
		State Meaning Asserted—A decode of OCMD[RS] indicates an external Nexus register is selected (0b1111100 encoding) for access using the e200z6 TAP controller. Negated—No Nexus register is selected.
j_lsrl_regsel	O	LSRL register select.
		State Meaning Asserted—A decode of OCMD[RS] indicates an external LSRL register is selected (0b1111101 encoding) for access using the e200z6 TAP controller.

Table 8-22. Descriptions of JTAG Interface Signals (continued)

Signal	I/O	Signal Description
j_serial_data	I	Serial data. Receives serial data from external JTAG registers. All external registers share this serial output back to the core. Therefore it must be muxed using *j_gp_regsel[0:11]*, *j_lsrl_regsel*, and *j_en_once_regsel*. The data is internally routed to *j_tdo*.
j_key_in	I	Key data 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. This input is provided to assist in implementing security logic outside of the e200z6 which conditionally asserts *jd_en_once*. During the Shift_IR state, when *jd_en_once* is negated, this input is sampled on the rising edge of *j_tclk*, and after a 2-clock delay the data is internally routed to *j_tdo*. This allows provision of a key value via the *j_tdo* output following a transition from Capture_IR to Shift_IR. The *j_key_in* input provides the key value.

Figure 8-2 shows one example for designing an external JTAG register set (2) using the inputs and outputs provided and by the JTAG primary inputs. 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 e200z6 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 8-2. Example External JTAG Register Design

8.3.2 JTAG ID Signals

Table 8-23 shows the JTAG ID register unique to Motorola as specified by the *IEEE 1149.1 JTAG Specification*. Note that bit 31 is the msb of this register.

Table 8-23. JTAG Register ID Fields

Bit Field	Type	Description	Value
[31:28]	Variable	Version number	Variable
[27:22]	Fixed	Design center number (e200z6)	01_1111
[21:12]	Variable	Sequence number	Variable
[11:1]	Fixed	Motorola manufacturer ID	000_0000_1110
0	Fixed	JTAG ID register identification bit	1

The e200z6 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, per the JTAG specification, and informs any JTAG controller that an ID register exists on the part. The e200z6 JTAG ID register is accessed by writing the OCMR (OnCE command register) with the value 0x02 in OCMD[RS].

The JTAG ID bit, manufacturer ID field, and design center number are fixed by the JTAG Consortium or Motorola. The version numbers and the 2 msbs of the sequence number are variable and brought out to external ports. The 8 lsbs of the sequence number are variable and are strapped internally to track variations in processor deliverables.

Table 8-24 shows the inputs to the JTAG ID register that are input ports on the e200z6 core. These bits can help a customer track revisions of a device using the e200z6 core.

Table 8-24. 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

Table 8-25 describes the JTAG ID signals.

Table 8-25. Descriptions of JTAG ID Signals

Signal	I/O	Signal Description
j_id_sequence[0:1]	I	JTAG ID sequence. Corresponds to the two msbs of the 10-bit sequence number in the JTAG ID register. These inputs are normally static and are provided for the integrator for further component variation identification.

Table 8-25. Descriptions of JTAG ID Signals (continued)

Signal	I/O	Signal Description
j_id_sequence[2:9]	I	JTAG ID sequence. Internally strapped by EPS to track variations in processor and module deliverables. Each e200z6 deliverable has a unique sequence number. Additionally, each revision of these modules can be identified by unique sequence numbers. EPS maintains a database of the sequence numbers.
j_id_version[0:3]	I	JTAG ID version. Corresponds to the 4-bit version number in the JTAG ID register. These inputs are normally static. They are provided to the customer for strapping to facilitate easy identification of component variants.

8.4 Internal Signals

Table 8-26 lists internal signals that are mentioned in this manual. These signals are not directly accessible to the user, but are used in this document to help describe the general behavior of the e200z6 processor.

Table 8-26. Internal Signal Descriptions

Signal Name	Description
p_addr[0:31]	Address bus. Provides the address for a bus transfer.
p_ta_b	Transfer acknowledge. Indicates completion of a requested data transfer operation. An external device asserts *p_ta_b* to terminate the transfer. For the e200z6 core to accept the transfer as successful, *p_tea_b* must remain high while *p_ta_b* is asserted.
p_tea_b	Transfer error acknowledge. Indicates that a transfer error condition has occurred and causes the e200z6 core to immediately terminate the transfer. An external device asserts *p_tea_b* to terminate the transfer with error. *p_tea_b* has higher priority than *p_ta_b*.
p_treq_b	Transfer request. The e200z6 core drives this output to indicate that a new access has been requested.
p_xfail_b	Store exclusive failure. An external agent causes assertion of *p_xfail_b* to indicate a failure of the store portion of a **stwcx.** for the current transfer. *p_xfail_b* is ignored if *p_tea_b* is asserted, because the store terminated with an error. Assertion of *p_xfail_b* with *p_ta_b* does not cause an exception; it indicates that the store was not performed due to a loss of reservation (determined by an external agent). The CPU updates the condition code accordingly and clears any outstanding reservation. *p_xfail_b* may be asserted by reservation logic or as a result of a system bus transfer with a failure response that is passed back to the CPU from the BIU. The AMBA XFAIL response is signaled back to the CPU using this signal. See Section 3.3, “Memory Synchronization and Reservation Instructions.” *p_xfail_b* is ignored for all transfers other than a **stwcx.**.

8.5 Timing Diagrams

The following sections discuss various types of timing diagrams.

8.5.1 Processor Instruction/Data Transfers

Transfer of data between the core and peripherals involves the address bus, data buses, and control and attribute signals. The address and data buses are parallel, non-multiplexed buses, supporting byte, half word, 3-byte, word, and double-word transfers. All bus inputs

and outputs are sampled and driven with respect to the rising edge of *m_clk*. 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 that may last for one or more cycles, depending on the state of *p_hready.*

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 1 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 to support sustained single cycle transfers. Up to two access requests may be in progress at any 1 cycle—one access outstanding and a second in the pending request phase.

Access requests are assumed to be accepted as long as no accesses are in progress, or if an access in progress is terminated during the same cycle a new request is active (*p_hready* asserted). When an access is accepted, the BIU is free to change the current request.

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 *p_hresp[2:0]* to signal that the current bus cycle has an error when a fault is detected, using the ERROR response encoding. ERROR assertion requires a 2-cycle response. In the first cycle of the response, *p_hresp[2:0]* 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 *p_htrans[2:0]* to the IDLE state in the second cycle of the 2-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.

8.5.1.1 Basic Read Transfer Cycles

During a read transfer, the core receives data from a memory or peripheral device. Figure 8-3 illustrates functional timing for basic read transfers, and clock-by-clock descriptions of activity follow.

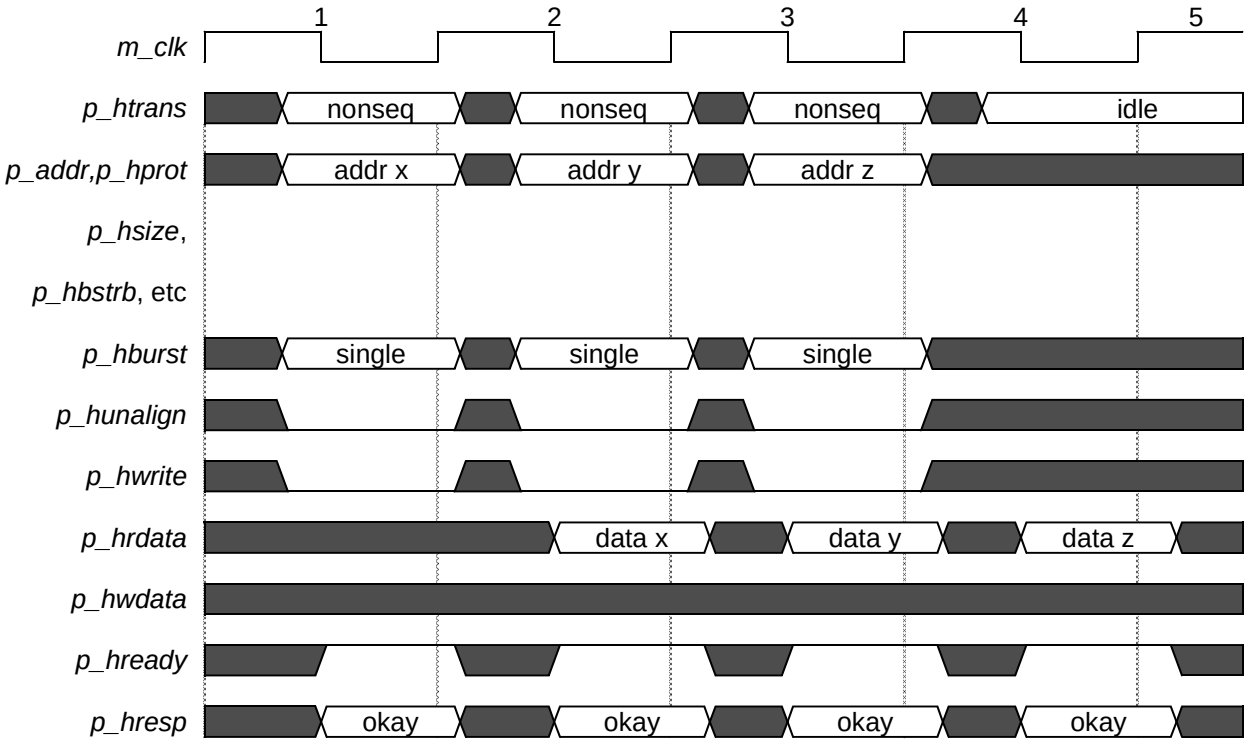


Figure 8-3. Basic Read Transfer—Single-cycle Reads, Full Pipelining

- Clock 1 (C1)—The first read transfer starts in clock cycle 1. During C1, the core places valid values on the address bus and transfer attributes. The burst type (*p_hburst[2:0]*), protection control (*p_hprot[5:0]*), and transfer type (*p_htrans[1:0]*) attributes identify the specific access type. The transfer size attributes (*p_hsize[1:0]*) indicate the size of the transfer. The byte strobes (*p_hbstrb[7:0]*) are driven to indicate active byte lanes. The write (*p_hwrite*) signal is driven low for a read cycle.

  The core asserts a transfer request (*p_htrans*= NONSEQ) during C1 to indicate that a transfer is being requested. Because the bus is currently idle, (0 transfers outstanding), the first read request to $addr_x$ is considered taken at the end of C1. The default slave drives an ready/OKAY response for the current idle cycle.
- Clock 2 (C2)—During C2, the $addr_x$ memory access takes place, using the address and attribute values that were driven during C1 to enable reading of 1 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 that the cycle is completing, and drives an OKAY response.

Another read transfer request is made during C2 to $addr_y$ (*p_htrans* = NONSEQ), and because the access to $addr_x$ is completing, it is considered taken at the end of C2.

- Clock 3 (C3)—During C3, the $addr_y$ memory access takes place, using the address and attribute values that were driven during C2 to enable reading 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 because the access to $addr_y$ is completing, it is considered taken at the end of C3.
- Clock 4 (C4)—During C4, the $addr_z$ memory access takes place, using the address and attribute values that were driven during C3 to enable reading 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.

  Because the CPU has no additional outstanding requests, *p_htrans* indicates IDLE and the address and attribute signals are undefined.

8.5.1.2 Read Transfer with Wait State

Figure 8-4 shows an example of wait state operation. Because signal *p_hready* for the first request ($addr_x$) is not asserted during C2, a wait state is inserted until *p_hready* is recognized (during C3).

Meanwhile, a subsequent request was generated by the CPU for $addr_y$ which is not taken in C2, because the previous transaction is still outstanding. The address and transfer attributes remain driven in cycle C3 and are taken at the end of C3 because the previous access is completing. Data for $addr_x$ and a ready/OKAY response is driven back by the slave device. In cycle C4, a request for $addr_z$ is made. The request for access to $addr_z$ is taken at the end of C4, and during C5, the slave device provides the data and a ready/OKAY response. In cycle C5, no further accesses are requested.

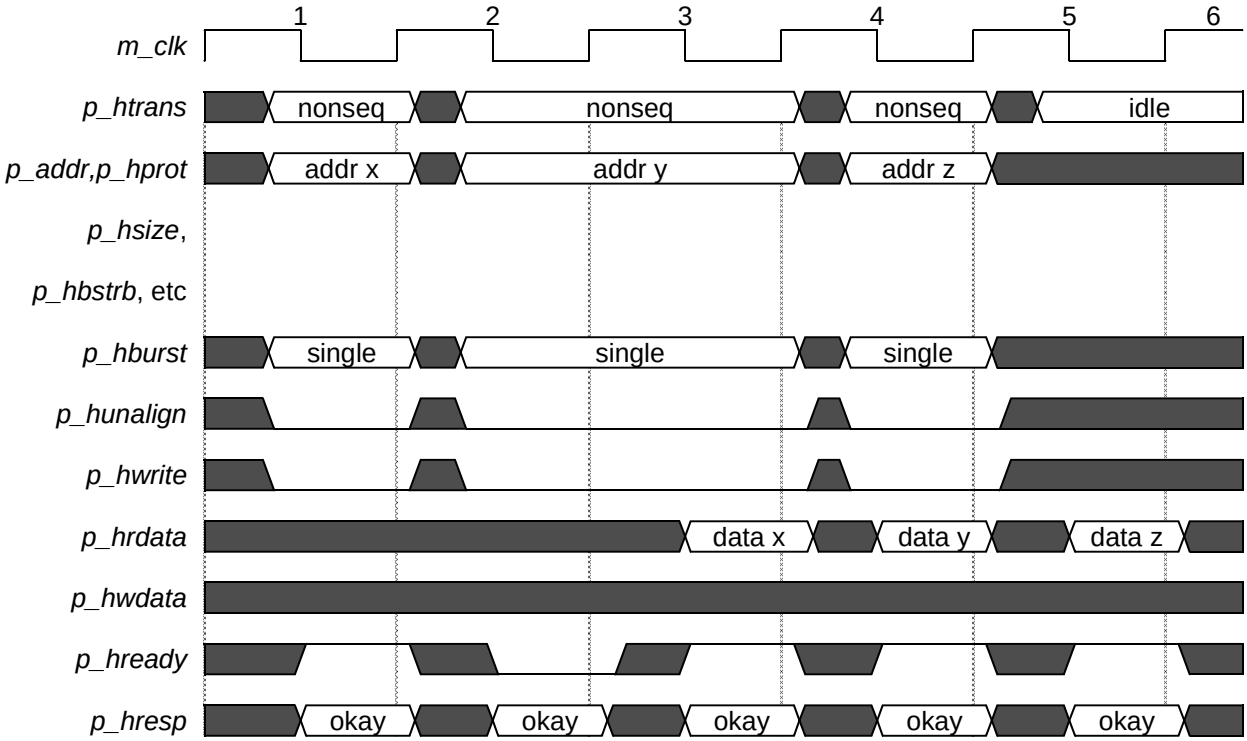


Figure 8-4. Read with Wait-State, Single-Cycle Reads, Full Pipelining

8.5.1.3 Basic Write Transfer Cycles

During a write transfer, the core provides write data to a memory or peripheral device. Figure 8-5 illustrates functional timing for basic write transfers. Clock-by-clock descriptions of activity in Figure 8-5 follow.

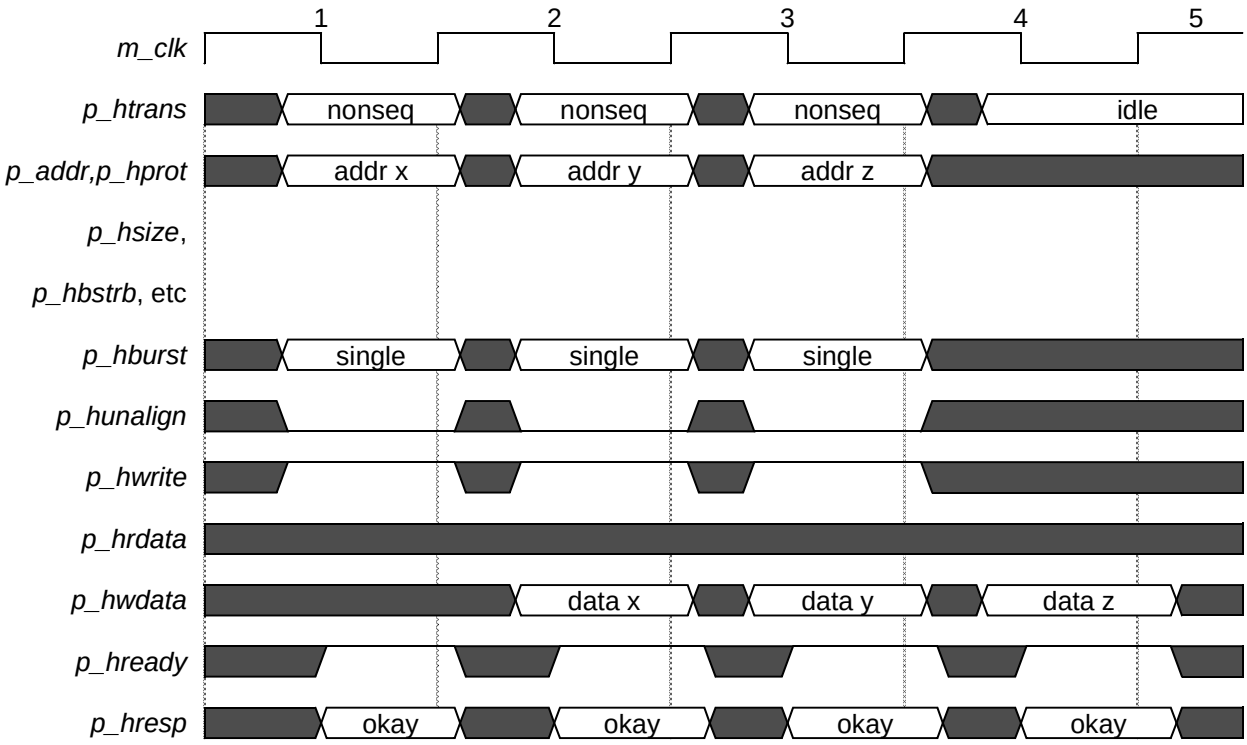


Figure 8-5. Basic Write Transfers—Single-Cycle Writes, Full Pipelining

- 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. Because the bus is idle, (0 transfers outstanding), the first read request to $addr_x$ is considered taken at the end of C1. The default slave drives an ready/OKAY response for the current idle cycle.

- Clock 2 (C2)—During C2, the write data for the access is driven and the $addr_x$ memory access occurs using the address and attribute values (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 because the access to $addr_x$ is completing, it is considered taken at the end of C2.

- Clock 3 (C3)—During C3, write data for $addr_y$ is driven, and the $addr_y$ memory access takes place using the address and attribute values (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 because the access to $addr_y$ is completing, it is considered taken at the end of C3.

- Clock 4 (C4)—During C4, write data for $addr_z$ is driven, and the $addr_z$ memory access takes place using the address and attribute values (driven during C3) to enable reading 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.

  Because the CPU has no more outstanding requests, *p_htrans* indicates IDLE and the address and attribute signals are undefined.

8.5.1.4 Write Transfer with Wait States

Figure 8-6 shows an example write wait state operation. Because *p_hready* for the first request ($addr_x$) is not asserted during C2, a wait state is inserted until *p_hready* is recognized (during C3).

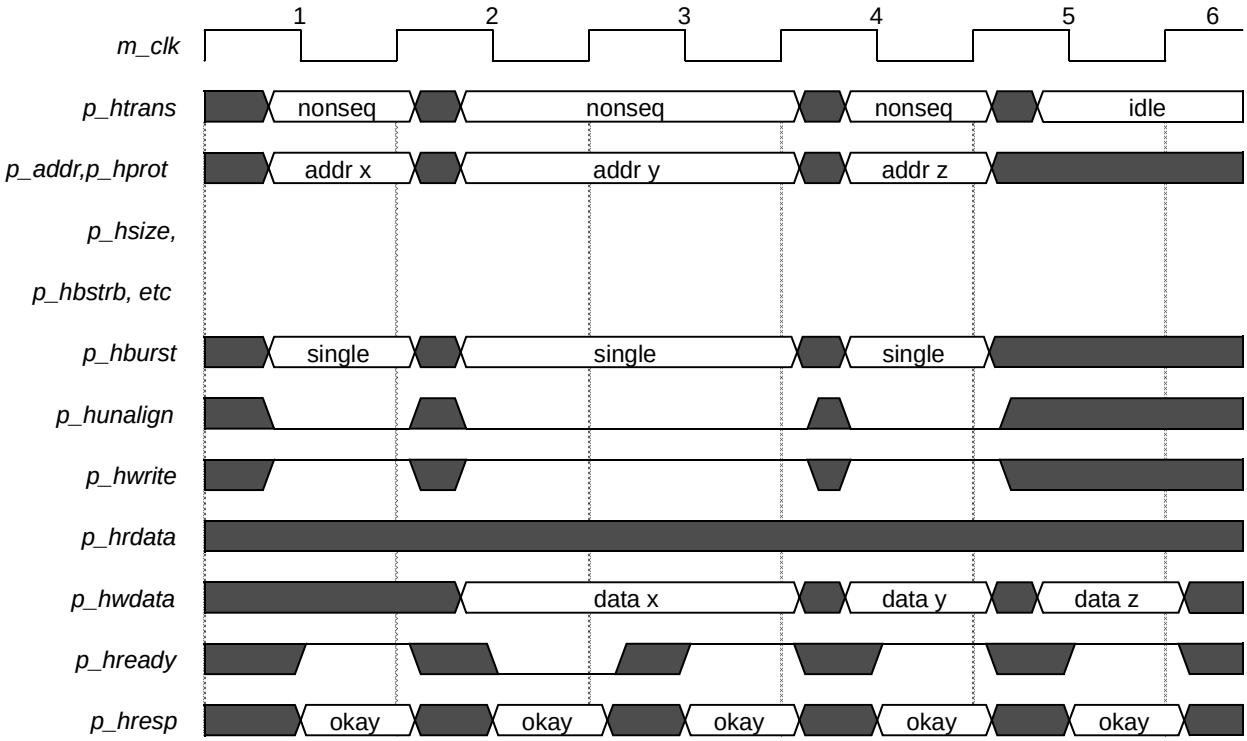


Figure 8-6. Write with Wait-state, Single-Cycle Writes, Full Pipelining

Meanwhile, the core generates a subsequent request for $addr_y$ which is not taken in C2, because the previous transaction is outstanding. The address, transfer attributes, and write data remain driven in cycle C3 and are taken at the end of C3 because 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, the slave device provides a ready/OKAY response. In C5, no further accesses are requested.

8.5.1.5 Read and Write Transfers

Figure 8-7 shows a sequence of read and write cycles.

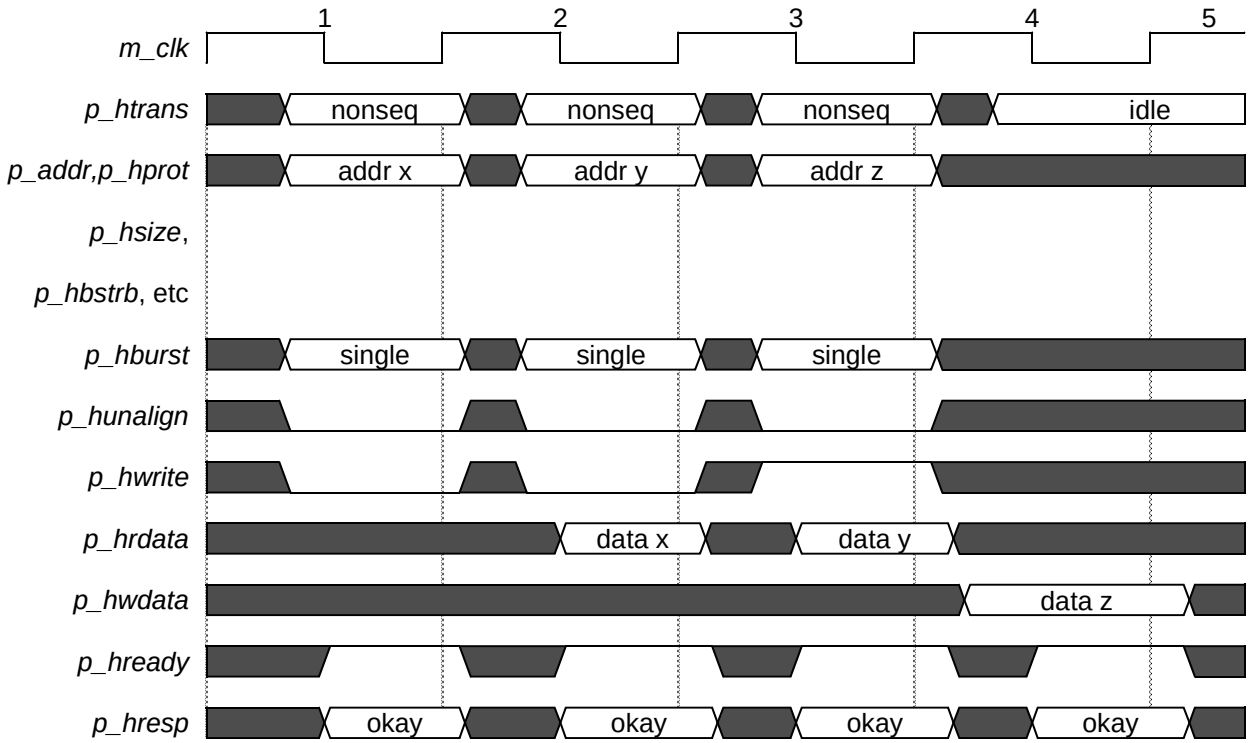


Figure 8-7. Single-Cycle Reads, Single-Cycle Write, Full Pipelining

The first read request ($addr_x$) is taken at the end of cycle C1 because the bus is idle. The second read request ($addr_y$) is taken at the end of C2 because a ready/OKAY response is asserted during C2 for the first read access ($addr_x$). During C3, a request is generated for a write to $addr_y$ which is taken at the end of C3 because the second access is terminating.

Data for the $addr_z$ write cycle is driven in C4, the cycle after the access is taken, and a ready/OKAY response is signaled to complete the write cycle to $addr_z$.

Figure 8-8 shows another sequence of read and write cycles. This example shows an interleaved write access between two reads.

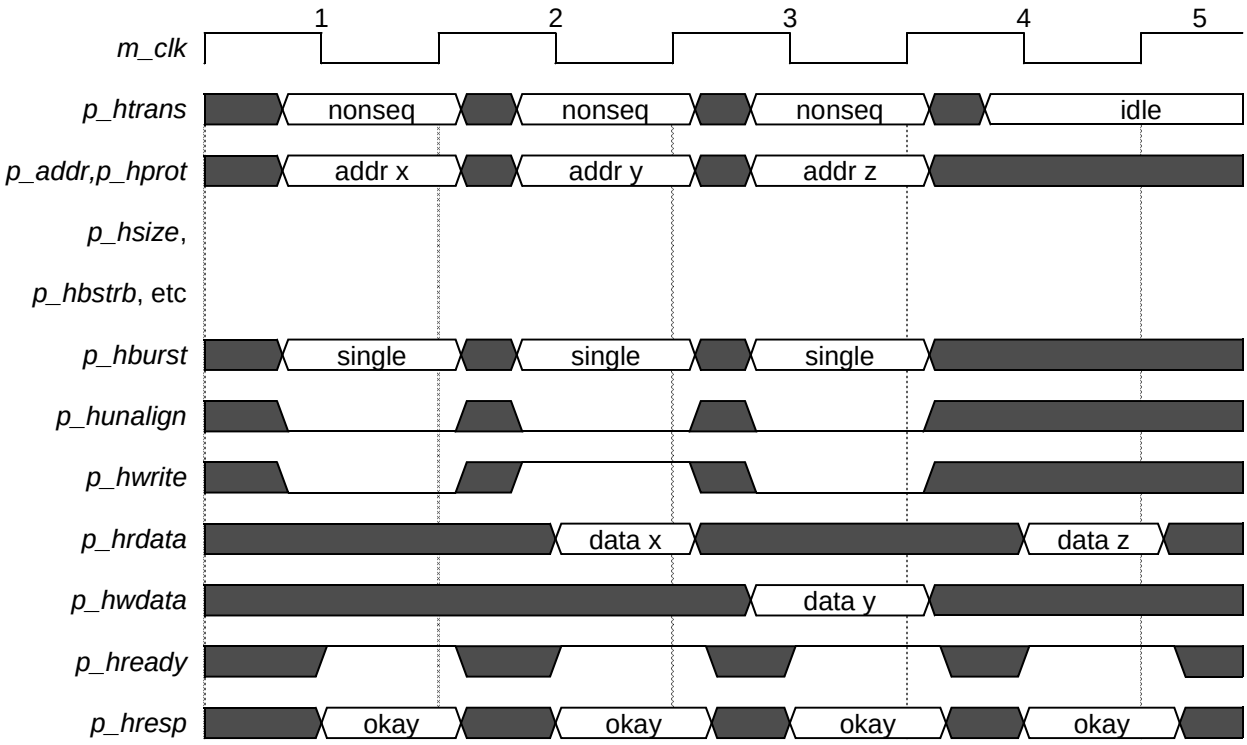


Figure 8-8. Single-Cycle Read, Write, Read—Full Pipelining

The first read request ($addr_x$) is taken at the end of cycle C1 because the bus is idle. The first write request ($addr_y$) is taken at the end of C2 because the first access is terminating ($addr_x$). Data for the $addr_y$ write cycle is driven in C3, the cycle after the access is taken. Also during C3, a request is generated for a read to $addr_z$, which is taken at the end of C3 because the write access is terminating.

During C4, the $addr_y$ write access is terminated, and no further access is requested.

Figure 8-9 shows another sequence of read and write cycles. In this example, reads incur a single wait state.

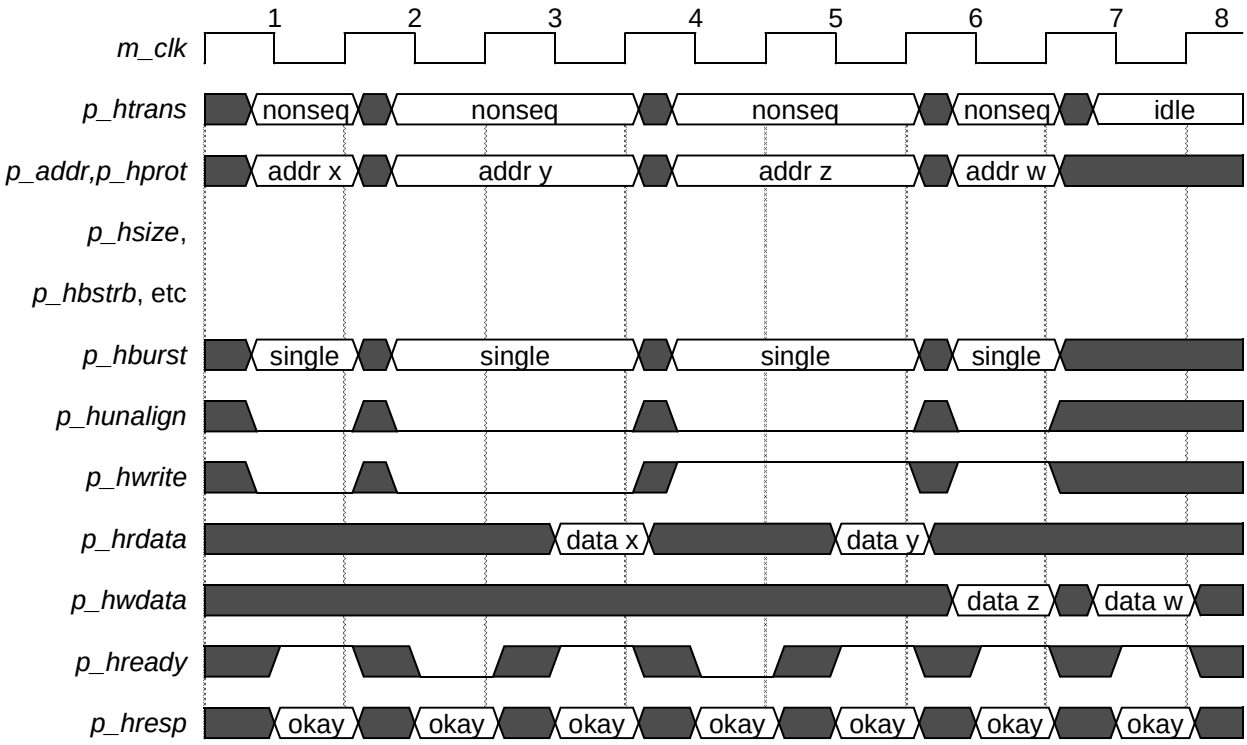


Figure 8-9. Multiple-Cycle Reads with Wait-State, Single-Cycle Writes, Full Pipelining

The first read request ($addr_x$) is taken at the end of cycle C1 because the bus is idle. The second read request ($addr_y$) is not taken at the end of cycle C2 because no ready response is signaled and only one access can be outstanding ($addr_x$). It is taken at the end of C3 once the first read request has signaled a ready/OKAY response.

The first write request ($addr_z$) is not taken during C4 because a ready response is not asserted during C4 for the second read access ($addr_y$). During C5, the request for a write to $addr_z$ is taken because the second access is terminating.

Data for the $addr_z$ write cycle is driven in C6, the cycle after the access is taken. During C6, the $addr_z$ write access is terminated and the $addr_w$ write request is taken.

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 8-10 shows another sequence of read and write cycles. In this example, reads incur a single wait state

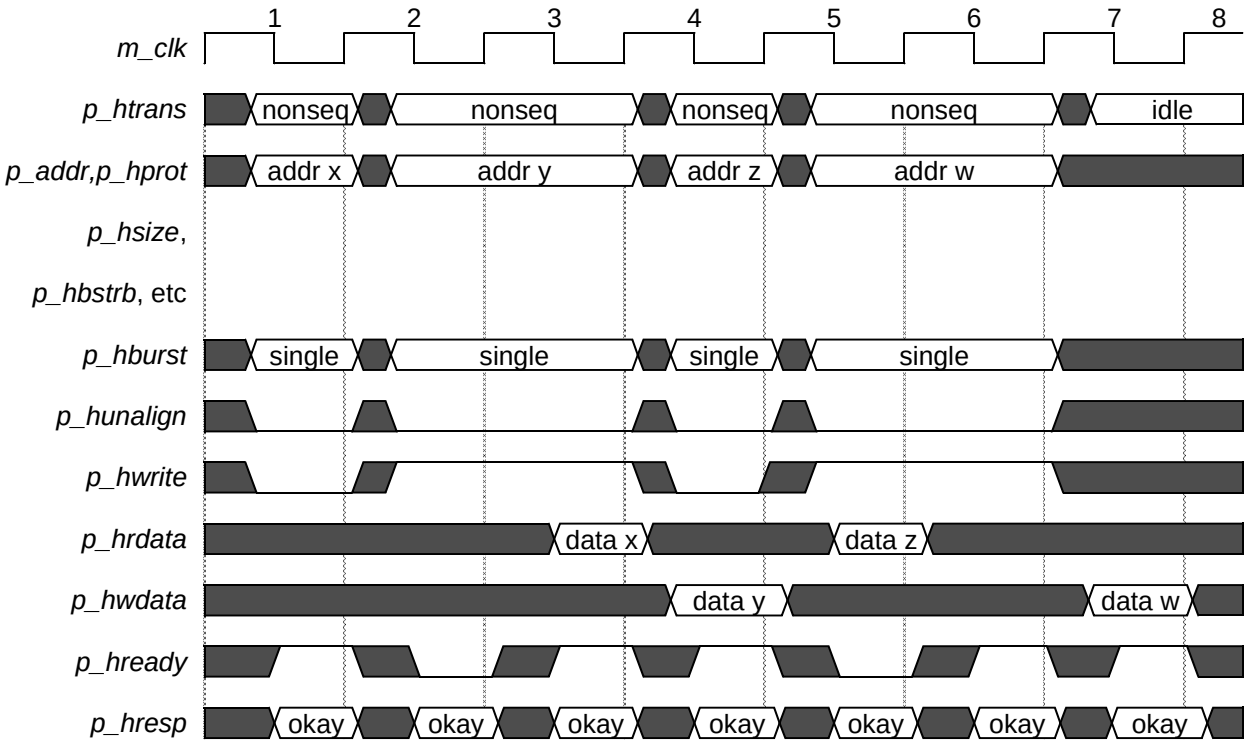


Figure 8-10. Multi-Cycle Read with Wait-State, Single-cycle write, Read with Wait-State, Single-Cycle Write, Full Pipelining

The first read request ($addr_x$) is taken at the end of cycle C1 because the bus is idle.

The first write request ($addr_y$) is not taken at the end of cycle C2 because no ready response is signaled and only one access can be outstanding ($addr_x$). It is taken at the end of C3 once the first read request has signaled a ready/OKAY response.

Data for the $addr_y$ write cycle is driven in C4, the cycle after the access is taken. The second read request ($addr_z$) is taken during C4 because the $addr_y$ write is terminating.

A second write request ($addr_w$) is not taken at the end of C5 because the second read access is not terminating, and it continues to drive the address and attributes into cycle C6. During C6, the $addr_z$ read access is terminated and the $addr_w$ write access is taken.

In cycle C7, data for the $addr_w$ write access is driven. During C7, a ready/OKAY response is asserted to complete the write cycle to $addr_w$. No further accesses are requested, so *p_htrans* signals IDLE.

8.5.1.6 Misaligned Accesses

Figure 8-11 illustrates functional timing for a misaligned read transfer. The read to $addr_x$ is misaligned across a 64-bit boundary. Note that only half word and word transfers may be misaligned; double-word transfers are always aligned.

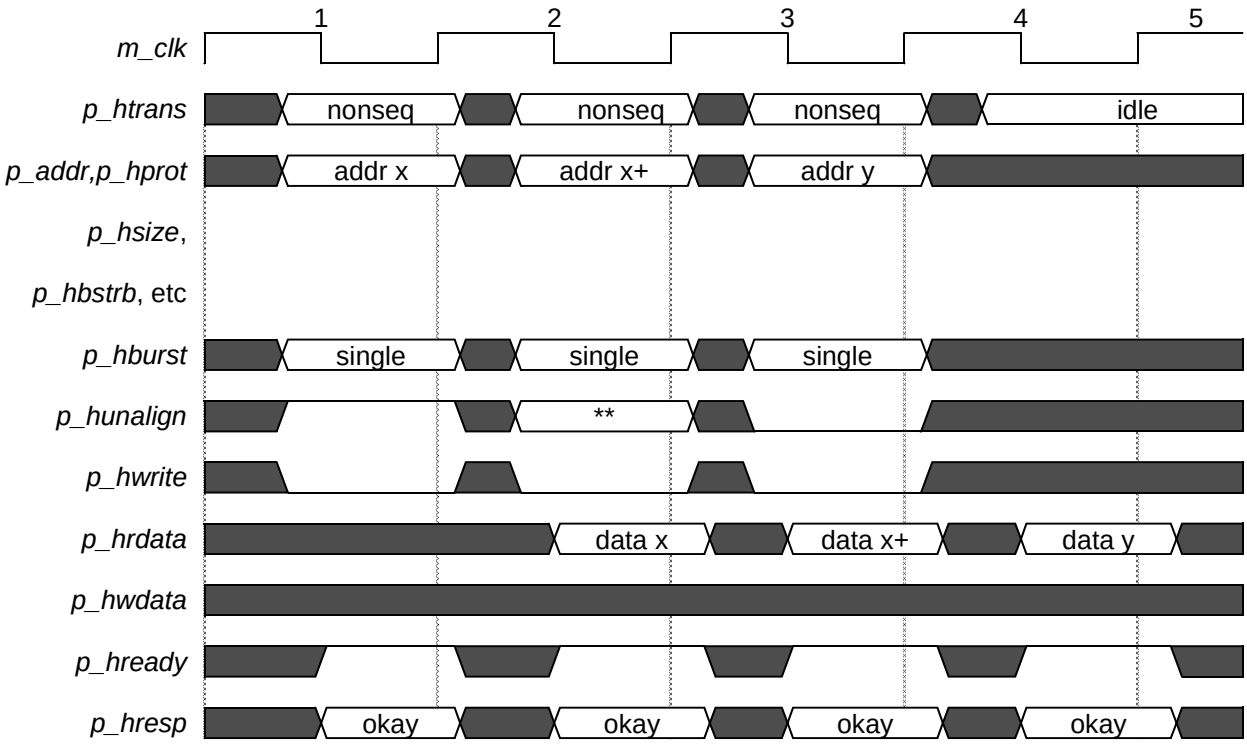


Figure 8-11. Misaligned Read, Read, Full Pipelining

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, *p_hunalign* 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, because this access is aligned.

Figure 8-12 illustrates functional timing for a misaligned write transfer. The write to $addr_x$ is misaligned across a 64-bit boundary. Note that only half word and word transfers may be misaligned; double-word transfers are always aligned.

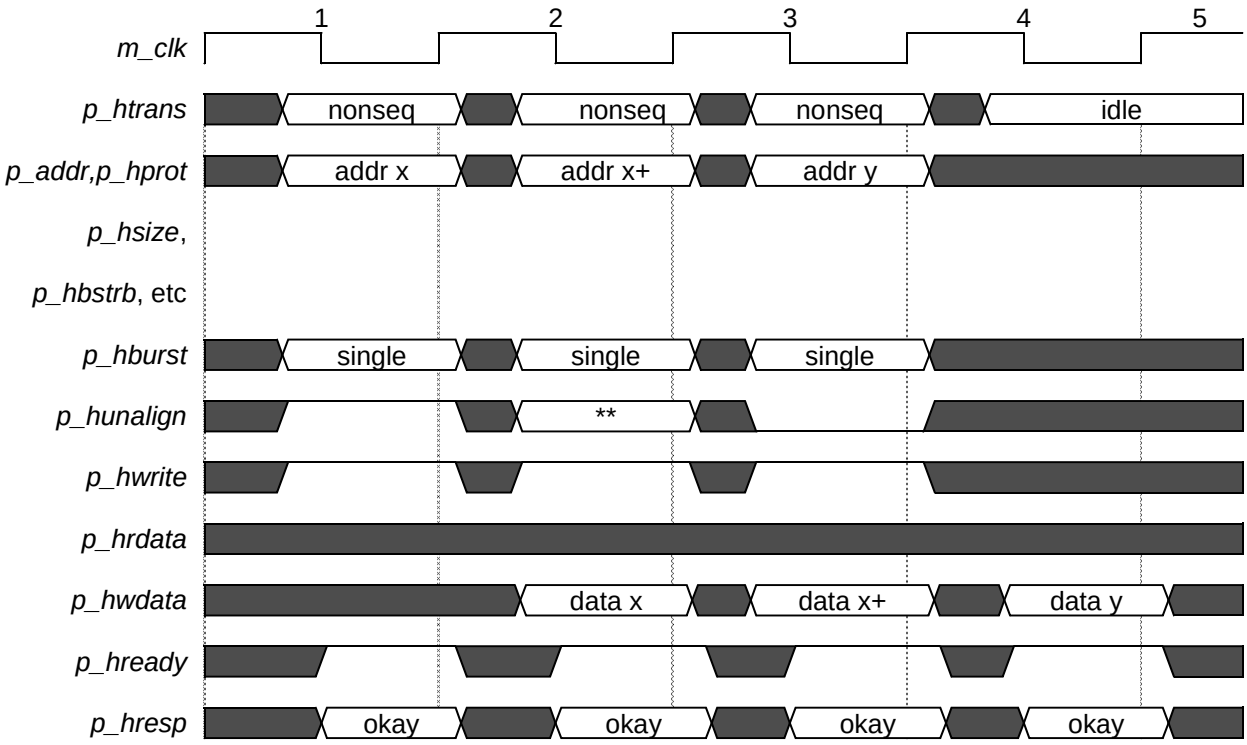


Figure 8-12. Misaligned Write, Write, Full Pipelining

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*) 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 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 that 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 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 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, *p_hunalign* 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 8-13. It is similar to the example in Figure 8-12.

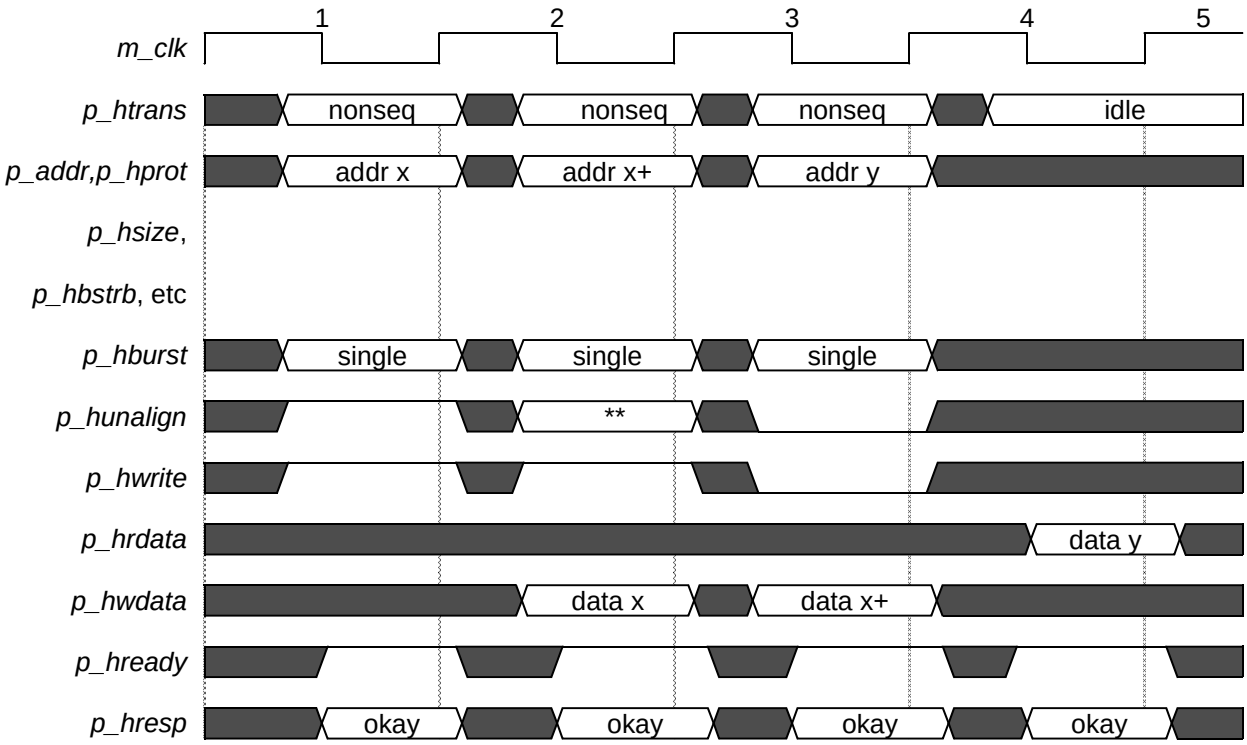


Figure 8-13. Misaligned Write, Single Cycle Read Transfer, Full Pipelining

8.5.1.7 Burst Accesses

Figure 8-14 illustrates functional timing for a burst read transfer.

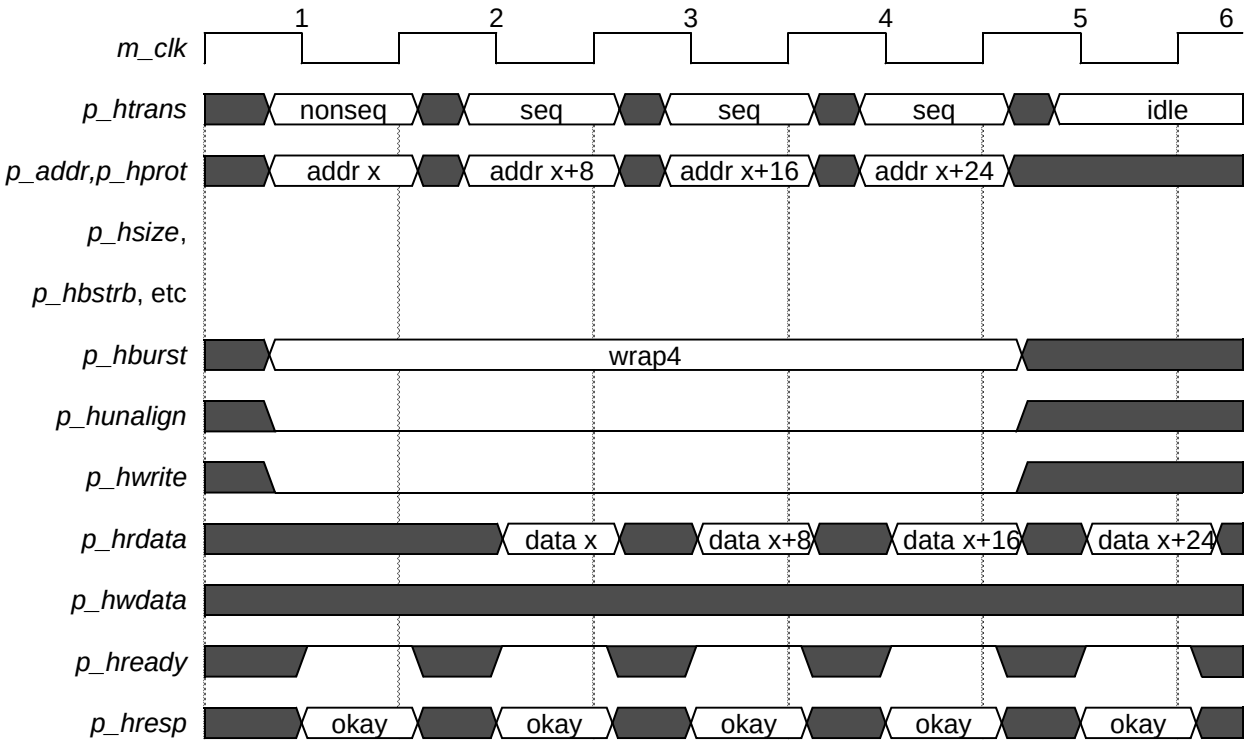


Figure 8-14. Burst Read Transfer

The *p_hburst* signals indicate WRAP4 for all burst transfers. The *p_hunalign* signal is negated. *p_hsize* indicates 64-bits, and all eight *p_hbstrb* signals are asserted. The burst address is aligned to a 64-bit boundary and wraps around modulo four double words. Note that in this example the *p_htrans* signals indicate IDLE after the last portion of the burst is 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 8-15 illustrates functional timing for a burst read with wait-state transfer.

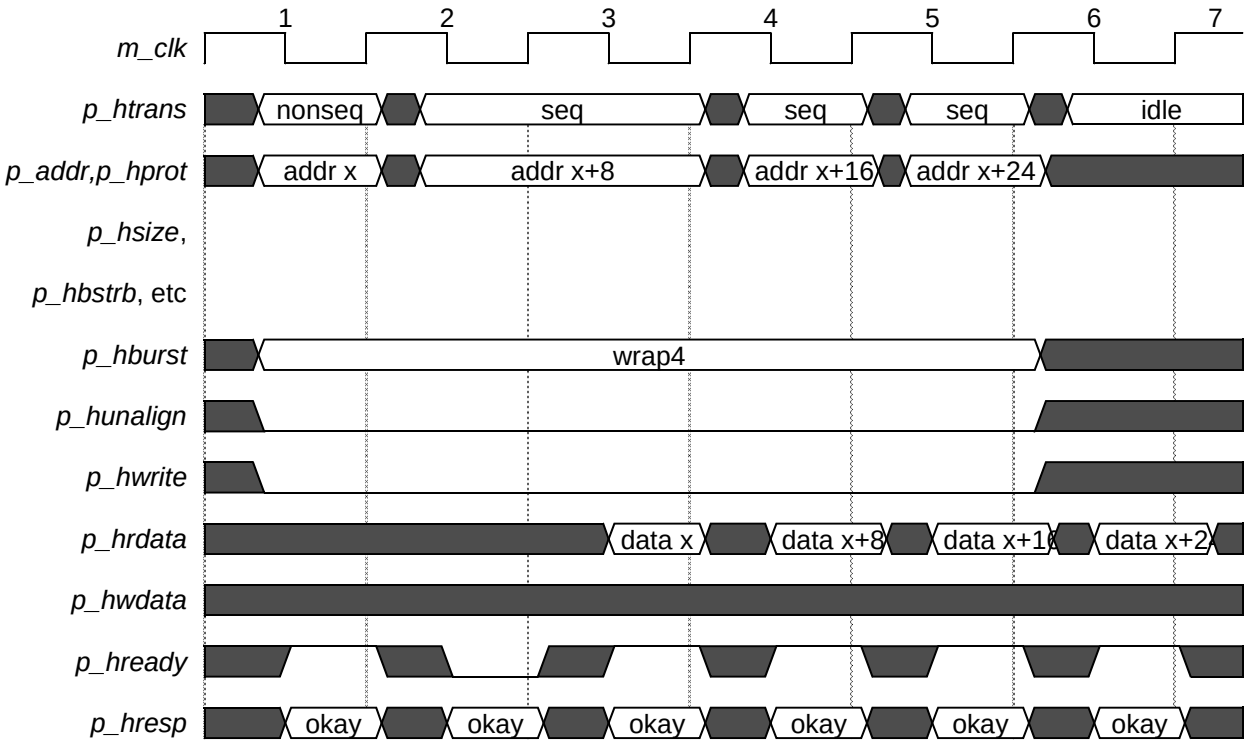


Figure 8-15. Burst Read with Wait-State Transfer

The first cycle of the burst incurs a single wait-state.

Figure 8-16 illustrates functional timing for a burst write transfer.

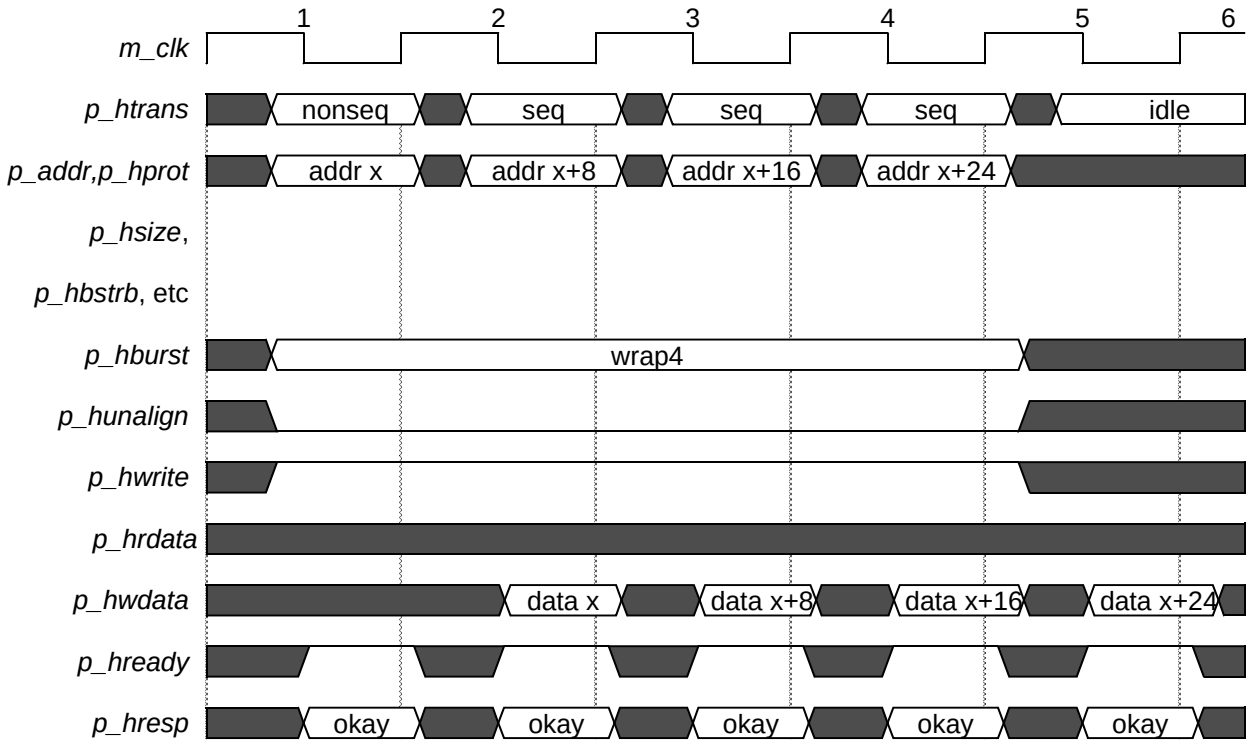


Figure 8-16. Burst Write Transfer

Figure 8-17 illustrates functional timing for a burst write with wait-state transfer.

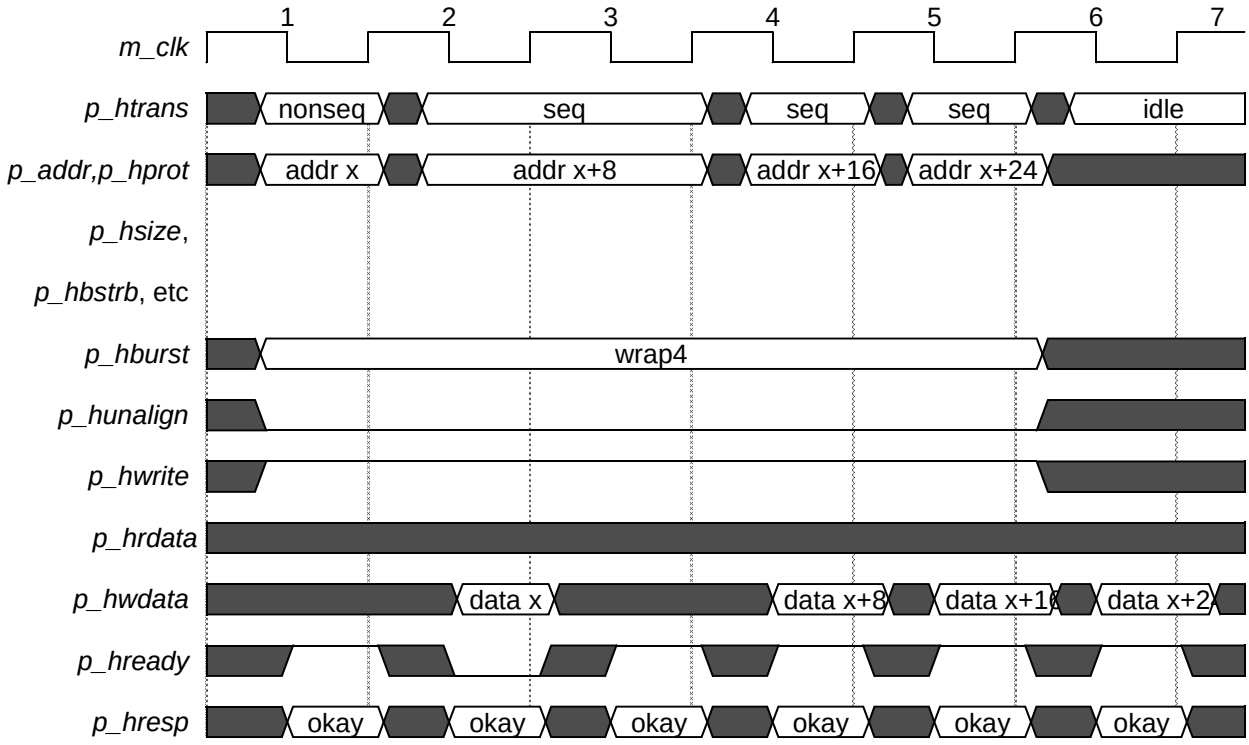


Figure 8-17. Burst Write with Wait-State Transfer

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.

8.5.1.8 Error Termination Operation

The *p_hresp[2:0]* inputs signal an error termination for an access in progress. The ERROR encoding is used 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 2-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 that may have been previously pending while waiting for a response that terminates with error may be changed. It is not required to remain unchanged when an error response is received.

Figure 8-18 shows an example of error termination.

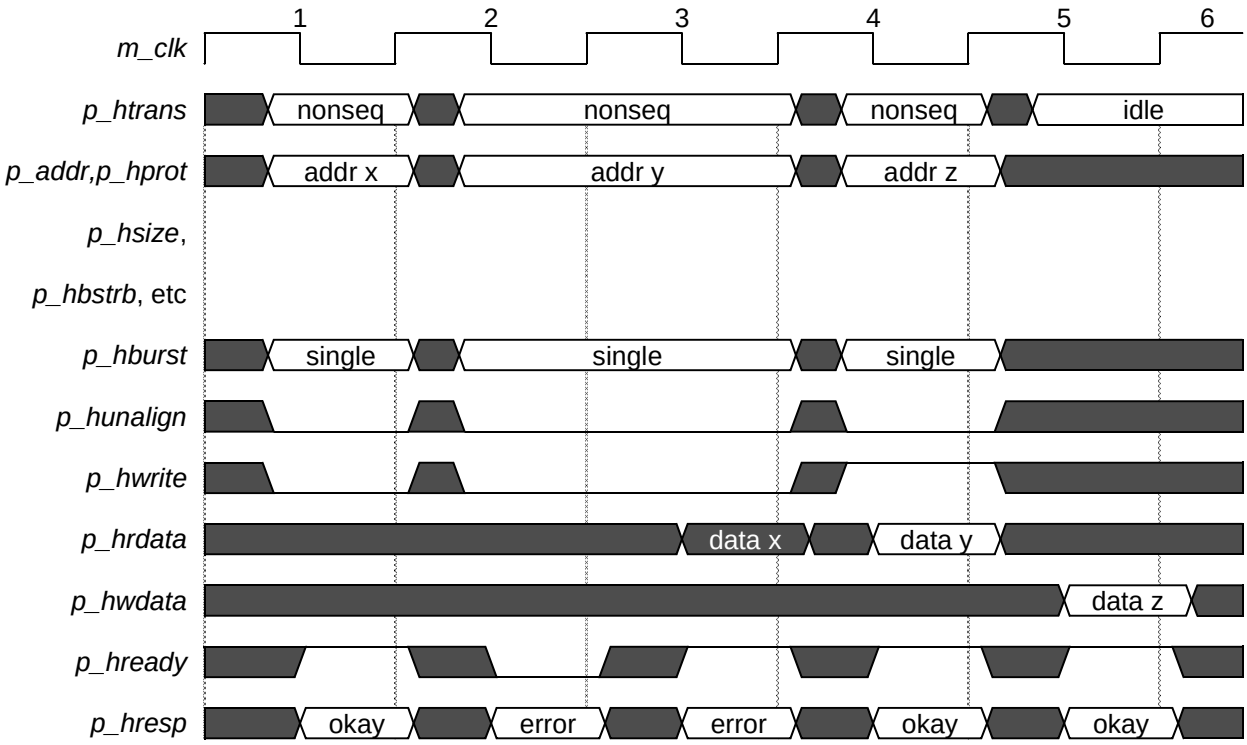


Figure 8-18. Read and Write Transfers: Instruction Read with Error, Data Read, Write, Full Pipelining

The first read request ($addr_x$) is taken at the end of cycle C1 because the bus is idle. It is an instruction prefetch.

The second read request ($addr_y$) is not taken at the end of C2 because 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.

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.

In this example of error termination, the CPU continues to request an access to $addr_y$. It is taken at the end of C3. During C4, read data is supplied for the $addr_y$ read, and the access is terminated normally during C4.

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.

Data for the $addr_z$ write cycle is driven in C5, the cycle after the access is *taken*.

During C5, a ready/OKAY response is signaled to complete the write cycle to $addr_z$.

In this example of error termination, a subsequent access remained requested. This does not always occur when certain types of transfers are terminated with error. The following figures outline cases where an error termination for a given cycle causes a pending request to be aborted prior to initiation.

Figure 8-19 shows another example of error termination.

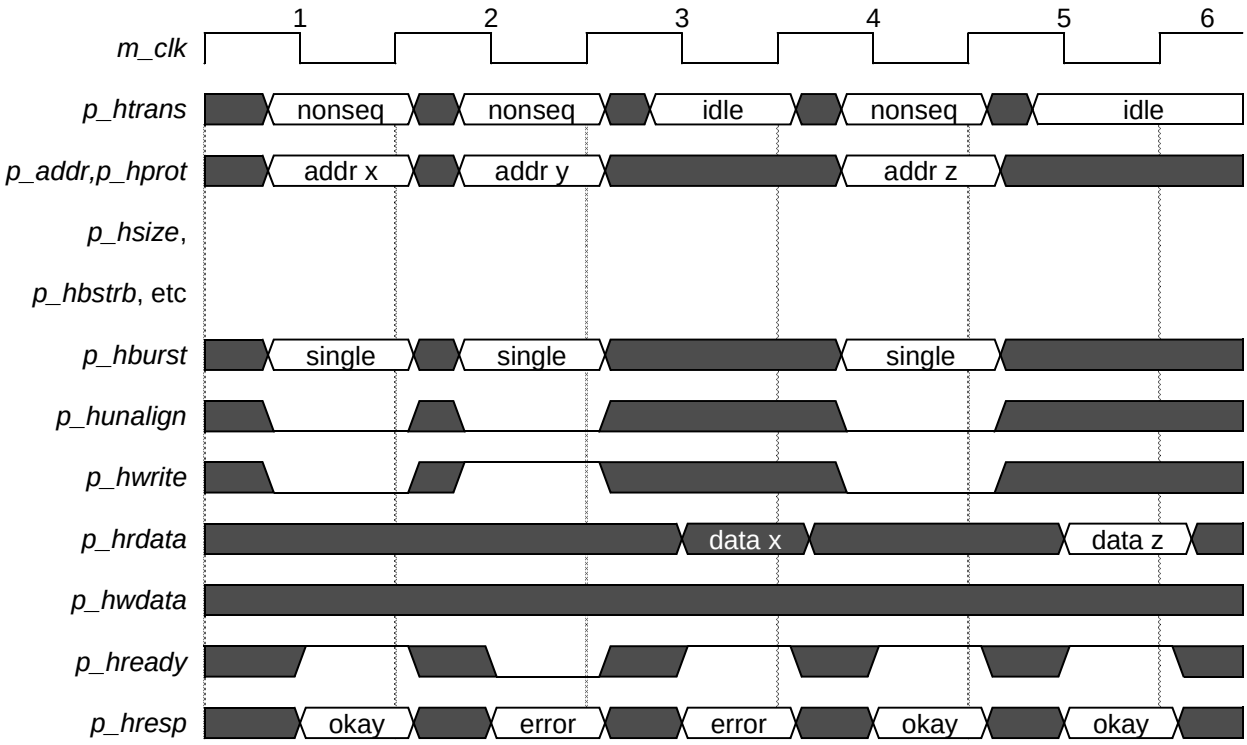


Figure 8-19. Data Read with Error, Data Write Retracted, Instruction Read, Full Pipelining

The first read request ($addr_x$) is *taken* at the end of cycle C1 because the bus is idle. It is a data read.

The second request (write to $addr_y$) is not *taken* at the end of C2 because 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.

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.

In this example of error termination, the CPU retracts the requested access to $addr_y$ by driving *p_htrans* signals to the IDLE state during the second cycle of the two-cycle error response.

A different access to $addr_z$ is requested during C4 and is taken at the end of C4. During C5, read data is supplied for the $addr_z$ read, and the access is terminated normally.

In this example of error termination, a subsequent access was aborted.

Figure 8-20 shows another example of error termination, this time on the initial portion of a misaligned write.

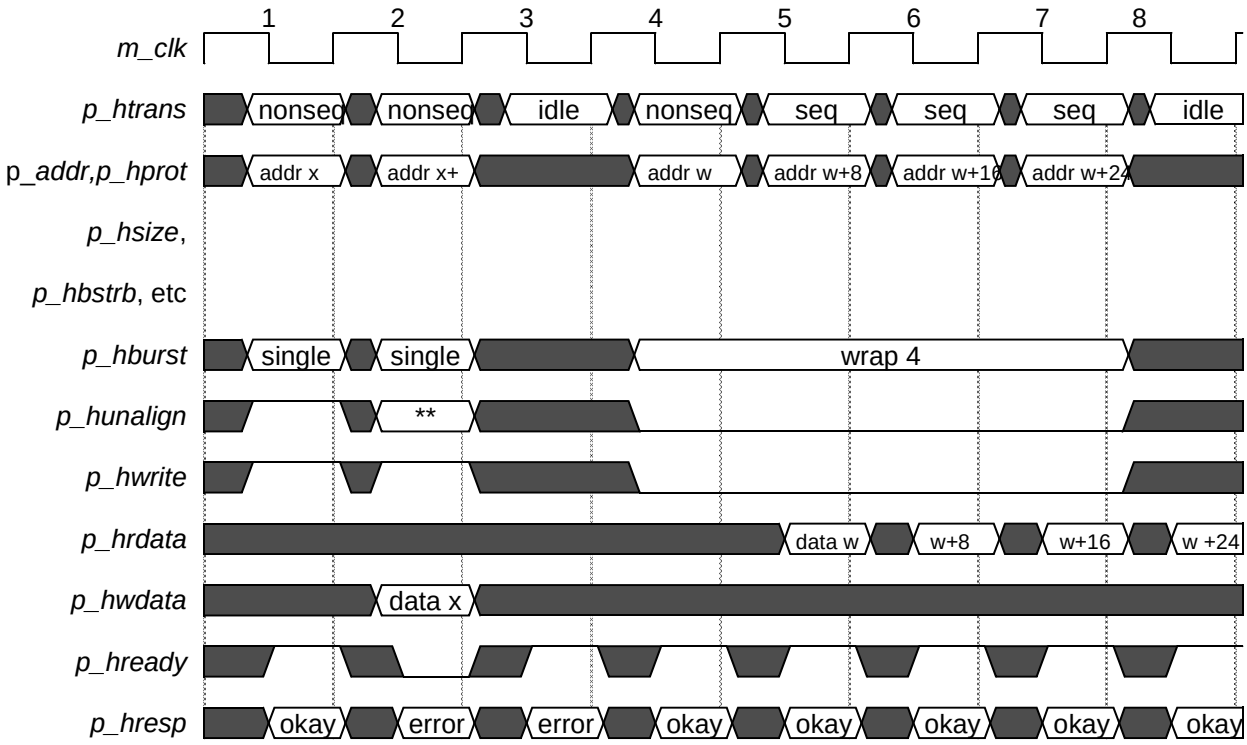


Figure 8-20. Misaligned Write with Error, Data Write Retracted, Burst Read Substituted, Full Pipelining

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 8-21 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.

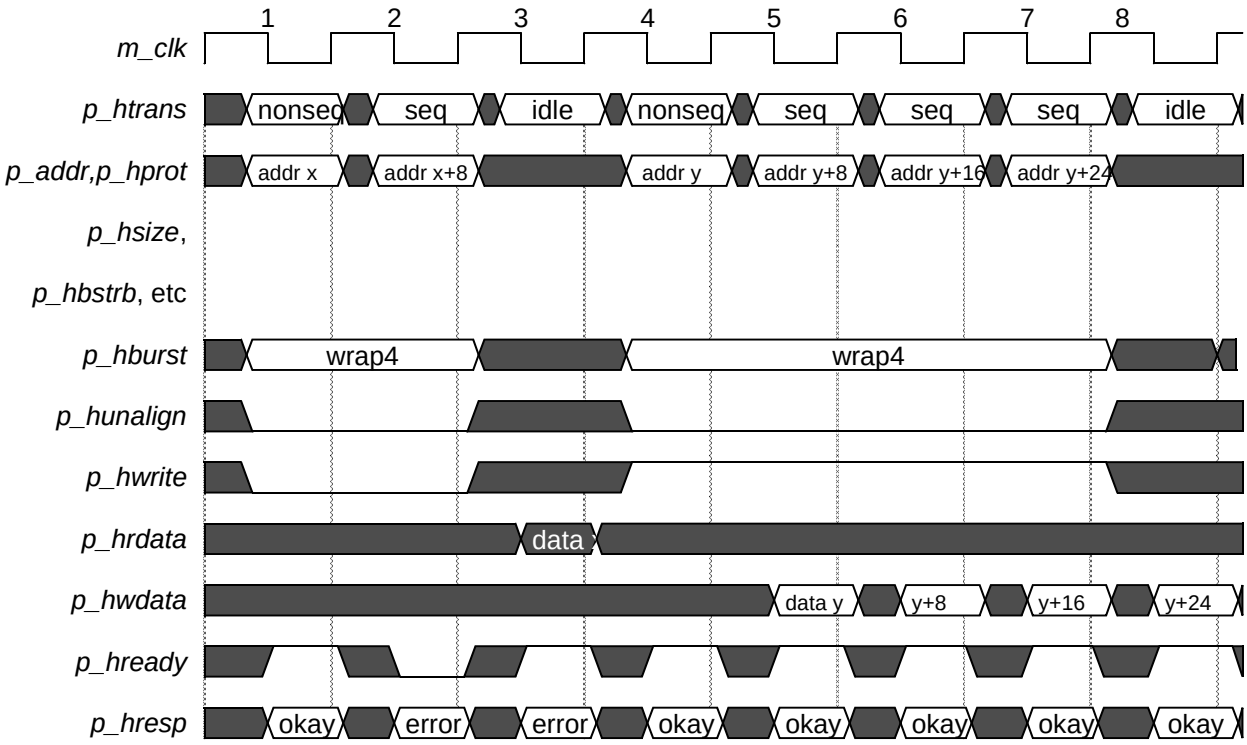


Figure 8-21. Burst Read with Error Termination, Burst Write

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.

8.5.2 Power Management

Figure 8-22 shows the relationship of the wake-up control signal *p_wakeup* to the relevant input signals.

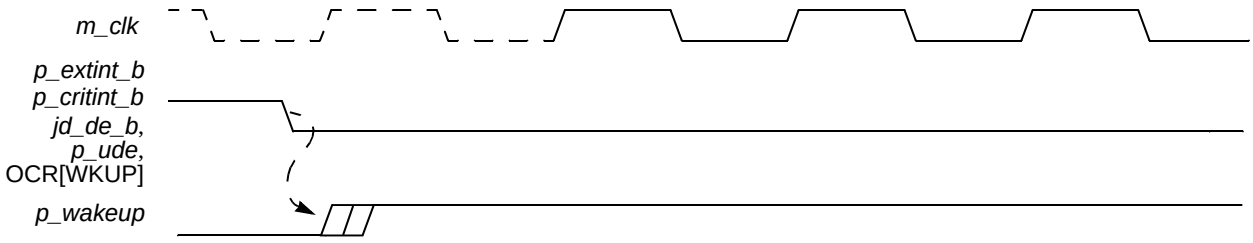


Figure 8-22. Wakeup Control Signal (*p_wakeup*)

8.5.3 Interrupt Interface

Figure 8-23 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 must meet setup and hold timing relative to the rising edge of the *m_clk*. In addition, during each clock cycle in which

either *p_extint_b* or *p_critint_b* is asserted, *p_avec_b* and *p_voffset[0:15]* are required to be in a valid state for the highest priority interrupt being requested.

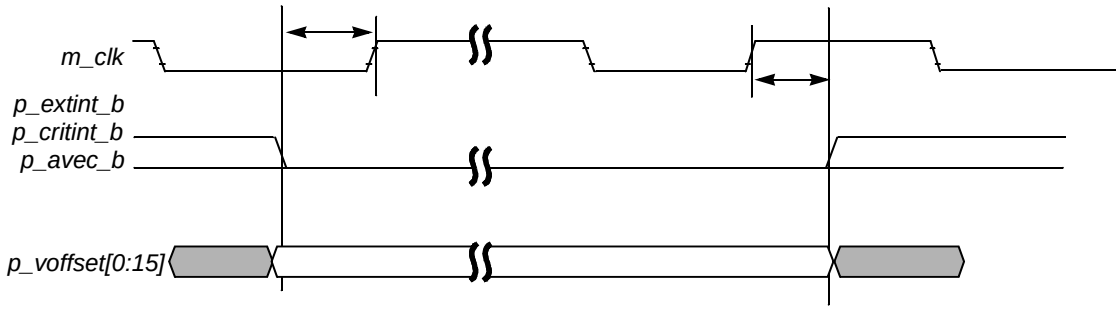


Figure 8-23. Interrupt Interface Input Signals

Figure 8-24 shows the interrupt pending signal's relationship to the interrupt request inputs. Note that *p_ipend* is asserted combinationally from the *p_extint_b* and *p_critint_b* inputs.

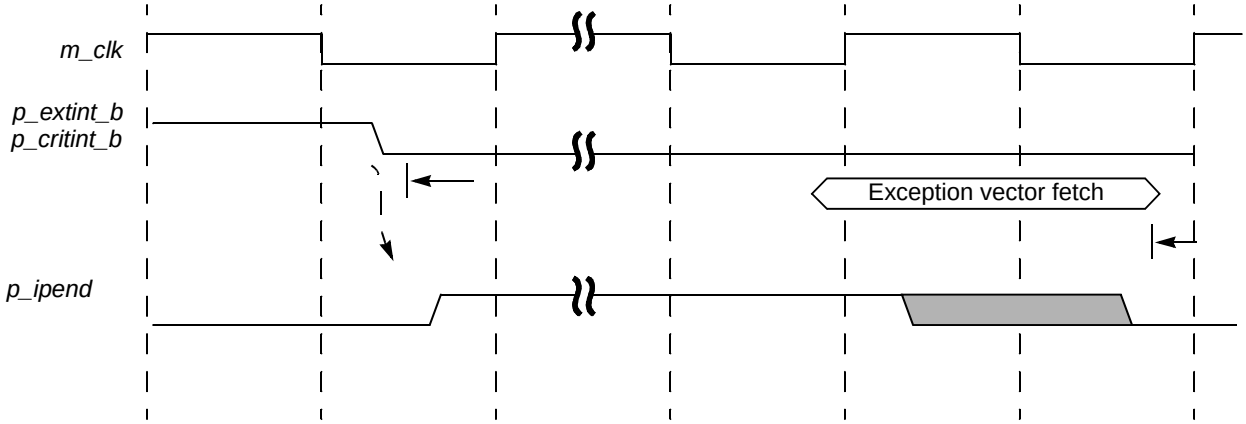


Figure 8-24. Interrupt Pending Operation

Figure 8-25 shows the relationship of the interrupt acknowledge signal to the interrupt request inputs and exception vector fetching.

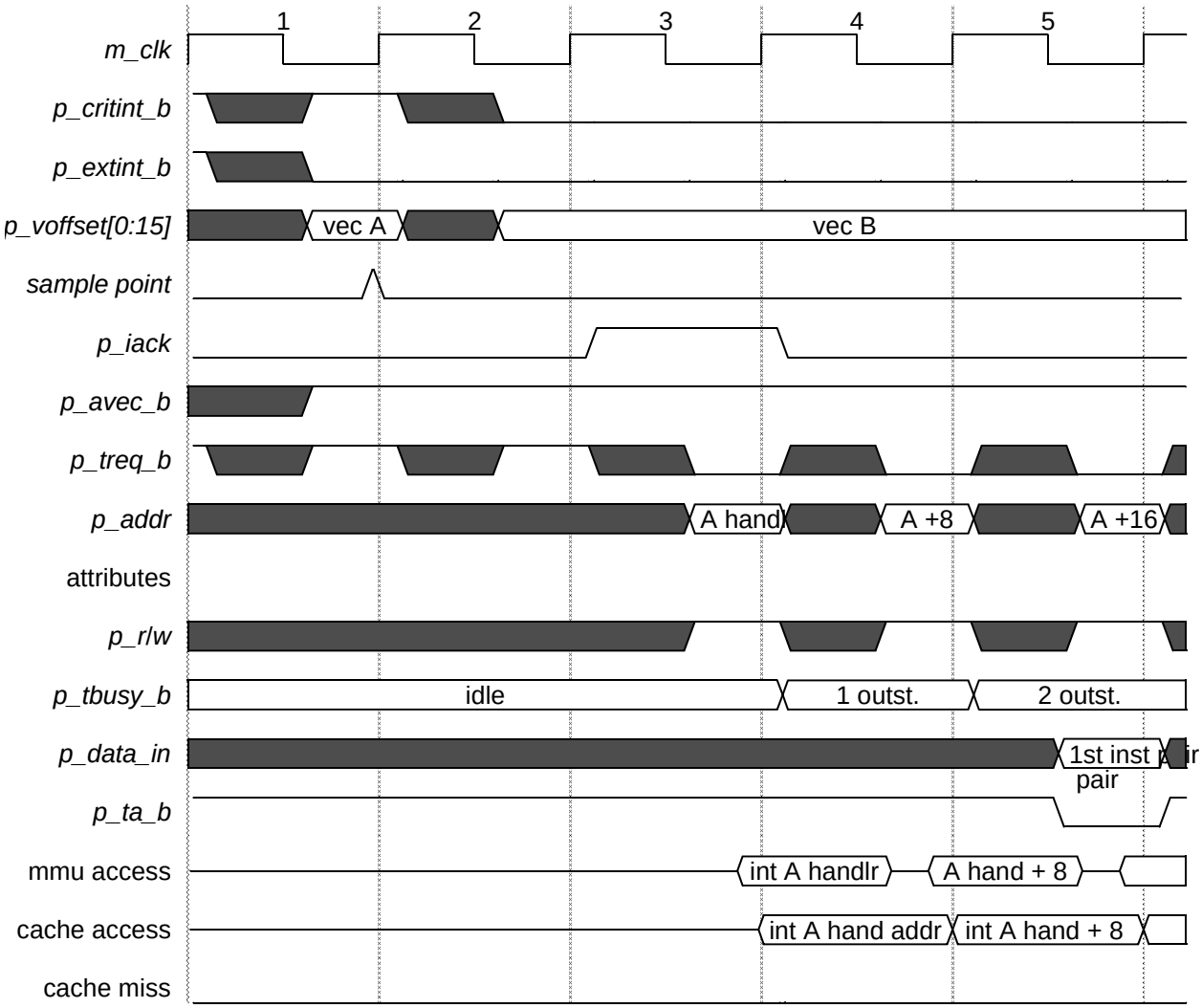


Figure 8-25. Interrupt Acknowledge Operation

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 and 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 at which an interrupt is committed to. In the following example, because 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 8-26.

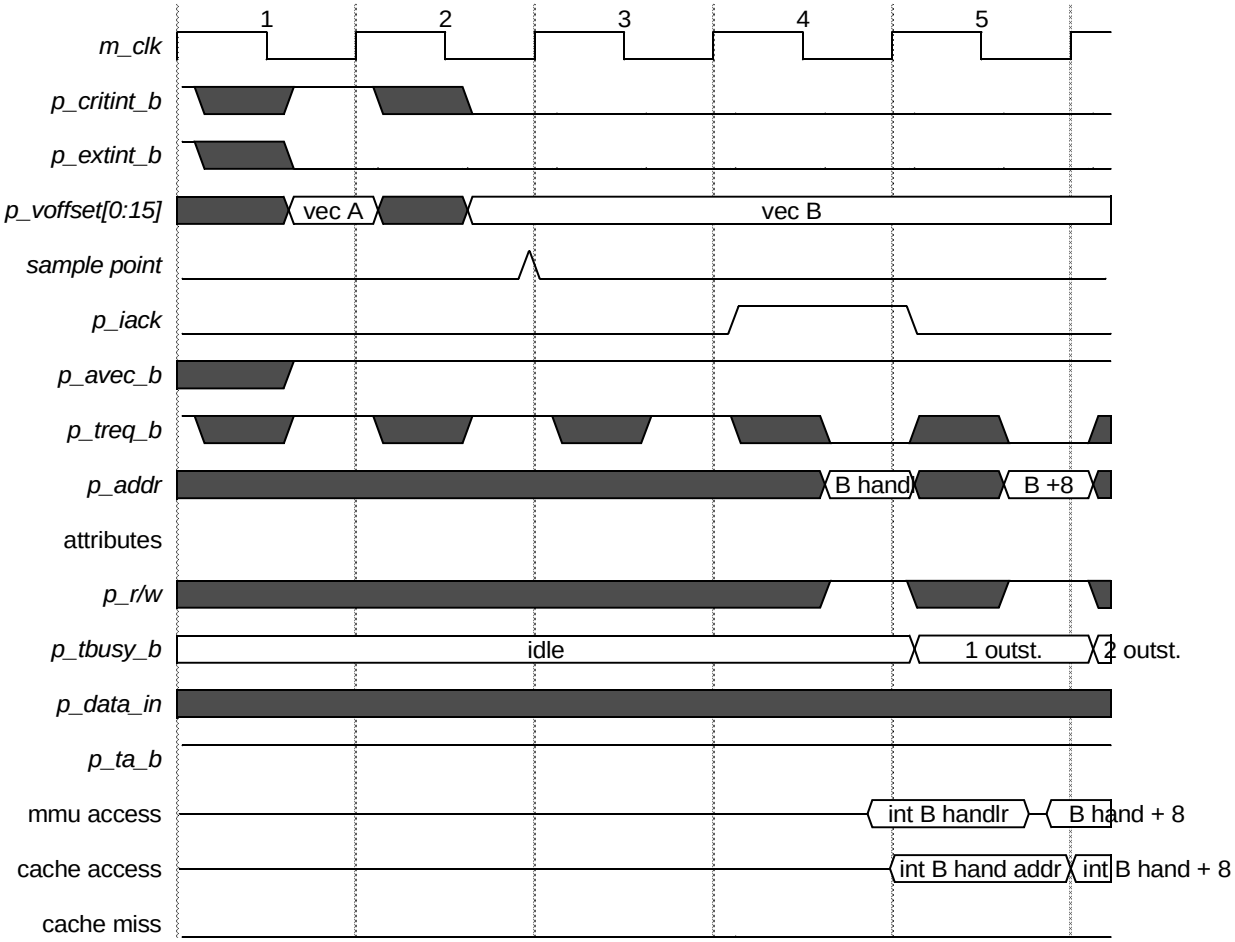


Figure 8-26. Interrupt Acknowledge Operation—2

Chapter 9
Power Management

This chapter describes the power management facilities as they are defined by Book E and implemented in devices that contain the e200z6 core. The scope of this chapter is limited to core complex features. Additional power management capabilities associated with a device that integrates this core (referred to as an integrated device) are documented separately.

9.1 Overview

e200z6 cores support power management to minimize overall system power consumption. The core provides the ability to initiate power management from external sources as well as through software techniques. Table 9-1 describes e200z6 core power states.

Table 9-1. Power States

State	Description
Active (default)	All internal units on the e200z6 core operate at full processor clock speed. The core provides dynamic power management in which idle internal units may stop clocking automatically.
Halted	Instruction execution and bus activity are suspended, and most internal clocks are gated off. The e200z6 core asserts *p_halted* to indicate it is in the halted state. Before entering halted state, all outstanding bus transactions complete, and the cache's store and push buffers are flushed. The *m_clk* input should remain running to allow further transitions into the power-down state if requested and to keep the time base operational if it is using *m_clk* as the clock source.
Power down (stopped)	All core functional units except the time base unit and clock control state machine logic are stopped. *m_clk* may be kept running to keep the time base active and to allow quick recovery to full-on state. Clocks are not running to functional units except to the time base. The core reaches power-down state after transitioning through halted state with *p_stop* asserted; at this point *p_stopped* output is asserted. Additional power may be saved by disabling the time base by asserting *p_tbdisable* or by integrated logic stopping *m_clk* after the core is in power-down state and has asserted *p_stopped*. To exit power-down state, integrated logic must first restart *m_clk*. Because the time base is off during power-down state, if *m_clk* is the clock source and is stopped, or if 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 to reinitialize the time base unit. A time-base related interrupt source (such as the decrementer) cannot be used to exit low power states. The e200z6 core also provides the ability to clock the time base from an independent (but externally synchronized) clock source, which allows the time base to be maintained during the power-down state, and allows a time-base related interrupt to be generated to indicate an exit condition from the power-down state.

Figure 9-1 is a power management state diagram.

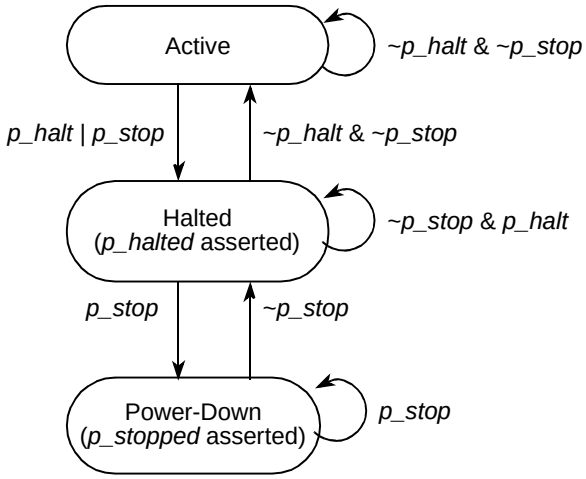


Figure 9-1. Power Management State Diagram

9.1.1 Power Management Signals

Table 9-1 summarizes power management signals. More detailed information is provided in Section 8.5.2, “Power Management.”

Table 9-2. Descriptions of Timer Facility and Power Management Signals

Signal	I/O	Signal Description
p_halt	I	Processor halt request. The active-high *p_halt* input requests that the core enter the halted state.
p_halted	O	Processor halted. The active-high *p_halted* output indicates that the core entered the halted state.
p_stop	I	Processor stop request. The active-high *p_stop* input requests that the core enter the stopped state.
p_stopped	O	Processor stopped. The active-high *p_stopped* output indicates that the core entered stopped state.
p_doze *p_nap* *p_sleep*	O	Low-power mode. These signals are asserted by the core to reflect the settings of the HID0[DOZE], HID0[NAP], and HID0[SLEEP] control bits when MSR[WE] is set. The e200z6 core can be placed in a low-power state by forcing *m_clk* 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.
p_wakeup	O	Wakeup. Used by external logic to remove the e200z6 core and system logic from a low-power state. It can also indicate to the system clock controller that *m_clk* should be re-enabled for debug purposes. *p_wakeup* (or other system state) should be monitored to determine when to release the core (and system if applicable) from a low-power state.
p_tbdisable	I	Timer disable. Used to disable the internal time base and decrementer counters. This signal can be used to freeze the state of the time base and decrementer during low power or debug operation.

Table 9-2. Descriptions of Timer Facility and Power Management Signals

Signal	I/O	Signal Description
p_tbclk	I	Timer external clock. Used as an alternate clock source for the time base and decrementer counters. Selection of this clock is made using HID0[SEL_TBCLK] (see Section 2.11.1, “Hardware Implementation-Dependent Register 0 (HID0)”).
p_tbint	O	Timer interrupt status. Indicates whether an internal timer facility unit is requesting an interrupt (TSR[WIS]=1 and TCR[WIE]=1, or TSR[DIS]=1 and TCR[DIE]=1, or TSR[FIS]=1 and TCR[FIE]=1). May be used to exit low power operation or for other system purposes.

9.1.2 Power Management Control Bits

Software uses the register fields listed in Table 9-3 to generate a request to enter a power-saving state and to choose the state to be entered.

Table 9-3. Power Management Control Bits

Bit	Description
MSR[WE]	Used to qualify assertion of the *p_doze*, *p_nap*, and *p_sleep* outputs to the integrated logic. When MSR[WE] is negated, these signals are negated. If MSR[WE] is set, these pins reflect the state of their respective HID0 control bits.
HID0[DOZE]	The interpretation of the DOZE mode bit is done by the external integrated logic. Doze mode on the e200z6 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 integrated logic. Nap mode on the e200z6 core may be used for a power-down state with the time base enabled.
HID0[SLEEP]	The interpretation of the SLEEP mode bit is done by the external integrated logic. Sleep mode on the e200z6 core may be used for a power-down state with the time base disabled.

9.1.3 Software Considerations for Power Management

Setting MSR[WE] generates a request to enter a power-saving state (doze, nap, or sleep). This state must be previously determined by setting the appropriate HID0 bit. Setting MSR[WE] does not directly affect execution, but is reflected on *p_doze*, *p_nap*, and *p_sleep*, depending on the setting of the HID0 DOZE, NAP, and SLEEP bits. Note that the core is not affected by assertion of these signals directly. External system hardware may interpret the state of these signals and activate the *p_halt* and/or *p_stop* inputs to cause the e200z6 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
```

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. 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.

9.1.4 Debug Considerations for Power Management

When a debug request is presented to the e200z6 core when it is in either the halted or stopped state, *p_wakeup* is asserted, and when *m_clk* is provided to the CPU, it temporarily exits the halted or stopped state and enters debug mode, regardless of the assertion of *p_halt* or *p_stop*. The *p_halted* and *p_stopped* outputs are negated as long as 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 halted or stopped state as appropriate.

Chapter 10
Debug Support

10.1 Introduction

This chapter describes the debug features of the e200z6 core and describes the e200z6 software and hardware debug facilities, events, and registers. It also details the external debug support features available and introduces the reader to the on-chip emulation circuitry (OnCE) and its key attributes, that is, the interface signals, debug inputs, and outputs. This chapter also covers watchpoint support, MMU and cache operations during debug, cache array access, and the basic steps for enabling, using, and exiting external debug mode.

10.2 Overview

Internal debug support in the e200z6 core allows for software and hardware debugging by providing debug functions such as instruction and data breakpoints and program trace modes. For software-based debugging, debug facilities consisting of a set of software-accessible debug registers and interrupt mechanisms are provided. These facilities are also available to a hardware-based debugger that communicates using a modified IEEE 1149.1 test access port (TAP) controller and pin interface. When hardware debugging is enabled, the debug facilities are protected from software modification.

Software debug facilities are defined as part of Book E. The e200z6 supports a subset of these defined facilities. In addition to the Book E–defined facilities, the e200z6 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 e200z6 core also supports an external Nexus real-time debug module. Real-time debugging in an e200z6-based system is supported by an external Nexus class 2, 3, or 4 module. Definitions and features of this module are part of the system/platform specification and are not further defined in this chapter. Additional information can be found in Chapter 11, “Nexus3 Module.”

10.2.1 Software Debug Facilities

The e200z6 debug facilities 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–DBCR3), a set of address compare registers (IAC1–IAC4, DAC1, and DAC2), a configurable debug counter register (DBCNT), 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 5.6.16, “Debug Interrupt (IVOR15)”). The debug facilities also provide mechanisms 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, DBCR0[IDM]. If DBCR0[IDM] is set, debug events can occur and can be enabled to record exceptions in the DBSR. If enabled by MSR[DE], these exceptions cause debug interrupts. If DBCR0[IDM] and DBCR0[EDM] (EDM represents the external debug mode bit) are cleared, no debug events occur and no status flags are set in DBSR unless already set. In addition, if DBCR0[IDM] is cleared (or is overridden by DBCR0[EDM] being set), no debug interrupts can occur, regardless of the contents of DBSR. A software debug interrupt handler can access all system resources and perform the necessary functions appropriate for system debugging.

10.2.1.1 PowerPC Book E Compatibility

The e200z6 core implements a subset of the PowerPC Book E 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.
- Data value compares are not supported.

10.2.2 Additional Debug Facilities

In addition to the debug functionality defined in Book E, the e200z6 provides the capability to link instruction and data breakpoints. The e200z6 also provides a configurable debug event counter to allow debug exception generation and a sequential breakpoint control mechanism.

The e200z6 also defines two new debug events (critical interrupt taken and critical return) for debugging around critical interrupts.

In addition, the e200z6 implements the debug auxiliary processing unit (APU) which, when enabled, allows debug interrupts to use a dedicated set of save/restore registers (DSRR0 and DSRR1) to save state information when a debug interrupt occurs and restore this state information at the end of a debug interrupt handler with the **rfdi** instruction.

10.2.3 Hardware Debug Facilities

The e200z6 core contains facilities that allow for external test and debugging. A modified IEEE 1149.1 control interface is used to communicate with core resources. This interface is implemented through a standard 1149.1 TAP (test access port) controller.

By using public instructions, the external debugger can freeze or halt the e200z6 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 (DBCR0[EDM]). Setting DBCR0[EDM] overrides the internal debug mode enable bit DBCR0[IDM]. If the hardware debug facility is enabled, software is blocked from modifying the debug facilities. In addition, because resources are owned by the hardware debugger, inconsistent values may be present if software attempts to read debug-related resources.

When hardware debug is enabled (DBCR0[EDM] = 1), the registers and resources described in Section 10.3, “Debug Registers,” are reserved for use by the external debugger. The events described in Section 10.3, “Debug Registers,” are also used for external debugging, but exceptions are not generated to running software. Debug events enabled in the respective DBCR0–DBCR3 registers are recorded in the DBSR regardless of MSR[DE], and no debug interrupts are generated. Instead, the CPU enters debug mode when an enabled event causes a DBSR bit to become set. DBCR0[EDM] may only be written through the OnCE port.

A program trace program counter FIFO (PC FIFO) is also provided to support program change-of-flow capture.

To perform write accesses from the external hardware debugger, most debug resources (registers) require the CPU clock (*m_clk*) to be running.

Figure 10-1 shows the e200z6 debug resources.

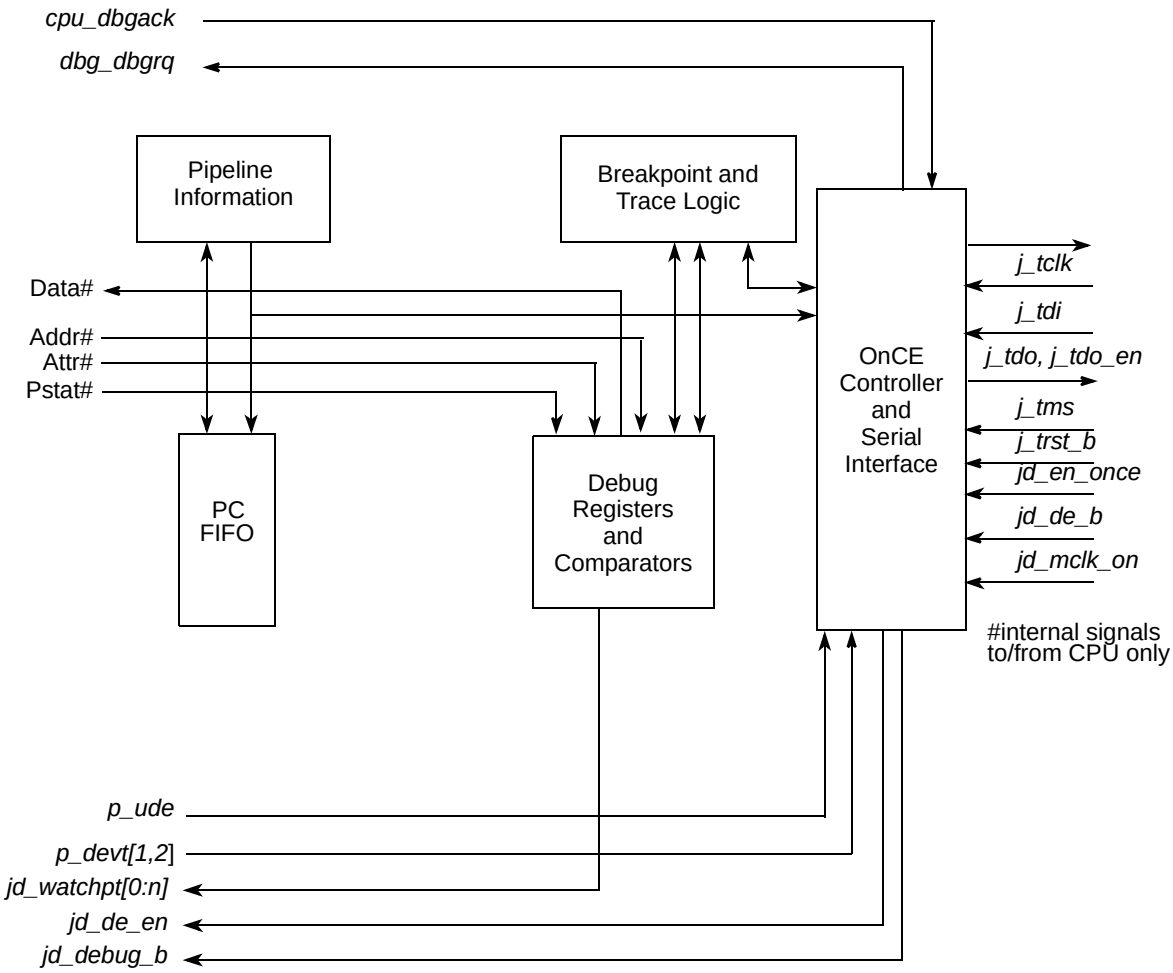


Figure 10-1. e200z6 Debug Resources

10.3 Debug Registers

The debug facility registers are listed in Table 10-1 and described in Section 2.10, “Debug Registers.”

Table 10-1. Debug Registers

Mnemonic	Name	SPR Number	Access	Privileged	e200z6 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
DBSR	Debug status register	304	Read/Clear [1]	Yes	No
DBCNT	Debug counter register	562	R/W	Yes	Yes
IAC1	Instruction address compare 1	312	R/W	Yes	No

Table 10-1. Debug Registers (continued)

Mnemonic	Name	SPR Number	Access	Privileged	e200z6 Specific
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
DAC1	Data address compare 1	316	R/W	Yes	No
DAC2	Data address compare 2	317	R/W	Yes	No

[1] The DBSR can be read using **mfspr rD,DBSR**. It cannot be directly written to. Instead, DBSR bits corresponding to 1 bits in GPR(**r**S) can be cleared using **mtspr DBSR,r**S.

10.4 Software Debug Events and Exceptions

Software debug events and exceptions are available if internal debug mode is enabled (DBCR0[IDM] = 1) and not overridden by external debug mode (DBCR0[EDM] = 0). When enabled, debug events cause debug exceptions to be recorded in the debug status register. Specific event types are enabled by DBCR0–DBCR3. The unconditional debug event (UDE) is an exception to this rule; it is always enabled. Once a DBSR bit other than MRR and CNT1TRG is set, if debug interrupts are enabled by MSR[DE], a debug interrupt is generated. 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[EDM] = 0. Under these conditions, no debug exception occurs and thus no DBSR bit is set. Other debug events may cause debug exceptions and set DBSR bits regardless of the state of MSR[DE]. A debug interrupt is delayed until MSR[DE] is set.

When a DBSR bit is set while MSR[DE] = 0 and DBCR0[EDM] = 0, an imprecise debug event flag (DBSR[IDE]) is also set to indicate that an exception bit in the DBSR was set while debug interrupts were disabled. The debug interrupt handler software can use this bit to determine whether DSRR0 holds the address associated with the instruction causing the debug exception or the address of the instruction that enabled a delayed debug interrupt by setting MSR[DE]. An **mtmsr** or **mtdbcr0**, which causes both MSR[DE] and DBCR0[IDM] to be set, enabling precise debug mode, may cause an imprecise (delayed) debug exception to be generated due to an earlier recorded event in the DBSR.

The following types of debug events are defined by Book E:

- 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 further detail in the EREF.

The e200z6 defines the following debug events, which are described in Table 10-2:

- The debug counter debug events DCNT1 and DCNT2
- The external debug events DEVT1 and DEVT2
- The critical interrupt taken debug event (CIRPT)
- The critical return debug event (CRET)

The e200z6 debug framework supports most of these event types. The following Book E–defined functionality is not supported:

- Instruction address compare and data address compare real address mode
- Data value compare mode

A brief description of each of the debug event types is shown in Table 10-2. In these descriptions, DSRR0 and DSRR1 are used to store the address of the instruction following a load or store, assuming that the debug APU is enabled. If it is disabled, CSRR0 is used.

Table 10-2. Debug Event Descriptions

Event Name	Type	Description
Instruction address compare event	IAC	Instruction address compare debug events occur if instruction address compare debug events are enabled and execution is attempted of an instruction at an address that meets the criteria specified in DBCR0, DBCR1, IAC1–IAC3, and IAC4. 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. This event can occur and be recorded in DBSR regardless of the setting of MSR[DE]. If a higher-priority interrupt (such as an asynchronous interrupt) prevents the instruction that had the IAC match from executing, the e200z6 behaves as if the IAC event had not occurred; for example, the corresponding DBSR[IAC*n*] bit is not set.
Data address compare event	DAC	Data address compare debug events occur if data address compare debug events are enabled and execution of a load or store class instruction or a cache maintenance instruction results in a data access with an address that meets the criteria specified in DBCR0, DBCR2, DAC1, and DAC2. 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 and DAC2) are provided. **Note:** In contrast to the Book E definition, data address compare events on the e200z6 do not prevent the load or store 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 APU is disabled) is the address of the instruction following the load or store class instruction. 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, the result is an imprecise debug interrupt, the address value saved in DSRR0 (or CSRR0 if the debug APU is disabled) is the address of the load or store class instruction, and DBSR[IDE] is set. In addition to occurring when DBCR0[IDM] = 1, this can also occur when DBCR0[EDM] = 1. **Note:** DAC events are not recorded or counted if an **lmw** or **stmw** instruction is interrupted before completion by a critical input or external input interrupt. **Note:** • DAC events are not signaled on: —The second portion of a misaligned load or store that is broken up into two separate accesses —The **tlbre**, **tlbwe**, **tlbsx**, or **tlbivax** instructions

Table 10-2. Debug Event Descriptions (continued)

Event Name	Type	Description
Linked instruction address and data address compare event	DAC1LNK, DAC2LNK	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 DBSR, regardless of whether a corresponding DAC1 (or DAC2) event occurs, or whether the IAC1 (or IAC3) event enable is set. When enabled and execution of a load or store class instruction results in a data access with an address, and that address meets the criteria specified in DBCR0, DBCR2, DAC1, and DAC2, and the instruction also meets the criteria for generating an instruction address compare event, a linked data address compare debug event occurs. This event can occur and be recorded in DBSR regardless of the setting of MSR[DE]. The normal DAC1 and DAC2 status bits in the DBSR are used for recording these events. The IAC1 and IAC3 status bits are not set if the corresponding instruction address compare register is linked. Linking is enabled using DBCR2 control bits. 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 that are linked may still affect the debug counter (if enabled to) and 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 an **lmw** or **stmw** instruction is interrupted before completion by a critical input or external input interrupt.
Trap debug event	TRAP	A trap debug event occurs if trap debug events are enabled (DBCR0[TRAP] = 1), a trap instruction (**tw, twi**) is executed, and the conditions specified by the instruction for the trap are met. This event can occur and be recorded in DBSR regardless of the setting of MSR[DE]. When a trap debug event occurs, DBSR[TRAP] is set.
Branch taken debug event	BRT	A branch taken debug event occurs if branch taken debug events are enabled (DBCR0[BRT] = 1) and execution is attempted of a branch instruction that will be taken (either an unconditional branch or a conditional branch whose branch condition is true), and MSR[DE] = 1 or DBCR0[EDM] = 1. Branch taken debug events are not recognized if MSR[DE] = 0 and DBCR0[EDM] = 0 at the time of execution of the branch instruction and thus DBSR[IDE] can not be set by a branch taken debug event. When a branch taken debug event is recognized, DBSR[BRT] is set to record the debug exception, and the address of the branch instruction is recorded in DSRR0 (only when the interrupt is taken).

Table 10-2. Debug Event Descriptions (continued)

Event Name	Type	Description
Instruction complete debug event	IAC	An instruction complete debug event occurs if instruction complete debug events are enabled (DBCR0[ICMP] = 1), execution of any instruction is completed, and MSR[DE] = 1 or DBCR0[EDM] = 1. If execution of an instruction is suppressed due to the instruction causing some other exception which is enabled to generate an interrupt, then the attempted execution of that instruction does not cause an instruction complete debug event. The **sc** instruction does not fall into the category of an instruction whose execution is suppressed, since the instruction actually executes and then generates a system call interrupt. In this case, the instruction complete debug exception is also set. When an instruction complete debug event is recognized, DBSR[ICMP] is set to record the debug exception and the address of the next instruction to be executed 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 DBSR[ICMP] and DBSR[IDE] to be set. This occurs when an embedded FPU round exception occurs. Because the instruction is by definition completed (SRR0 points to the following instruction), this interrupt takes higher priority than the debug interrupt so as not to be lost, and DBSR[IDE] = 1 to indicate imprecise recognition of a debug interrupt. In this case, the debug interrupt is taken with SRR0 pointing to the instruction following the instruction which generated the SPEFPU round exception, and DSRR0 points to the round exception handler. In addition to occurring when DBCR0[IDM] = 1, this circumstance can also occur when DBCR0[EDM] = 1. **Note:** Instruction complete debug events are not generated by the execution of an instruction that sets MSR[DE] while DBCR0[ICMP] = 1, nor by the execution of an instruction that sets DBCR0[ICMP] while MSR[DE] = 1 or DBCR0[EDM] = 1.
Interrupt taken debug event	IRPT	An interrupt-taken debug event 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, DBSR[IRPT] is set to record the debug exception. DSRR0 holds the address of the non-critical interrupt handler.
Critical interrupt taken debug event	CIRPT	A critical interrupt taken debug event occurs if critical interrupt taken debug events are enabled (DBCR0[CIRPT] = 1) and a critical interrupt (other than a debug interrupt when the debug APU is disabled) occurs. Only critical class interrupts cause a critical interrupt taken debug event. This event can occur and be recorded in DBSR regardless of the setting of MSR[DE]. When a critical interrupt taken debug event occurs, DBSR[CIRPT] bit is set, ensuring that debug exceptions are recorded. DSRR0 holds the address of the critical interrupt handler. **Note:** To avoid corruption of CSRR0 or CSRR1, this debug event should not normally be enabled unless the debug APU is also enabled.
Return debug event	RET	A return debug event occurs if return debug events are enabled (DBCR0[RET] = 1) and an attempt is made to execute an **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 so the debug exceptions are recorded. If MSR[DE] = 0 and DBCR0[EDM] = 0 when **rfi** executes (that is, before the MSR is updated by the **rfi**), DBSR[IDE] is also set to record the imprecise debug event. If MSR[DE] = 1 when **rfi** executes, a debug interrupt occurs provided no higher priority exception is enabled to cause an interrupt. DSRR0 holds the address of the **rfi** instruction.

Table 10-2. Debug Event Descriptions (continued)

Event Name	Type	Description
Critical return debug event	CRET	A critical return debug event occurs if critical return debug events are enabled (DBCR0[CRET] = 1) and an attempt is made to execute an **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 record the debug exception. If MSR[DE] = 0 and DBCR0[EDM] = 0 at the time of the execution of the **rfci** (that is before the MSR is updated by the **rfci**), DBSR[IDE] is also set to record the imprecise debug event. If MSR[DE] = 1 at the time of the execution of the **rfci**, a debug interrupt occurs provided no higher priority exception is enabled to cause an interrupt. Debug save/restore register 0 is set to the address of the **rfci** instruction. Note that this debug event should not normally be enabled unless the debug APU is also enabled to avoid corruption of CSRR0 or CSRR1.
Debug counter debug event	DCNT1, DCNT2	A debug counter debug event 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[DCNT1] or DBSR[DCNT2] is set to record the debug exception.
External debug event	DEVT1, DEVT2	An external debug event 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 set state. This event can occur and be recorded in DBSR regardless of the setting of MSR[DE]. When an external debug event occurs, DBSR[DEVT1] or DBSR[DEVT2] is set to record the debug exception.
Unconditional debug event	UDE	An unconditional debug event occurs when the unconditional debug event (*p_ude*) input transitions to the set 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, so debug exceptions are recorded.

10.5 External Debug Support

External debug support is supplied through the e200z6 OnCE controller serial interface, which allows access to internal CPU registers and other system state while in external debug mode (DBCR0[EDM] is set). All debug resources, including DBCR0–DBCR3, DBSR, IAC1–IAC4, DAC1, DAC2 and DBCNT are accessible through the serial on-chip emulation (OnCE) interface in external debug mode. Setting the DBCR0[EDM] bit through the OnCE interface enables external debug mode and disables software updates to the debug registers. When DBCR0[EDM] is set, debug events enabled to set respective DBSR status bits also cause the CPU to enter debug mode, as opposed to generating debug interrupts. In debug mode, the CPU is halted at a recoverable boundary, and an external debug control module may control CPU operation through the OnCE logic. No debug interrupts can occur while DBCR0[EDM] remains set.

NOTE

On the initial setting of DBCR0[EDM], other bits in DBCR0 are unchanged. After DBCR0[EDM] is set, all debug register resources may be subsequently controlled through the OnCE interface. DBSR should be cleared as part of the process of enabling external debug activity. The CPU should be placed into debug mode through the OCR[DR] control bit before setting EDM. This allows the debugger to cleanly write to the DBCR*n* registers and the DBSR to clear out any residual state/control information that could cause unintended operation.

NOTE

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 through the OCR[DR], by performing OnCE Go+NoExit commands, or by assertion of *jd_de_b*.

NOTE

DBCR0[EDM] operation is blocked if the OnCE operation is disabled (*jd_en_once* negated) regardless of whether it is set or cleared. This means that if DBCR0[EDM] was previously set and then *jd_en_once* 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 for writes to be performed to debug registers other than the OnCE command register (OCMD), the OnCE control register (OCR), or DBCR0[EDM]. Register read data is synchronized back to the *j_tclk* clock domain. The OnCE control register provides the capability of signaling the system level clock controller that the CPU clock should be activated if not already active.

Updates to DBCR*n*, DBSR, and DBCNT through 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 incorrect status information posted in DBSR.

10.5.1 OnCE Introduction

The e200z6 on-chip emulation circuitry (OnCE/Nexus class 1 interface) provides a means of interacting with the e200z6 core and integrated system so that a user may examine registers, memory, or on-chip peripherals. OnCE operation is controlled through an

industry-standard IEEE 1149.1 TAP controller. By using JTAG 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 (using the same set of resources available to software while the e200z6 is in internal debug mode), and is present in all e200z6-based designs. The OnCE module also provides support for directly integrating a Nexus class 2 or class 3 real-time debug unit with the e200z6 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 debugging allows for capability and cost tradeoffs to be made.

The e200z6 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 one is present in the system. Thus, the e200z6 module 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 through the TAP and appropriate entry of serial data. Refer to the documentation for the integrating device. The *jd_enable_once* input should not change state during a debug session, or undefined activity may occur.

Figure 10-2 shows the TAP controller and TAP registers implemented by the OnCE logic.

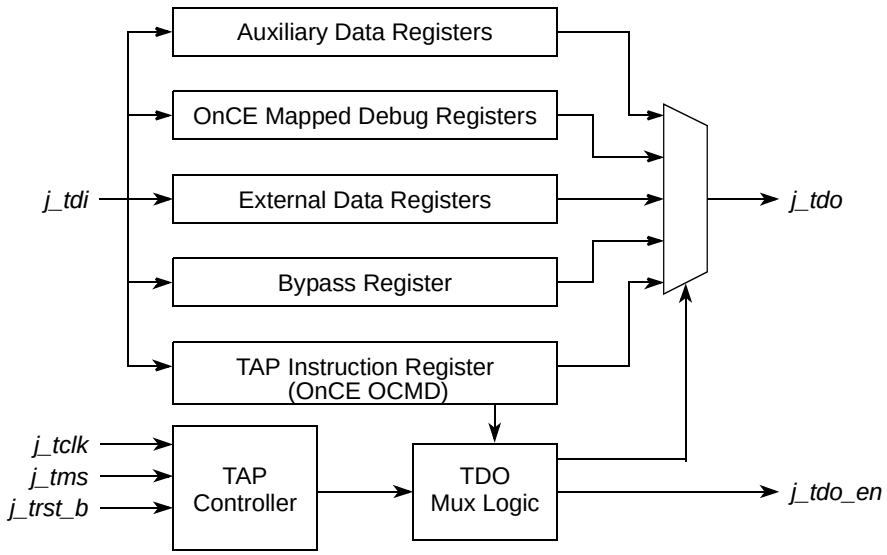


Figure 10-2. OnCE TAP Controller and Registers

The OnCE controller is implemented as a 16-state finite state machine (FSM), shown in Figure 10-3, with a one-to-one correspondence to the states defined for the JTAG TAP controller.

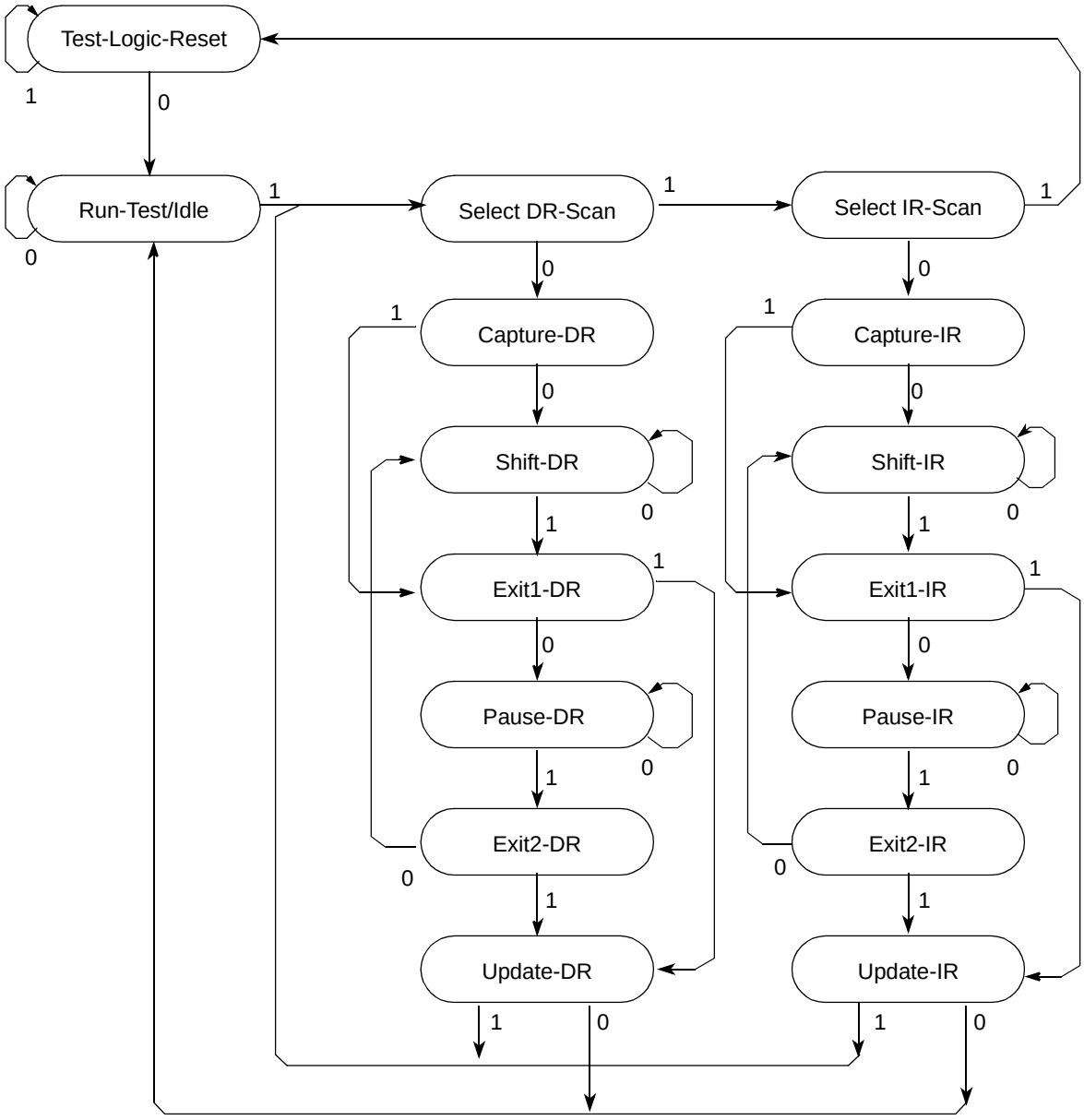


Figure 10-3. OnCE Controller as an FSM

Access to e200z6 processor registers and the contents of memory locations is performed by enabling external debug mode (setting DBCR0[EDM]), placing the processor into debug mode, and scanning instructions and data into and out of the e200z6 CPU scan chain (CPUSCR); execution of scanned instructions by the e200z6 is used as the method for accessing required data. Memory locations may be read by scanning a load instruction into the e200z6 core that 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.

The initial entry by the CPU into the debug state (or mode) from normal, stopped, halted, or checkstop states (all indicated by the OnCE status register (OSR) described in Section 10.5.5.1, “e200z6 OnCE Status Register (OSR)”) by assertion of one or more debug requests begins a debug session. The *jd_debug_b* output signal indicates that a debug session is in progress, and the OSR indicates that the CPU is in the debug state. Instructions may be single-stepped by scanning new values into the CPUSCR and performing a OnCE Go+NoExit command (See Section 10.5.5.2, “e200z6 OnCE Command Register (OCMD).”) The CPU then temporarily exits the debug state (but not the debug session) to execute the instruction and returns to the debug state (again indicated by the OSR). The debug session remains in force until the final 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 before executing each Go+Exit or Go+NoExit command.

10.5.2 JTAG/OnCE Signals

The JTAG/OnCE interface is used to transfer OnCE instructions and data to the OnCE control block. Depending on the resource being accessed, the CPU may need to be placed in debug mode. For resources outside the CPU block and contained in the OnCE block, the processor is not disturbed and may continue execution. If a processor resource is required, an internal debug request (*dbg_dbgrq*) may be asserted to the CPU by the OnCE controller, and 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 setting MSR[WE].

Table 10-3 details the primary JTAG/OnCE interface signals.

Table 10-3. JTAG/OnCE Primary Interface Signals

Signal Name	I/O	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	O	Enables TDO output buffer Set when the TAP controller is in the Shift-DR or Shift-IR state.

A full description of JTAG signals is provided in Section 8.3.2, “JTAG ID Signals.”

10.5.3 OnCE Internal Interface Signals

The following sections describe the e200z6 OnCE interface signals to other internal blocks associated with the e200z6 OnCE controller. Table 10-4 shows the OnCE internal interface signals.

Table 10-4. OnCE Internal Interface Signals

Signal Name	I/O	Description
CPU debug request (*dbg_dbgrq*)	O	The db*g_dbgrq* signal is set by the e200z6 OnCE control logic to request the CPU to enter the debug state. It may be set for a number of different conditions, and causes the CPU to finish the current instruction being executed, save the instruction pipeline information, enter debug mode, and wait for further commands.
CPU debug acknowledge (*cpu_dbgack*)	I	The *cpu_dbgack* signal is set by the CPU upon entering the debug state. This signal is used as part of the handshake mechanism between the e200z6 OnCE control logic and the rest of the CPU. The CPU core may enter debug mode through either a software or hardware event.

10.5.3.1 CPU Address and Attributes

The CPU address and attribute information are used by an external Nexus class 2–4 debug unit with information for real-time address trace information.

10.5.3.2 CPU Data

The CPU data bus is used to supply an external Nexus class 2–4 debug unit with information for real-time data trace capability.

10.5.4 OnCE Interface Signals

The following sections describe additional e200z6 OnCE interface signals to other external blocks such as a Nexus controller and external blocks that may need information pertaining to debug operation.

Table 10-5 describes the OnCE interface signals.

Table 10-5. OnCE Interface Signals [1]

Signal Name	I/O	Description
OnCE enable (*jd_en_once)*	I	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 e200z6 OnCE unit, and all other commands default to the Bypass command. The OSR is not visible when OnCE operation is disabled. Also OCR functions are also disabled, as is the operation of the *jd_de_b* input. Secure systems may choose to leave *jd_en_once* negated until a security check has been performed. Other systems should tie this signal asserted to enable full OnCE operation. The *j_en_once_regsel* output signal is provided to assist external logic performing security checks. Refer to Section 10.5.5.3, “e200z6 OnCE Control Register (OCR),” for a description of the *j_en_once_regsel* output. The *jd_en_once* input must change state only during the test-logic-reset, Run-Test/Idle, or Update-DR TAP states. A new value takes effect after one additional *j_tclk* cycle of synchronization. In addition, *jd_enable_once* must not change state during a debug session, or undefined activity may occur.
OnCE debug request (*jd_de_b)*/event *(jd_de_en*)	I/O	The system-level bidirectional open drain debug event pin, *DE_b*, (not part of the e200z6 interface described in Chapter 8, “External Core Complex Interfaces”) provides a fast means of entering the debug mode of operation from an external command controller (when input) as well as a fast means of acknowledging entry into debug mode of operation to an external command controller (when output). The assertion of this pin by a command controller causes the CPU core to finish the current instruction being executed, save the instruction pipeline information, enter debug mode, and wait for commands to be entered. If *DE_b* was used to enter debug mode, *DE_b* must be negated after the OnCE controller responds with an acknowledge and before sending the first OnCE command. The assertion of this pin by the CPU core acknowledges that it has entered the debug mode and is waiting for commands to be entered. To support operation of this system pin, the OnCE logic supplies the *jd_de_en* output and samples the *jd_de_b* input when OnCE is enabled (*jd_en_once* set). 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.
e200z6 OnCE debug output (jd_debug_b)	O	The e200z6 OnCE debug output *jd_debug_b* is used to indicate to on-chip resources that a debug session is in progress. Peripherals and other units may use this signal to modify normal operation for the duration of a debug session, which may involve the CPU executing a sequence of instructions solely for the purpose of visibility/system control that are not part of the normal instruction stream the CPU would have executed had it not been placed in debug mode. This signal is set the first time the CPU enters the debug state, and remains set until the CPU is released by a write to the e200z6 OnCE command register (OCMD) with the GO and EX bits set, and a register specified as either no register selected or the CPUSCR. This signal remains set even though the CPU may enter and exit the debug state for each instruction executed under control of the e200z6 OnCE controller. See Section 10.5.5.2, “e200z6 OnCE Command Register (OCMD),” for more information on the function of the GO and EX bits. This signal is not normally used by the CPU.

Table 10-5. OnCE Interface Signals [1]

Signal Name	I/O	Description
e200z6 CPU clock on input (jd_mclk_on)	I	The e200z6 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 e200z6 core, 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 proper scan sequences occur to access debug resources in the *m_clk* clock domain.
Watchpoint events (*jd_watchpt[0:7]*)	O	The *jd_watchpt[0:7]* signals may be set by the e200z6 OnCE control logic to signal that a watchpoint condition has occurred. Watchpoints do not cause the CPU to be affected. They are provided to allow external visibility only. Watchpoint events are conditioned by the settings in DBCR0, DBCR1, and DBCR2.

1 These are high–level descriptions of the OnCE interface signals, more detailed descriptions can be found in Section 8.3, "Signal Descriptions."

10.5.5 e200z6 OnCE Controller and Serial Interface

The e200z6 OnCE controller contains the e200z6 OnCE command register, the e200z6 OnCE decoder, and the status/control register. Figure 10-4 is a block diagram of the e200z6 OnCE controller. In operation, the e200z6 OnCE command register acts as the instruction register (IR) for the e200z6 TAP controller, and all other OnCE resources are treated as data registers (DR) by the TAP controller. The command register is loaded by serially shifting in commands during the TAP controller Shift-IR state, and is loaded during the Update-IR state. The command register selects a resource to be accessed as a data register (DR) during the TAP controller Capture-DR, Shift-DR, and Update-DR states.

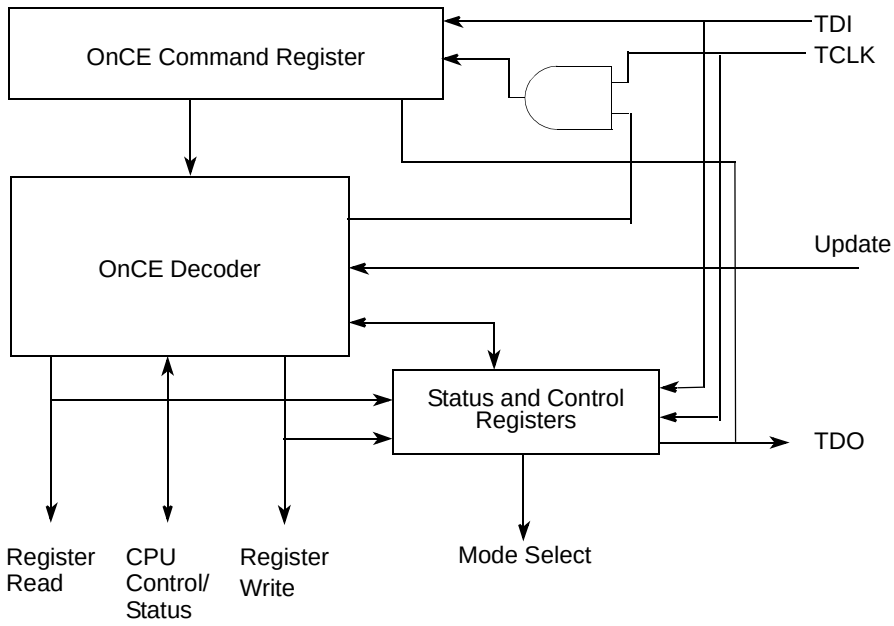


Figure 10-4. e200z6 OnCE Controller and Serial Interface

10.5.5.1 e200z6 OnCE Status Register (OSR)

Status information regarding the state of the e200z6 CPU is latched into the OSR when the OnCE controller state machine enters the Capture-IR state. When OnCE operation is enabled, this information is provided on the *j_tdo* output in serial fashion when the Shift-IR state is entered following a Capture-IR. Information is shifted out least-significant bit first.

	0	1	2	3	4	5	6	7	8	9
Field	MCLK	ERR	CHKSTOP	RESET	HALT	STOP	DEBUG	0		1

Figure 10-5. OnCE Status Register (OSR)

Table 10-6 describes OnCE status register bits.

Table 10-6. OSR Field Descriptions

Bits	Name	Description
0	MCLK	*m_clk* status bit. Reflects the logic level on the *jd_mclk_on* input signal after capture by *j_tclk*. 0 Inactive state 1 Active state
1	ERR	Error. 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, and so on) occurred while attempting to perform the instruction single-step. In this case, the CPUSCR contains information related to the first instruction of the interrupt handler, and no portion of the handler will have been executed.
2	CHKSTOP	Checkstop mode. Reflects the logic level on the CPU *p_chkstop* output after capture by *j_tclk*.
3	RESET	Reset mode. Reflects the inverted logic level on the CPU *p_reset_b* input after capture by *j_tclk*.
4	HALT	Halt mode. Reflects the logic level on the CPU *p_halted* output after capture by *j_tclk*.
5	STOP	Stop mode. Reflects the logic level on the CPU *p_stopped* output after capture by *j_tclk*.
6	DEBUG	Debug mode. Set once the CPU is in debug mode. It is negated once the CPU exits debug mode (even during a debug session).
7	—	Reserved, set to 0
8	—	Reserved, set to 0 for 1149.1 compliance
9	—	Reserved, set to 1 for 1149.1 compliance

10.5.5.2 e200z6 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 e200z6 OnCE decoder. OCMD is shown in Figure 10-5. It 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 OCMD is updated during the Update-IR TAP controller state, the corresponding resource is accessed in the DR scan sequence of the TAP controller, and as such, the Update-DR state must be transitioned through 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.

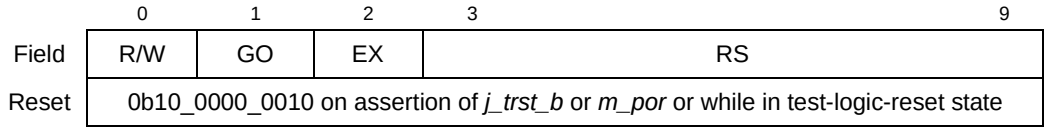


Figure 10-6. OnCE Command Register (OCMD)

Table 10-7 describes OCMD fields.

Table 10-7. OCMD Field Descriptions

Bits	Name	Description
0	R/W	Read/Write. Specifies the direction of data transfer. 0 Write the data associated with the command into the register specified by RS 1 Read the data contained in the register specified by RS Note: The R/W bit is generally ignored for read-only or write-only registers, although the PC FIFO pointer is only guaranteed to be updated 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 0 Inactive (no action taken) 1 Execute instruction in IR If the GO bit is set, the chip executes the instruction which resides in the IR register in the CPUSCR. To execute the instruction, the processor leaves debug mode, executes the instruction, and if the EX bit is cleared, returns to 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 set. 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 leaves debug mode after the TAP controller Update-DR state is entered. On a Go+NoExit operation, returning to debug mode is treated as a debug event; thus, exceptions such as machine checks and interrupts may take priority and prevent execution of the intended instruction. Debug firmware should mask these exceptions as appropriate. OSR[ERR] indicates such an occurrence.

Table 10-7. OCMD Field Descriptions (continued)

Bits	Name	Description
2	EX	Exit 0 Remain in debug mode 1 Leave debug mode If the EX bit is set, the processor leaves debug mode and resumes 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 leaves debug mode after the TAP controller Update-DR state is entered. Note that if the DR bit in the OnCE control register is set or remains set, or if a bit in the DBSR is set, or if a bit in the DBSR is set and DBCR0[EDM] = 1 (external debug mode is enabled), then the processor may return to the debug mode without execution of an instruction, even though the EX bit was set.
3–9	RS	Register select. Defines which register is the source for the read or the destination for the write operation. Table 10-9 indicates the e200z6 OnCE register addresses. Attempted writes to read-only registers are ignored. 000 0000–000 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 0011–001 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 Reserved DVC1 future use 010 0111 Reserved DVC2 future use 010 1000–010 1011 Reserved 010 1100 Debug counter register DBCNT 010 1101 Debug PCFIFO (PCFIFO) read–only 010 1110–010 1111 Reserved 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 0101–101 1111 Reserved (do not access) 111 0000–111 1001 General purpose register selects [0–9] 111 1010 CDACNTL–See Section 4.19, “Cache Memory Access during Debug” 111 1011 CDADATA–See Section 4.19, “Cache Memory Access during Debug” 111 1100 Nexus3–Access–See Chapter 11, “Nexus3 Module” 111 1101 Reserved 111 1110 Enable_OnCE [1]

[1] Causes assertion of the *j_en_once*_regsel output. Refer to Section 10.5.5.3, “e200z6 OnCE Control Register (OCR).”

The OnCE decoder receives as input the 10-bit command from the OCMD and the status signals from the processor, and generates all the strobes required for reading and writing the selected OnCE registers.

Single-stepping of instructions is performed by placing the CPU in debug mode, scanning appropriate information into the CPUSCR, and setting the GO bit (with the EX bit cleared) with the RS field indicating either the CPUSCR or no register selected. After executing a single instruction, the CPU re-enters debug mode and awaits further commands. During single-stepping, exception conditions may occur if not properly masked by debug firmware (interrupts, machine checks, bus error conditions, and so on) and may prevent the desired instruction from being successfully executed. The OSR[ERR] bit is set to indicate this condition. In these cases, values in the CPUSCR correspond to the first instruction of the exception handler.

Additionally, while single-stepping, to prevent debug events from generating debug interrupts, DBCR0[EDM] is internally forced to 1. Also, during a debug session, DBSR and DBCNT are frozen from updates due to debug events regardless of DBCR0[EDM]. They may still be modified during a debug session through a single-stepped **mtspr** instruction if DBCR0[EDM] is cleared, or through OnCE access if DBCR0[EDM] is set.

10.5.5.3 e200z6 OnCE Control Register (OCR)

The e200z6 OnCE control register (OCR) forces the e200z6 core into debug mode and enables/disables sections of the e200z6 OnCE control logic. It also provides control over the MMU during a debug session. (See Section 10.7, “MMU and Cache Operation during Debug.”) The control bits are read/write. These bits are effective only while OnCE is enabled (*jd_en_once* set). The OCR is shown in Figure 10-7.

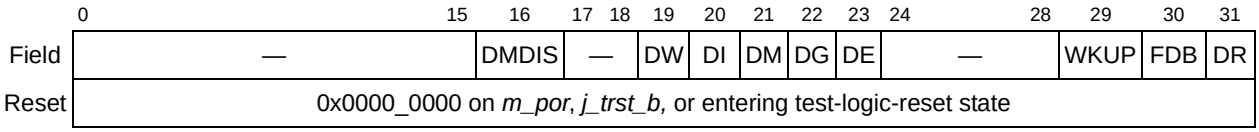


Figure 10-7. OnCE Control Register

Table 10-8 provides bit definitions for the OnCE control register.

Table 10-8. OnCE Control Register Bit Definitions

Bits	Name	Description
0–15	—	Reserved, should be cleared.
16	DMDIS	Debug MMU disable control bit 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. When enabled, the MMU functions normally. When disabled, no address translation is performed (1:1 address mapping), and the TLB WIMGE bits are taken from the OCR bits DW, DI, DM, DG, and DE. The SX, SR, SW, UX, UR, and UW access permission control bits are set to1 to allow full access. When disabled, no TLB miss or TLB exceptions are generated. External access errors can still occur.
17–18	—	Reserved

Table 10-8. OnCE Control Register Bit Definitions (continued)

Bits	Name	Description
19	DW	Debug TLB W attribute bit. Provides the W attribute bit to be used when the MMU is disabled during a debug session.
20	DI	Debug TLB I attribute bit. Provides the I attribute bit to be used when the MMU is disabled during a debug session.
21	DM	Debug TLB M attribute bit. Provides the M attribute bit to be used when the MMU is disabled during a debug session.
22	DG	Debug TLB G attribute bit. Provides the G attribute bit to be used when the MMU is disabled during a debug session.
23	DE	Debug TLB E attribute bit. Provides the E attribute bit to be used when the MMU is disabled during a debug session.
24–28	—	Reserved
29	WKUP	Wakeup request bit. Forces the e200z6 *p_wakeup* output signal to be set. This control function may be used by debug firmware to request that the chip-level clock controller restore the *m_clk* input to normal operation regardless of whether the CPU is in a low-power state to ensure that debug resources may be properly accessed by external hardware through scan sequences.
30	FDB	Force breakpoint debug mode bit. Determines whether the processor is operating in breakpoint debug enable mode. 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 causes the processor to enter debug mode as if the *jd_de_b* input had been set. **Note:** This bit is qualified with DBCR0[EDM], which must be set for FDB to take effect.
31	DR	CPU debug request control bit. Unconditionally requests the CPU to enter debug mode. The CPU indicates that debug mode has been entered through the data scanned out in the Shift-IR state. 0 No debug mode request 1 Unconditional debug mode request When the DR bit is set the processor enters debug mode at the next instruction boundary.

10.5.6 Access to Debug Resources

Resources contained in the e200z6 OnCE module that do not require the e200z6 processor core to be halted for access may be accessed while the e200z6 core is running, and will not interfere with processor execution. Accesses to other resources such as the CPUSCR require the e200z6 core to be placed in debug mode to avoid synchronization hazards. Debug firmware may ensure that it is safe to access these resources by determining the state of the e200z6 core before access.

NOTE

A scan operation to update the CPUSCR is required before exiting debug mode.

Some cases of write accesses other than accesses to the OnCE command and control registers or DBCR0[EDM] require the e200z6 *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, because the CPU may cause multiple bits of certain registers to change state, reads of certain registers while the CPU is running (for example, DBSR and DBCNT) may not have consistent bit settings unless read twice with the same value indicated. 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 10-9 lists access requirements for OnCE registers.

Table 10-9. OnCE Register Access Requirements

Register Name	Access Requirements					Notes
	***jd_en_once* to be Set**	**DBCR0 [EDM] = 1**	***m_clk* active for Write Access**	**CPU to be Halted for Read Access**	**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	*[1]	Reads of DBCNT while the CPU is running may not give data that is self-consistent due to synchronization across clock domains.
DBCR0	Y	Y	Y	N	*[1]	*DBCR0[EDM] access only requires *jd_en_once* set
DBCR1	Y	Y	Y	N	*[1]	
DBCR2	Y	Y	Y	N	*[1]	
DBCR3	Y	Y	Y	N	*[1]	
DBSR	Y	Y	Y	N	*[1]	Reads of DBSR while the CPU is running may not give data that is self-consistent due to synchronization across clock domains.
IAC1	Y	Y	Y	N	*[1]	
IAC2	Y	Y	Y	N	*[1]	
IAC3	Y	Y	Y	N	*[1]	
IAC4	Y	Y	Y	N	*[1]	
JTAG ID	N	N	—	N	—	Read only
OCR	Y	N	N	N	N	
OSR	Y	N	—	N	—	Read only, accessed by scanning out IR while *jd_en_once* is set

Table 10-9. OnCE Register Access Requirements (continued)

Register Name	Access Requirements					Notes
	jd_en_once to be Set	DBCR0 [EDM] = 1	*m_clk* active for Write Access	CPU to be Halted for Read Access	CPU to be Halted for Write Access	
PC FIFO	Y	N	—	N	—	Read only, updates frozen while OCMD holds PCFIFO register encoding **Note:** No updates occur to the PCFIFO while the OnCE state machine is in the Test_Logic_Reset state
Cache Debug Access Control (CDACNTL)	Y	N	Y	Y	Y	CPU must be in debug mode with clocks running
Cache Debug Access Data (CDADATA)	Y	N	Y	Y	Y	CPU must be in debug mode with clocks running
Nexus3-Access	Y	N	N	N	N	
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 the 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.

10.5.7 Methods for Entering Debug Mode

The OSR indicates that the CPU has entered the debug mode through the debug status bit. The following sections describe how e200z6 debug mode is entered assuming the OnCE circuitry has been enabled. e200z6 OnCE operation is enabled by the assertion of the *jd_en_once* input (see Table 10-2).

Table 10-10 describes the methods for entering debug mode.

Table 10-10. Methods for Entering Debug Mode

Method Name	Description
External debug request during reset	Holding *jd_de_b* asserted during the assertion of *p_reset_b* and continuing to hold it asserted following the negation of *p_reset_b* causes the e200z6 core to enter debug mode. After receiving an acknowledge through the OnCE status register debug bit, the external command controller should negate the *jd_de_b* signal before sending the first command. Note that in this case the e200z6 core does not execute an instruction before entering debug mode, although the first instruction to be executed may be fetched 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 debug mode is exited; thus, the debug controller must initialize PC, MSR, and IR to the image that the processor would have obtained in performing reset exception processing or must cause the appropriate bit reset to be re-asserted.
Debug request during reset	Asserting a debug request by setting the OCR[DR] 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 debug mode is exited; thus, the debug controller must initialize PC, MSR, and IR to the image that the processor would have obtained in performing reset exception processing or must cause the appropriate reset to be re-asserted.
Debug request during normal activity	Asserting a debug request by setting OCR[DR] during normal chip activity causes the chip to finish execution of the current instruction and then enter debug mode. Note that in this case the chip completes 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 are aborted by the interrupt processing.
Debug request during halted or stopped state	Asserting a debug request by setting OCR[DR] when the chip is in the halted state (*p_halted* set) or stopped state (*p_stopped* set) causes the CPU to exit the state and enter debug mode once the CPU clock *m_clk* has been restored. Note that in this case, the CPU negates both the *p_halted* and *p_stopped* outputs. Once the debug session has ended, the CPU returns to the state it was in before entering debug mode. To signal the chip-level clock generator to re-enable *m_clk*, the *p_wakeup* output is set whenever the debug block is asserting a debug request to the CPU due to OCR[DR] being set, or *jd_de_b* assertion, and remains set from then until the debug session ends (*jd_debug_b* goes from set 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.
Software request during normal activity	Upon executing a *'bkpt'* pseudo-instruction (for the e200z6, defined to be an all zeros instruction opcode), when OCR [FDB] 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.

10.5.8 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 e200z6 OnCE controller. CPUSCR contains these processor resources, which are used to restore the pipeline and resume normal chip activity upon return from debug mode, as well as a mechanism for the emulator software to access processor and memory contents. Figure 10-8 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 before exiting debug mode.

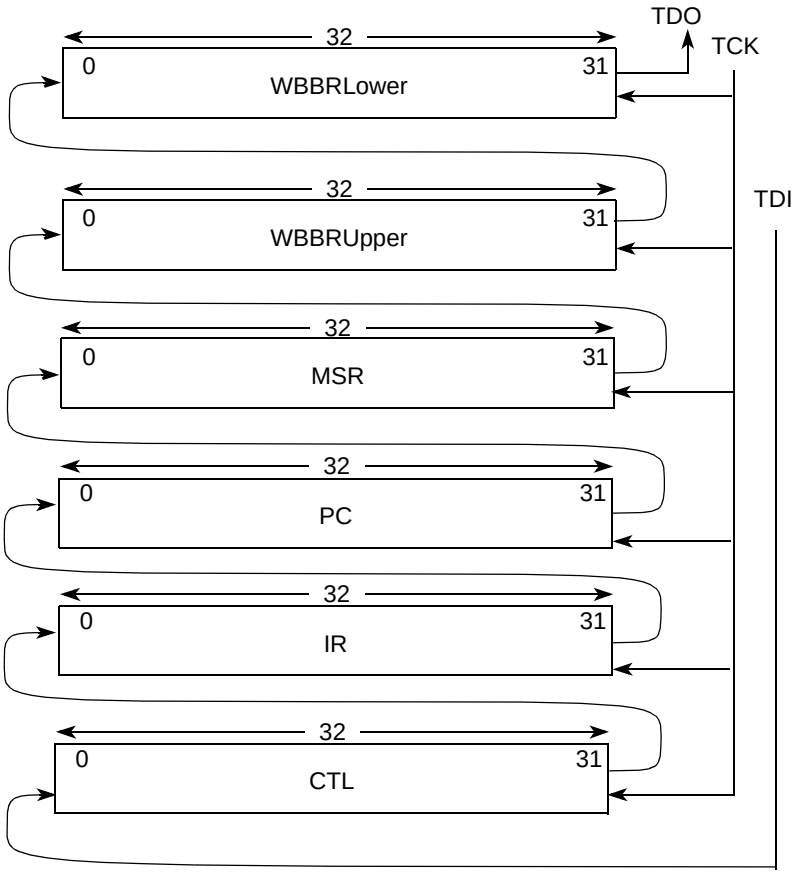


Figure 10-8. CPU Scan Chain Register (CPUSCR)

10.5.8.1 Instruction Register (IR)

The instruction register (IR) provides a way to control the debug session by serving as a means for forcing in selected instructions and causing them to be executed in a controlled manner by the debug control block. The opcode of the next instruction to be executed when entering debug mode is contained in this register when the scan-out of this chain begins. This value should be saved for later restoration if continuation of the normal instruction stream is desired.

On scan-in, in preparation for exiting debug mode, this register is filled with an instruction opcode selected by debug control software. By selecting appropriate instructions and controlling the execution of those instructions, the results of execution may be used to examine or change memory locations and processor registers. The debug control module external to the processor core controls execution by providing a single-step capability. Once the debug session is complete and normal processing is to be resumed, this register may be loaded with the value originally scanned out.

10.5.8.2 Control State Register (CTL)

The control state register (CTL), shown in Figure 10-9, stores the value of certain internal CPU state variables before debug mode is entered. This register is affected by the operations performed during the debug session and should normally be restored by the external command controller when returning to normal mode. In addition to saved internal state variables, two of the bits are used by emulation firmware to control the debug process. In certain circumstances, emulation firmware must modify the content of this register as well as the PC and IR values in the CPUSCR before exiting debug mode. These cases are described more specifically in the text after the table.

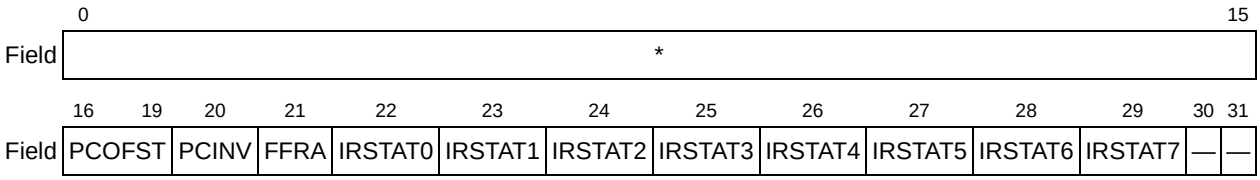


Figure 10-9. Control State Register (CTL)

Table 10-11. CTL Field Definitions

Bits	Name	Description
0–15	*	Internal state bits.These control bits represent internal processor state and should be restored to their original value after a debug session is completed, that is, when an e200z6 OnCE command is issued with the GO and EX bits set and not ignored. When performing instruction execution during a debug session (see Section 10.2.1, “Software Debug Facilities)”, these bits should be cleared.
16–19	PCOFST	PC offset field. Indicates whether the value in the PC portion of the CPUSCR must be adjusted before 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 before 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.

Table 10-11. CTL Field Definitions (continued)

Bits	Name	Description
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 before 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 $WBBR_{lower}$ to be used as the **r**A (**r**S for logical and shift operations) operand value of the first instruction to be executed following an update of the CPUSCR. This allows the debug firmware to update processor registers, initialize the $WBBR_{lower}$ 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 **r**A (**r**S for logical and shift operations) operand value
22	IRSTAT0	IR status bit 0.This control bit indicates an ERROR termination status for the IR. 0 No TEA occurred on the fetch of this instruction. 1 A TEA occurred on the fetch of this instruction
23	IRSTAT1	IR status bit 1. Indicates a TLB miss status for the IR. 0 No TLB miss occurred on the fetch of this instruction. 1 TLB miss occurred on the fetch of this instruction.
24	IRSTAT2	IR status bit 2. 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. 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. 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. 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 A parity error occurred on the fetch of this instruction.
29	IRSTAT7	IR status bit 7. Indicates a precise external termination error status for the IR. 0 No precise external termination error occurred on the fetch of this instruction. 1 Precise external termination error occurred on the fetch of this instruction.

Emulation firmware should modify the CTL, PC, and IR values in the CPUSCR during execution of debug-related instructions as well as just before exiting debug with a Go+Exit command. During the debug session, the CTL register should be written with the FFRA bit set as appropriate and with all other bits cleared, and IR set to the value of the desired instruction to be executed.

The PCINV status bit which was originally present when debug mode was first entered should be tested before exiting debug mode with a Go+Exit and if set, the PC and IR initialized for performing whatever recovery sequence is appropriate for a faulted exception vector fetch. If the PCINV bit is cleared, the PCOFST bits should be examined to determine whether the PC value must be adjusted. Due to the pipelined nature of the CPU, the PC value must be backed up by emulation software in certain circumstances. The PCOFST field specifies the value to be subtracted from the original value of the PC. This adjusted PC value should be restored into the PC portion of the CPUSCR just before exiting debug mode with a Go+Exit. In the event that PCOFST is non-zero, the IR should be loaded with a nop instruction (such as **ori r0,r0,0**) instead of the original IR value; otherwise, the original value of IR should be restored. Note that when a correction is made to the PC value, it generally points to the last completed instruction, although that instruction will not be re-executed. The nop instruction is executed instead, and instruction fetch and execution resumes at location PC+4.

For CTL, the internal state bits should be restored to their original value. The IRStatus bits should be cleared if the PC was adjusted. If no PC adjustment was performed, emulation firmware should determine whether IRStat2–5 should be cleared 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 before any further instruction execution.

10.5.8.3 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 debug mode. It is affected by the operations performed during 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 nop and the PC set to point to the location before the location at which it is desired to redirect flow to. On exiting debug mode the nop is executed, and instruction fetch and execution resumes at PC+4.

10.5.8.4 Write-Back Bus Register (WBBR (lower) and WBBR (upper))

WBBR provides a way to pass 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. $WBBR_{lower}$ holds the 32-bit result of most instructions including load data returned for a load or load with update instruction. For SPE instructions that generate 64-bit results, $WBBR_{lower}$ holds the low-order 32 bits of the result. $WBBR_{upper}$holds the updated effective address calculated by a load with update

instruction. For SPE instructions that generate 64-bit results, $WBBR_{upper}$ holds the high-order 32 bits of the result. It is undefined for other instructions.

As an example, to read the lower 32 bits of processor register r1, an **ori r1,r1,0** instruction is executed, and the result value of the instruction is latched into $WBBR_{lower}$. The contents of $WBBR_{lower}$ 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 is executed that uses this value as a substitute data value. The control state register FFRA bit forces the value of the $WBBR_{lower}$ 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 10.5.8.2, “Control State Register (CTL),” for more details.).

$WBBR_{lower}$ and $WBBR_{upper}$ are generally undefined on instructions that do not writeback a result, and due to control issues are not defined on **lmw** or branch instructions as well.

To read and write the entire 64 bits of a GPR, both $WBBR_{lower}$ and $WBBR_{upper}$ are used. For reads, an **evslwi r_n,r_n,0** may be used. For writes, the same instruction may be used, but the CTL[FFRA] bit must be set as well.

NOTE

MSR[SPE] must be set in order for these operations to be performed properly.

10.5.8.5 Machine State Register (MSR)

The MSR is a 32-bit register used to read/write the machine state register (MSR). Whenever the external command controller needs to save or modify the contents of the machine state register, this register is used.This register is affected by the operations performed during debug mode and must be restored by the external command controller when returning to normal mode. Chapter 2, “Register Model,” further describes the MSR.

10.5.9 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.

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 read 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 10-10 shows the block diagram of the PC FIFO.

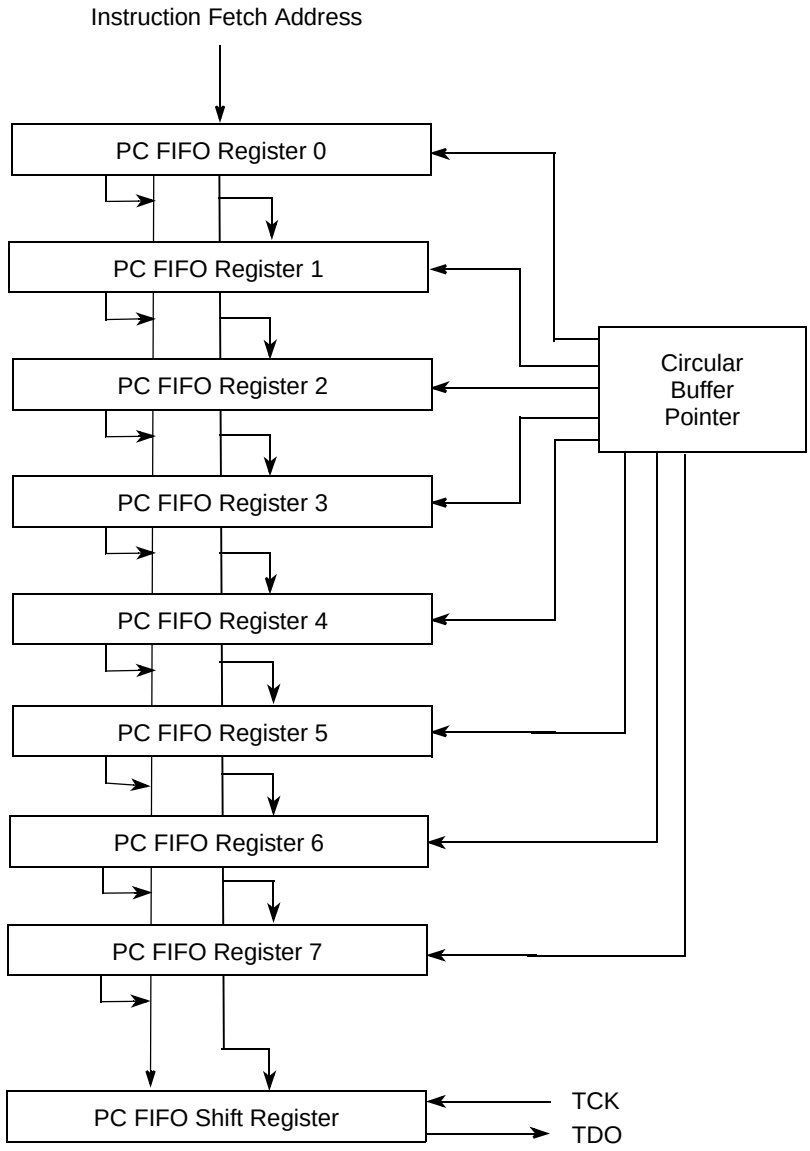


Figure 10-10. OnCE PC FIFO

The FIFO is not affected by the operations performed during a debug session except for the FIFO pointer increment when reading the FIFO. When entering debug mode, the FIFO counter is pointing to the FIFO register containing the address of the oldest of the eight change of flow prefetches. When OCMD [RS] 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 are performed. The first FIFO read obtains the oldest address and the following FIFO read returns the other addresses from the oldest to the newest (the order of execution).

Updates to the FIFO are frozen whenever the OCMD register contains a command whose RS[0–6] field points to the PC FIFO (010 1101) to allow firmware to read 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 reads of the FIFO should be performed since each read increments the temporary FIFO pointer, thus making it point to the next location. After eight reads the pointer points to the same location it pointed to before starting the read 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* is set.

10.5.10 Reserved Registers

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.

10.6 Watchpoint Support

The e200z6 supports the generation and signalling of watchpoints when operating in internal debug mode (DBCR0[IDM] = 1) or in external debug mode (DBCR0[EDM] = 1). Watchpoints are indicated with a dedicated set of interface signals. The *jd_watchpoint[0:7]* output signals are used to indicate that a watchpoint has occurred.

Each debug address compare function (IAC1–IAC4, DAC1 and DAC2) and debug counter event (DCNT1 and DCNT2) can trigger a watchpoint output. The DBCR1, DBCR2, and DBCR3 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 (except during a debug session). If not desired, the base address values for these events may be programmed to an unused system address. MSR[DE] has no effect on watchpoint generation.

External logic may monitor the assertion of these signals for debugging purposes. Watchpoints are signaled in the clock cycle following the occurrence of the actual event. The Nexus3 module also monitors assertion of these signals for various development control purposes (see Section 11.9, “Watchpoint Support)”.

Table 10-12. Watchpoint Output Signal Assignments

Signal Name	Type	Description
jd_watchpt[0]	IAC1	Instruction address compare 1 watchpoint Set whenever an IAC1 compare occurs regardless of whether IAC1 compares are enabled to set DBSR status.
jd_watchpt[1]	IAC2	Instruction address compare 2 watchpoint Set whenever an IAC2 compare occurs regardless of whether IAC2 compares are enabled to set DBSR status.
jd_watchpt[2]	IAC3	Instruction address compare 3 watchpoint Set whenever an IAC3 compare occurs regardless of whether IAC3 compares are enabled to set DBSR status.
jd_watchpt[3]	IAC4	Instruction address compare 4 watchpoint Set whenever an IAC4 compare occurs regardless of whether IAC4 compares are enabled to set DBSR status.
jd_watchpt[4]	DAC1 [1]	Data address compare 1 watchpoint Set whenever a DAC1 compare occurs regardless of whether DAC1 compares are enabled to set DBSR status.
jd_watchpt[5]	DAC2[1]	Data address compare 2 watchpoint Set whenever a DAC2 compare occurs regardless of whether DAC2 compares are enabled to set DBSR status.
j*d_watchpt[6]*	DCNT1	Debug counter 1 watchpoint Set whenever debug counter 1 decrements to zero regardless of whether DCNT1 compares are enabled to set DBSR status.
jd_watchpt[7]	DCNT2	Debug counter 2 watchpoint Set whenever debug counter 2 decrements to zero regardless of whether DCNT2 compares are enabled to set DBSR status.

[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.

10.7 MMU and Cache Operation during Debug

Normal operation of the MMU may be modified during a debug session using the OnCE 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 may substitute default values for the access protection (UX, UR, UW, SX, SR, SW) bits, and values obtained from the OnCE control register and page attribute (W, I, M, G, E) bits normally provided by a matching TLB entry. In addition, no address translation is performed, and instead, a 1:1 mapping of effective-to-real addresses is performed.

When disabled during a debug session, TLB miss or TLB-related DSI conditions cannot 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) remains in effect.

The OCRDMDIS, DW, DI, DM, DG, and DE control bits are used when debug mode is entered. Refer to the bit definitions in the OCR (See Section 10.5.5.3, “e200z6 OnCE Control Register (OCR),” for more detail). These substituted page attribute bits control cache operation on accesses initiated during debug. No address translation is performed; instead, a 1:1 mapping between effective and real addresses is performed.

10.8 Cache Array Access During Debug

The cache arrays may be read and written during debug mode through the CDACNTL and CDADATA debug registers. This functionality is described in detail in Section 4.19, “Cache Memory Access during Debug.”

10.9 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.

Enable external debug mode and initialize debug registers:

1. The debugger should ensure that the *jd_en_once* control signal is set in order to enable OnCE operation.
2. Select the OCR and write a value to it in which OCR[DR] and OCR[WKUP] are set. The TAP controller must step through the proper states as outlined earlier. This step places the CPU in a debug state where 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 DBCR0[EDM] set.
5. Clear the DBSR status bits.
6. Write appropriate values to the DBCR0–DBCR3, IAC, DAC, and DBCNT registers.

NOTE

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.

1. Optionally set the OCR[DMDIS] control bit to ensure that no TLB misses occur while performing the debug operations.
2. Optionally ensure that the values entered into the MSR portion of the CPUSCR during the following steps cause interrupts 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. The 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 10.5.8.2, “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. OCR[DMDIS] and OCR[DR] should be cleared, leaving OCR[WKUP] 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 10.5.8.2, “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 through the OCR[DR] with OCR[WKUP] set, scanning out and saving the CPUSCR.
2. The debugger should write to DBCR0–DBCR3 as needed, likely clearing every enable except DBCR0[EDM].
3. The debugger should write the DBSR to a cleared state.
4. The debugger should rewrite 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 10.5.8.2, “Control State Register (CTL))”,with a Go+Exit OnCE command value.
7. OCR[WKUP] may then be cleared.

NOTE

These steps are meant by way of examples, and are not meant to be an exact template for debugger operation.

Chapter 11
Nexus3 Module

The e200z6 Nexus3 module provides real-time development capabilities for e200z6 processors in compliance with the *IEEE-ISTO Nexus 5001-2003* 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/Nexus1 unit. The *IEEE-ISTO 5001-2003* standard defines an extensible auxiliary port which is used in conjunction with the JTAG port in e200z6 processors.

11.1 Introduction

11.1.1 General Description

This chapter defines the auxiliary pin functions, transfer protocols and standard development features of a class 3 device in compliance with the *IEEE-ISTO Nexus 5001-2003* standard. The development features supported are program trace, data trace, watchpoint messaging, ownership trace, and read/write access through the JTAG interface. The Nexus3 module also supports two class 4 features: watchpoint triggering, and processor overrun control.

11.1.2 Terms and Definitions

Table 11-1 contains a set of terms and definitions associated with the Nexus3 module.

Table 11-1. Terms and Definitions

Term	Description
IEEE-ISTO 5001	Consortium and standard for real-time embedded system design. World Wide Web documentation at http://www.ieee-isto.org/Nexus5001
Auxiliary port	Refers to Nexus auxiliary port. Used as auxiliary port to the IEEE 1149.1 JTAG interface.
Branch trace messaging (BTM)	Visibility of addresses for taken branches and exceptions, and the number of sequential instructions executed between each taken branch.
Data read message (DRM)	External visibility of data reads to memory-mapped resources.

Table 11-1. Terms and Definitions (continued)

Term	Description
Data write message (DWM)	External visibility of data writes to memory-mapped resources.
Data trace messaging (DTM)	External visibility of how data flows through the embedded system. This may include DRM and/or DWM.
JTAG compliant	Device complying to IEEE 1149.1 JTAG standard.
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 through a JTAG data register (DR) scan.
Nexus1	The e200z6 (OnCE) debug module. This module integrated with each e200z6 processor provides all static, core-halted, debug functionality. This module complies with class 1 of the *IEEE-ISTO 5001 standard*.
Ownership trace message (OTM)	Visibility of process/function that is currently executing.
Public messages	Messages on the auxiliary pins for accomplishing common visibility and controllability requirements.
SOC	System-on-a-chip (SOC) signifies all of the modules on a single die. This generally includes one or more processors with associated peripherals, interfaces and memory modules.
Standard	The phrase “according to the standard” is used to indicate 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 which does not cause the processor to halt. Instead, a pin is used to signal that the condition occurred. A watchpoint message is also generated.

11.1.3 Feature List

The Nexus3 module is compliant with class 3 of the *IEEE-ISTO 5001-2003* standard. The following features are implemented:

- Program trace through branch trace messaging (BTM). Displays program flow discontinuities, direct and indirect branches, and exceptions, 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 (DWM) and data read messaging (DRM). DRM and DWM provide 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). 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 registers and memory map through the JTAG port. This allows for enhanced download/upload capabilities.
- Watchpoint messaging through the auxiliary pins
- Watchpoint trigger enable of program and/or data trace messaging
- 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*
  - One watchpoint event pin, *nex_evto_b*
  - One event in pin, *nex_evti_b*
  - One message clock out (MCKO) pin
- Registers for program trace, data trace, ownership trace and watchpoint trigger
- All features controllable and configurable through the JTAG port

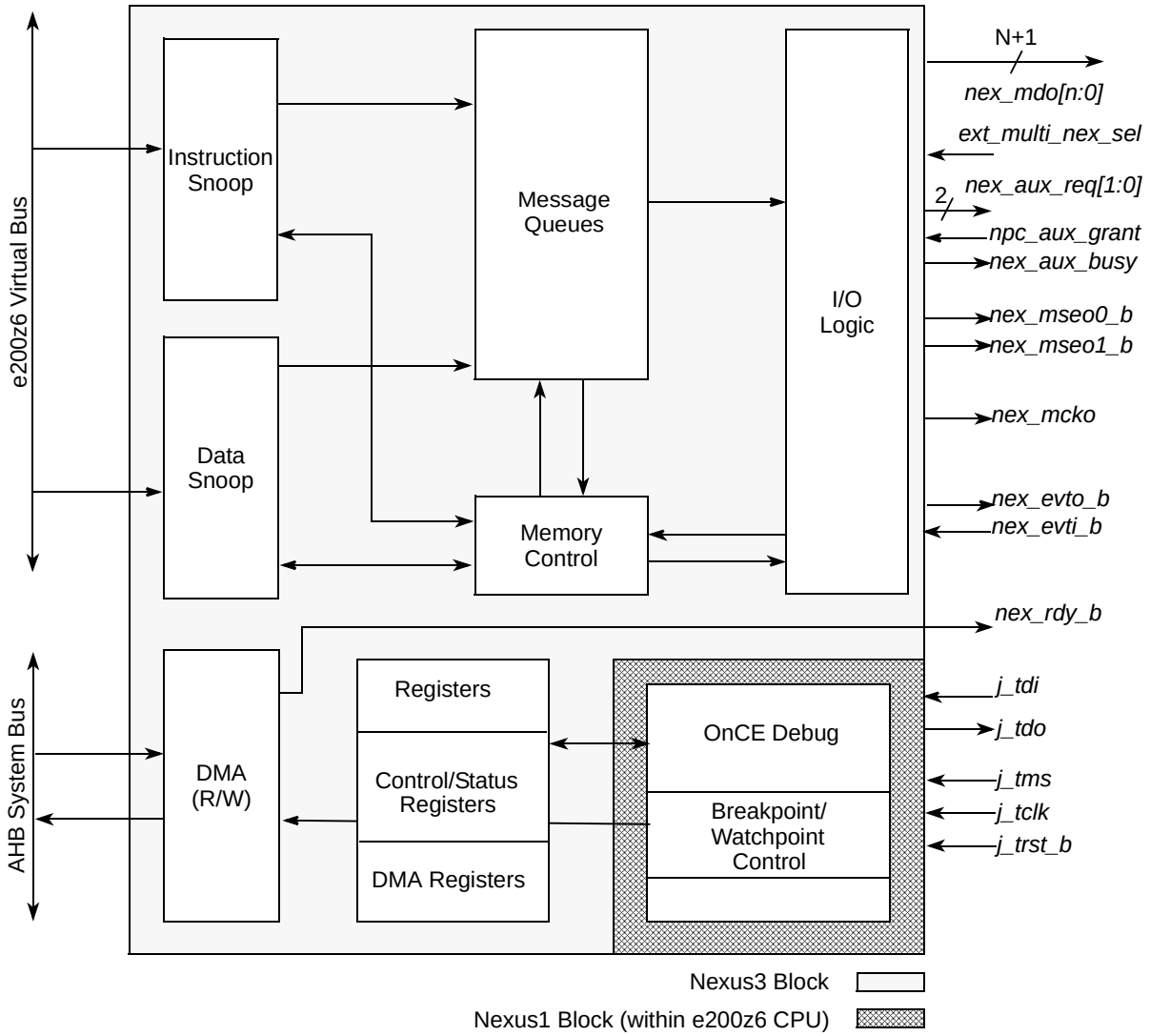
NOTE

Configuration of the message data out pins is controlled by the port control register at the SoC level in multiple Nexus implementations. For single Nexus implementations, this configuration is controlled by DC1 within the e200z6 Nexus3 module.

In either implementation, full port mode (FPM—maximum number of MDO pins) or reduced port mode (RPM—minimum number of MDO pins) is supported. This setting should not be changed while the system is running.

NOTE

The configuration of the message start/end out pins, 1 or 2, is determined at the SOC integration level. This option is hard wired based on SOC bandwidth requirements. Figure 11-1 shows the functional block diagram.



Note: The *nex_aux_req[1:0]*, *npc_aux_grant*, and *nex_aux_busy* signals are used for inter-module communication in a multiple Nexus environment. They are not pins on the SoC.

Figure 11-1. Nexus3 Functional Block Diagram

11.2 Enabling Nexus3 Operation

The Nexus module is enabled by loading a single instruction, NEXUS3-Access, into the JTAG instruction register/OnCE OCMD register. For the e200z6 Nexus3 module, the OCMD value is 0b00_0111_1100. Once enabled, the module is ready to accept control input through the JTAG/OnCE pins.

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 can also be disabled if a power-on

reset (POR) event occurs. If the Nexus3 module is disabled, no trace output is provided, and the module disables auxiliary port output pins, *nex_mdo[n:0]*, *nex_mseo[1:0]*, and *nex_mcko*. Nexus registers are not available for reads or writes.

11.3 TCODEs Supported

The Nexus3 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 5001-2003* standard defines a set of public messages. The Nexus3 block supports the public TCODEs seen in Table 11-2. Each message contains multiple packets transmitted in the order shown in the table.

Table 11-2. Public TCODEs Supported

Message Name	Minimum Packet Size (Bits)	Maximum Packet Size (Bits)	Packet Type	Packet Description
Debug status	6	6	Fixed	TCODE number = 0 (0x00)
	4	4	Fixed	Source processor identifier (multiple Nexus configuration)
	8	8	Fixed	Debug status register (DS[31–24])
Ownership trace message	6	6	Fixed	TCODE number = 2 (0x02)
	4	4	Fixed	Source processor identifier (multiple Nexus configuration)
	32	32	Fixed	Task/process ID tag
Program trace–Direct branch message	6	6	Fixed	TCODE number = 3 (0x03)
	4	4	Fixed	Source processor identifier (multiple Nexus configuration)
	1	8	Variable	Number of sequential instructions executed since last taken branch
Program trace–Indirect branch message	6	6	Fixed	TCODE number = 4 (0x04)
	4	4	Fixed	Source processor identifier (multiple Nexus configuration)
	1	8	Variable	Number of sequential instructions executed since last taken branch
	1	32	Variable	Unique part of target address for taken branches/exceptions
Data trace–Data write message	6	6	Fixed	TCODE number = 5 (0x05)
	4	4	Fixed	Source processor identifier (multiple Nexus configuration)
	3	3	Fixed	Data size. Refer to Table 11-6.
	1	32	Variable	Unique portion of the data write address
	1	64	Variable	Data write value(s). See data trace section for details.

Table 11-2. Public TCODEs Supported (continued)

Message Name	Minimum Packet Size (Bits)	Maximum Packet Size (Bits)	Packet Type	Packet Description
Data trace–Data read message	6	6	Fixed	TCODE number = 6 (0x06)
	4	4	Fixed	Source processor identifier (multiple Nexus configuration)
	3	3	Fixed	Data size. Refer to Table 11-6.
	1	32	Variable	Unique portion of the data read address
	1	64	Variable	Data read value(s). See data trace section for details.
Error message	6	6	Fixed	TCODE number = 8 (0x08)
	4	4	Fixed	Source processor identifier (multiple Nexus configuration)
	5	5	Fixed	Error code
Program trace–Direct branch message with synchronization	6	6	Fixed	TCODE number = 11 (0x0B)
	4	4	Fixed	Source processor identifier (multiple Nexus configuration)
	1	8	Variable	Number of sequential instructions executed since last taken branch
	1	32	Variable	Full target address (leading zeros truncated)
Program trace–Indirect branch message with synchronization	6	6	Fixed	TCODE number = 12 (0x0C)
	4	4	Fixed	Source processor identifier (multiple Nexus configuration)
	1	8	Variable	Number of sequential instructions executed since last taken branch
	1	32	Variable	Full target address (leading zeros truncated)
Data trace–Data write message with synchronization	6	6	Fixed	TCODE number = 13 (0x0D)
	4	4	Fixed	Source processor identifier (multiple Nexus configuration)
	3	3	Fixed	Data size. Refer to Table 11-6.
	1	32	Variable	Full access address (leading zeros truncated)
	1	64	Variable	Data write value(s). See data trace section for details.
Data trace–Data read message with synchronization	6	6	Fixed	TCODE number = 14 (0x0E)
	4	4	Fixed	Source processor identifier (multiple Nexus configuration)
	3	3	Fixed	Data size. Refer to Table 11-6.
	1	32	Variable	Full access address (leading zeros truncated)
	1	64	Variable	Data read value(s). See data trace section for details.
Watchpoint message	6	6	Fixed	TCODE number = 15 (0x0F)
	4	4	Fixed	Source processor identifier (multiple Nexus configuration)
	8	8	Fixed	Number indicating watchpoint source(s)

Table 11-2. Public TCODEs Supported (continued)

Message Name	Minimum Packet Size (Bits)	Maximum Packet Size (Bits)	Packet Type	Packet Description
Port replacement–Output message	6	6	Fixed	TCODE number = 20 (0x14)
	16	16	Fixed	Direction of each low-speed I/O bit (0 = input / 1=output)
	16	16	Fixed	Low-speed output data (per bit of direction)
Resource full message	6	6	Fixed	TCODE number = 27 (0x1B)
	4	4	Fixed	Source processor identifier (multiple Nexus configuration)
	4	4	Fixed	Resource code. Refer to Table 11-4. Indicates which resource is the cause of this message.
	1	32	Variable	Branch / predicate instruction history (see Section 11.7.1, “Branch Trace Messaging (BTM))
Program trace–Indirect branch history message	6	6	Fixed	TCODE number = 28 (0x1C). See note below.
	4	4	Fixed	Source processor identifier (multiple Nexus configuration)
	1	8	Variable	Number of sequential instructions executed since last taken branch
	1	32	Variable	Unique part of target address for taken branches/exceptions
	1	32	Variable	Branch / predicate instruction history (See Section 11.7.1, “Branch Trace Messaging (BTM).”)
Program trace–Indirect branch history message with synchronization	6	6	Fixed	TCODE number = 29 (0x1D). See note below.
	4	4	Fixed	Source processor identifier (multiple Nexus configuration)
	1	8	Variable	Number of sequential instructions executed since last taken branch
	1	32	Variable	Full target address (leading zero (0) truncated)
	1	32	Variable	Branch / predicate instruction history (See Section 11.7.1, “Branch Trace Messaging (BTM).”)
Program trace–Program correlation message	6	6	Fixed	TCODE number = 33 (0x21)
	4	4	Fixed	Source processor identifier (multiple Nexus configuration)
	4	4	Fixed	Event correlated with program flow. Refer to Table 11-5.
	1	8	Variable	Number of sequential instructions executed since last taken branch
	1	32	Variable	Branch / predicate instruction history (see Section 11.7.1, “Branch Trace Messaging (BTM)”)

Table 11-3 shows the error code encodings used when reporting an error through the Nexus3 error message.

Table 11-3. Error Code Encodings (TCODE = 8)

Error Code (ECODE)	Description
00000	Ownership trace overrun
00001	Program trace overrun
00010	Data trace overrun
00011	Read/write access error
00101	Invalid access opcode (Nexus register unimplemented)
00110	Watchpoint overrun
00111	Program trace or data trace and ownership trace overrun
01000	Program trace or data trace or ownership trace and watchpoint overrun
01001–10111	Reserved
11000	BTM lost due to collision with higher priority message
11001–11111	Reserved

Table 11-4 shows the encodings used for resource codes for certain messages.

Table 11-4. Resource Code Encodings (TCODE = 27)

Resource Code (RCODE)	Description
0001	Program trace, branch/predicate instruction history. This type of packet is terminated by a stop bit set to 1 after the last history bit.

Table 11-5 shows the event code encodings used for certain messages.

Table 11-5. Event Code Encodings (TCODE = 33)

Event Code (EVCODE)	Description
0000	Entry into debug mode
0001	Entry into low power mode (CPU only)
0010–1111	Reserved

Table 11-6 shows the data trace size encodings used for certain messages.

Table 11-6. Data Trace Size Encodings (TCODE = 5, 6, 13, or 14)

DTM Size Encoding	Transfer Size
000	Byte
001	Half word (2 bytes)
010	Word (4 bytes)

Table 11-6. Data Trace Size Encodings (TCODE = 5, 6, 13, or 14) (continued)

DTM Size Encoding	Transfer Size
011	Double word (8 bytes)
100	String (3 bytes)
101–111	Reserved

NOTE

Program trace can be implemented using either branch history/predicate instruction messages, or traditional direct/indirect branch messages, and the user can select between the two types. The advantages of each are discussed in Section 11.7.1, "Branch Trace Messaging (BTM)." If the branch history method is selected, the shaded TCODES above will not be messaged out.

11.4 Nexus3 Programmer's Model

This section describes the Nexus3 programmers model. Nexus3 registers are accessed using the JTAG/OnCE port in compliance with IEEE 1149.1. See Section 11.5, "Nexus3 Register Access through JTAG/OnCE," for details on Nexus3 register access.

NOTE

Nexus3 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 standard*.

Table 11-7 shows the register map for the Nexus3 module.

Table 11-7. Nexus3 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 control1 (DC1)	0x2	R/W	0x04	0x05
Development control2 (DC2)	0x3	R/W	0x06	0x07
Development status (DS)	0x4	R	0x08	—
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

Table 11-7. Nexus3 Register Map (continued)

Nexus Register	Nexus Access Opcode	Read/Write	Read Address	Write Address
Data trace control (DTC)	0xD	R/W	0x1A	0x1B
Data trace start address1 (DTSA1)	0xE	R/W	0x1C	0x1D
Data trace start address2 (DTSA2)	0xF	R/W	0x1E	0x1F
Data trace end address1 (DTEA1)	0x12	R/W	0x24	0x25
Data trace end address2 (DTEA2)	0x13	R/W	0x26	0x27
Reserved	0x14–0x3F	—	0x28–0x7E	0x29–7F

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 Nexus3 controller in a multiple Nexus implementation, not in the e200z6 Nexus3 module. The SoC's CSC register is readable through Nexus3, but the PCR is shown here for reference only.

2 PCR_INDEX is a parameter determined by the SoC. Refer to the reference manual for the device integrating the e200z6 core for more information on how this parameter is implemented for each Nexus module.

11.4.1 Client Select Control Register (CSC)

The CSC register determines which Nexus client is under development. This register is present at the top-level SOC Nexus3 controller to select one of multiple on-chip Nexus3 units. Figure 11-2 shows the CSC register.

	7 4	3 0
Field	—	CS
Reset	All zeros	
R/W	Read only	
Number	0x1	

Figure 11-2. Client Select Control Register

Table 11-8. CSC Field Descriptions

Bits	Name	Description
7–4	—	Reserved, should be cleared.
3–0	CSC	Client select control 0xX = Nexus client (SoC level)

11.4.2 Port Configuration Register (PCR)

The port configuration register (PCR), shown in Figure 11-3, controls the basic port functions for all Nexus modules in a multiple Nexus environment. This includes clock

control and auxiliary port width. All bits in this register are writable only once after system reset.

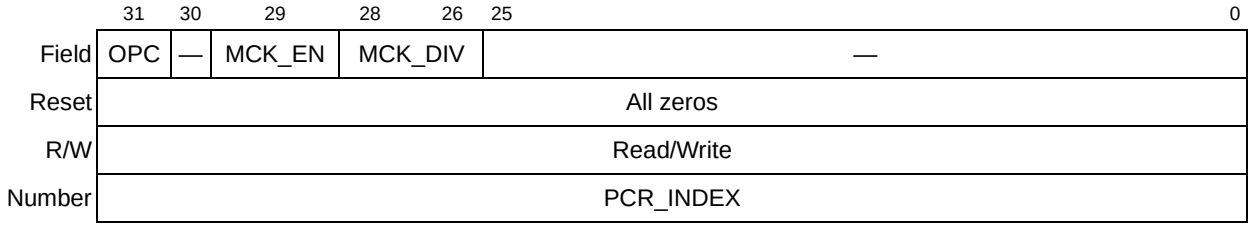


Figure 11-3. Port Configuration Register

Table 11-9. PCR Field Descriptions

Bits	Name	Description
31	OPC	Output port mode control 0 Reduced port mode configuration (minimum number of *nex_mdo[n:0]* pins defined by SOC) 1 Full port mode configuration (maximum number of *nex_mdo[n:0]* pins defined by SOC)
30	—	Reserved
29	MCK_EN	MCKO clock enable. See note below. 0 *nex_mcko* is disabled 1 *nex_mcko* is enabled
28–26	MCK_DIV	MCKO clock divide ratio 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

NOTE

The CSC and PCR registers exist in a separate module at the SoC level in a multiple Nexus environment. If the e200z6 Nexus3 module is the only Nexus module, these registers are not implemented and the e200z6 Nexus3-defined development control register 1 (DC1) is used to control Nexus port functionality.

11.4.3 Development Control Register 1, 2 (DC1, DC2)

The development control registers are used to control the basic development features of the Nexus3 module. Development control register 1 is shown in Figure 11-4 and its fields are described in Table 11-10.

	31	30 29	28 27	26	25	24	23 8	7 5	4 3	2 0
Field	OPC	MCK_DIV	EOC	—	PTM	WEN	—	OVC	EIC	TM
Reset	All zeros									
R/W	Read/Write									
Number	0x2									

Figure 11-4. Development Control Register 1 (DC1)

Table 11-10. DC1 Field Descriptions

Bits	Name	Description
31	OPC	Output port mode control 0 Reduced port mode configuration (minimum number of *nex_mdo[n:0]* pins defined by SOC) 1 Full port mode configuration (maximum number of *nex_mdo[n:0]* pins defined by SOC)
30–29	MCK_DIV	MCKO clock divide ratio. See note below. 00 *nex_mcko* is 1x processor clock freq. 01 *nex_mcko* is 1/2x processor clock freq. 10 *nex_mcko* is 1/4x processor clock freq. 11 *nex_mcko* is 1/8x processor clock freq.
28–27	EOC	EVTO control 00 *nex_evto_b* upon occurrence of watchpoints (configured in DC2) 01 *nex_evto_b* upon entry into debug mode 10 *nex_evto_b* upon timestamping event 11 Reserved
26	—	Reserved
25	PTM	Program trace method 0 Program trace uses traditional branch messages. 1 Program trace uses branch history messages.
24	WEN	Watchpoint trace enable 0 Watchpoint messaging disabled 1 Watchpoint messaging enabled
23–8	—	Reserved
7–5	OVC	Overrun control 000 Generate overrun messages 001–010 Reserved 011 Delay processor for BTM/DTM/OTM overruns 1XX Reserved

Table 11-10. DC1 Field Descriptions (continued)

Bits	Name	Description
4–3	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
2–0	TM	Trace mode 000 No trace 1XX Program trace enabled X1X Data trace enabled XX1 Ownership trace enabled

NOTE

OPC and 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 DC1 be modified only in one of these two modes.

Development control register 2 is shown in Figure 11-5 and its fields are described in Table 11-11.

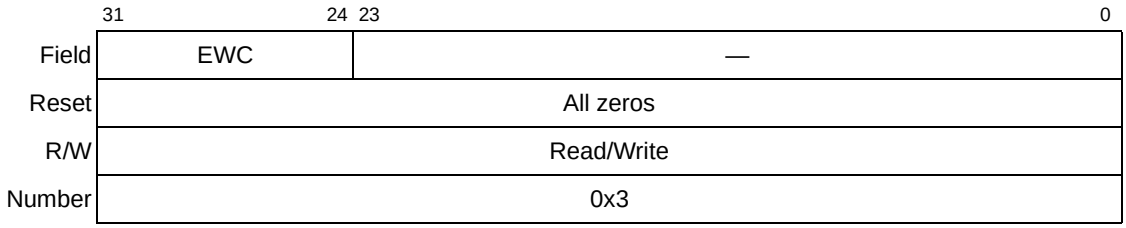


Figure 11-5. Development Control Register 2 (DC2)

Table 11-11. DC2 Field Descriptions

Bits	Name	Description
31–24	EWC	EVTO Watchpoint Configuration 00000000 No watchpoints trigger *nex_evto_b* 1xxxxxxx Watchpoint #0 (IAC1 from Nexus1) triggers *nex_evto_b* x1xxxxxx Watchpoint #1 (IAC2 from Nexus1) triggers *nex_evto_b* xx1xxxxx Watchpoint #2 (IAC3 from Nexus1) triggers *nex_evto_b* xxx1xxxx Watchpoint #3 (IAC4 from Nexus1) triggers *nex_evto_b* xxxx1xxx Watchpoint #4 (DAC1 from Nexus1) triggers *nex_evto_b* xxxxx1xx Watchpoint #5 (DAC2 from Nexus1) triggers *nex_evto_b* xxxxxx1x Watchpoint #6 (DCNT1 from Nexus1) triggers *nex_evto_b* xxxxxxx1 Watchpoint #7 (DCNT2 from Nexus1) triggers *nex_evto_b*
23–0	—	Reserved

NOTE

The EOC bits in DC1 must be programmed to trigger $\overline{\text{EVTO}}$ on watchpoint occurrence for the EWC bits to have any effect.

11.4.4 Development Status Register (DS)

The development status register, Figure 11-6, is used to report system debug status. When debug mode is entered or exited, or an SOC- or e200z6-defined low-power mode is entered, a debug status message is transmitted with DS[31–25]. The external tool can read this register at any time.

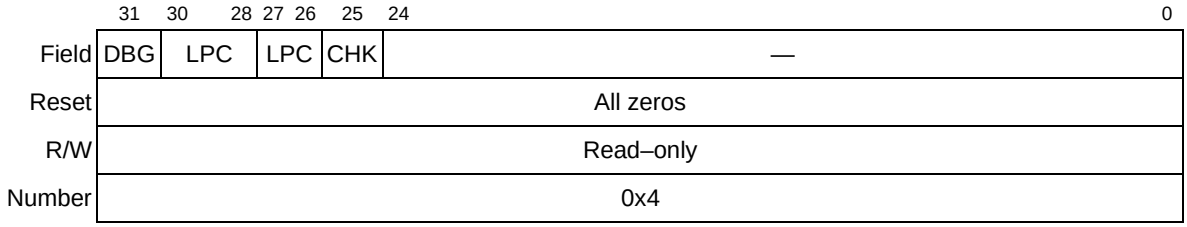


Figure 11-6. Development Status Register (DS)

Table 11-12. DS Field Descriptions

Bits	Name	Description
31	DBG	e200z6 CPU debug mode status 0 CPU not in debug mode 1 CPU in debug mode (*jd_debug_b* signal asserted)
30–28	LPS	e200z6 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	e200z6 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 Reserved
25	CHK	e200z6 CPU checkstop status 0 CPU not in checkstop state 1 CPU in checkstop state (*p_chkstop* signal asserted)
24–0	—	Reserved, should be cleared.

11.4.5 Read/Write Access Control/Status Register (RWCS)

The read write access control/status register, shown in Figure 11-7, 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. RWCS also provides read/write access status information; see Table 11-14.

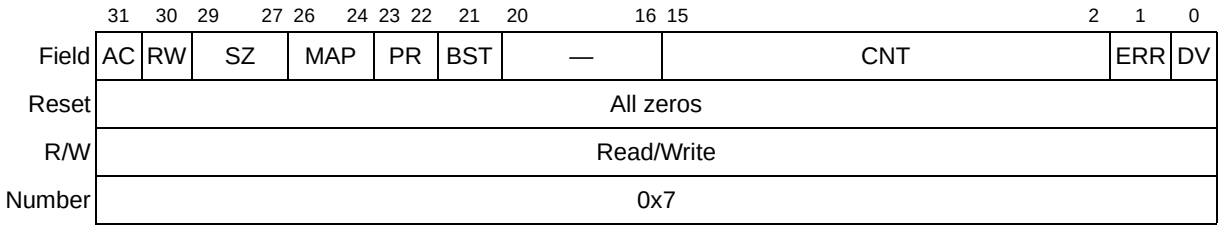


Figure 11-7. Read/Write Access Control/Status Register (RWCS)

Table 11-13. RWCS Field Descriptions

Bits	Name	Description
31	AC	Access control 0 End access 1 Start access
30	RW	Read/write select 0 Read access 1 Write access
29–27	SZ	Word size 000 8-bit (byte) 001 16-bit (half-word) 010 32-bit (word) 011 64-bit (double word - only in burst mode) 100–111 Reserved (default to word)
26–24	MAP	MAP select 000 Primary memory map 001–111 Reserved
23–22	PR	Read/write access priority 00 Lowest access priority 01 Reserved (default to lowest priority) 10 Reserved (default to lowest priority) 11 Highest access priority
21	BST	Burst Control 0 Block accesses are single bus cycle at a time 1 Block accesses are performed as burst operation **Note:** This optional (determined by SOC integration) feature allows limited burst accesses to the AHB. The Nexus buffer (if implemented) holds 32-bytes of data and only supports double-word bursts (RWA 2–0 are ignored). See Section 11.10, "Nexus3 Read/Write Access to Memory-Mapped Resources" for details on burst implementation.
20–16	—	Reserved
15–2	CNT	Access control count. Number of accesses of word size SZ
1	ERR	Read/write access error. See Table 11-14.
0	DV	Read/write access data valid. See Table 11-14.

Table 11-14 details the status bit encodings.

Table 11-14. Read/Write Access Status Bit Encodings

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

11.4.6 Read/Write Access Data Register (RWD)

The read/write access data register, shown in Figure 11-8, provides the data to/from system bus memory-mapped locations when initiating a read or a write access.

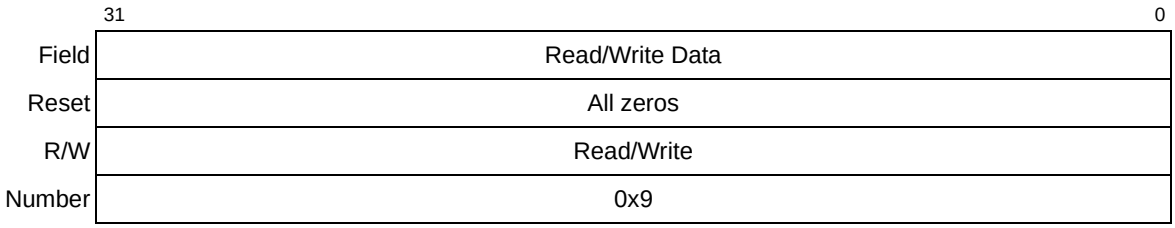


Figure 11-8. Read/Write Access Data Register (RWD)

11.4.7 Read/Write Access Address Register (RWA)

The read/write access address register, shown in Figure 11-9, provides the system bus address to be accessed when initiating a read or a write access.

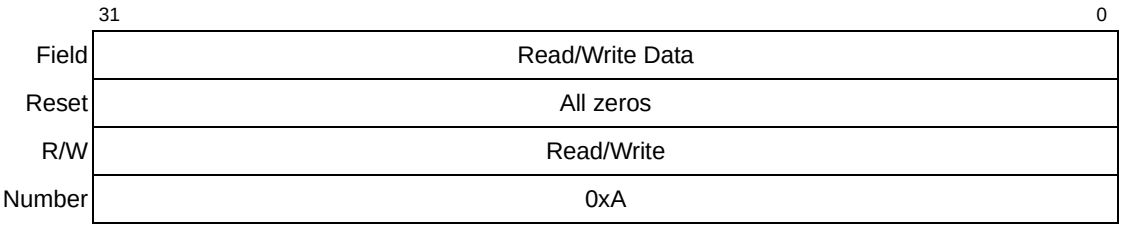


Figure 11-9. Read/Write Access Address Register (RWA)

11.4.8 Watchpoint Trigger Register (WT)

The watchpoint trigger register, shown in Figure 11-10, allows the watchpoints defined within the e200z6 Nexus1 logic to trigger actions. These watchpoints can control program and/or data trace enable and disable. The WT bits can be used to produce an address related window for triggering trace messages.

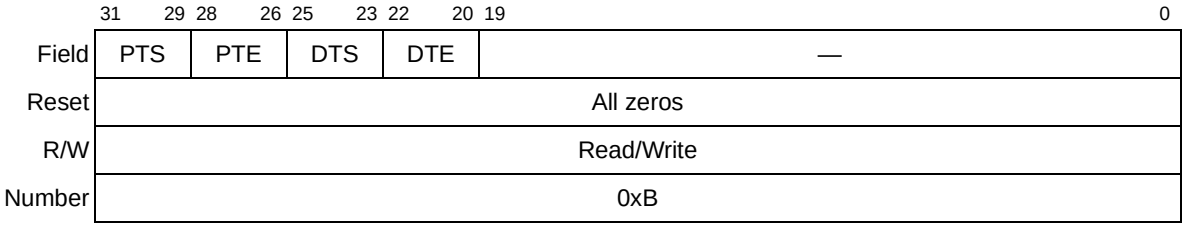


Figure 11-10. Watchpoint Trigger Register (WT)

Table 11-15 details the watchpoint trigger register fields.

Table 11-15. WT Field Descriptions

Bits	Name	Description
31–29	PTS	Program trace start control 000 Trigger disabled 001 Use watchpoint #0 (IAC1 from Nexus1) 010 Use watchpoint #1 (IAC2 from Nexus1) 011 Use watchpoint #2 (IAC3 from Nexus1) 100 Use watchpoint #3 (IAC4 from Nexus1) 101 Use watchpoint #4 (DAC1 from Nexus1) 110 Use watchpoint #5 (DAC2 from Nexus1) 111 Use watchpoint #6 or #7 (DCNT1 or DCNT2 from Nexus1)
28–26	PTE	Program trace end control 000 Trigger disabled 001 Use watchpoint #0 (IAC1 from Nexus1) 010 Use watchpoint #1 (IAC2 from Nexus1) 011 Use watchpoint #2 (IAC3 from Nexus1) 100 Use watchpoint #3 (IAC4 from Nexus1) 101 Use watchpoint #4 (DAC1 from Nexus1) 110 Use watchpoint #5 (DAC2 from Nexus1) 111 Use watchpoint #6 or #7 (DCNT1 or DCNT2 from Nexus1)
25–23	DTS	Data trace start control 000 Trigger disabled 001 Use watchpoint #0 (IAC1 from Nexus1) 010 Use watchpoint #1 (IAC2 from Nexus1) 011 Use watchpoint #2 (IAC3 from Nexus1) 100 Use watchpoint #3 (IAC4 from Nexus1) 101 Use watchpoint #4 (DAC1 from Nexus1) 110 Use watchpoint #5 (DAC2 from Nexus1) 111 Use watchpoint #6 or #7 (DCNT1 or DCNT2 from Nexus1)
22–20	DTE	Data trace end control 000 Trigger disabled 001 Use watchpoint #0 (IAC1 from Nexus1) 010 Use watchpoint #1 (IAC2 from Nexus1) 011 Use watchpoint #2 (IAC3 from Nexus1) 100 Use watchpoint #3 (IAC4 from Nexus1) 101 Use watchpoint #4 (DAC1 from Nexus1) 110 Use watchpoint #5 (DAC2 from Nexus1) 111 Use watchpoint #6 or #7 (DCNT1 or DCNT2 from Nexus1)
19–0	—	Reserved, should be cleared

NOTE

The WT bits only control program/data trace if the TM bits within DC1 have not already been set to enable program and data trace respectively.

11.4.9 Data Trace Control Register (DTC)

The data trace control register controls whether DTM messages are restricted to reads, writes, or both for a user programmable address range. There are two data trace channels controlled by the DTC for the Nexus3 module. Each channel can also be programmed to trace data accesses or instruction accesses. Figure 11-11 shows DTC.

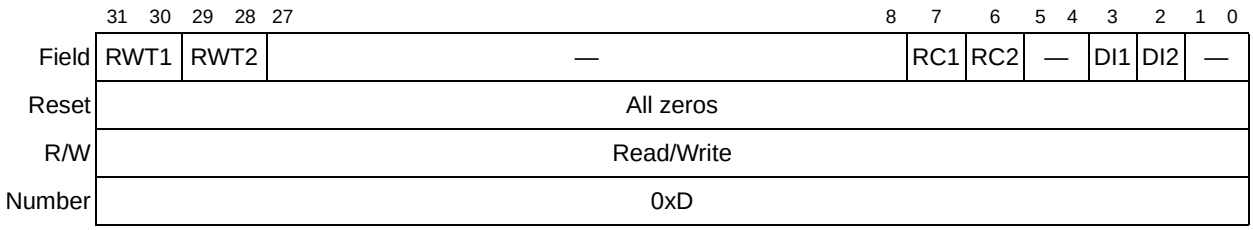


Figure 11-11. Data Trace Control Register (DTC)

Table 11-16 details the data trace control register fields.

Table 11-16. DTC Field Descriptions

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
27–8	—	Reserved, should be cleared.
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
5–4	—	Reserved, should be cleared.

Table 11-16. DTC Field Descriptions (continued)

Bits	Name	Description
3	DI1	Data access/instruction access trace 1 0 Condition trace on data accesses 1 Condition trace on instruction accesses
2	DI2	Data access/instruction access trace 2 0 Condition trace on data accesses 1 Condition trace on instruction accesses
1–0	—	Reserved, should be cleared.

11.4.10 Data Trace Start Address 1 and 2 Registers (DTSA1 and DTSA2)

The data trace start address registers, shown in Figure 11-12, define the start addresses for each trace channel.

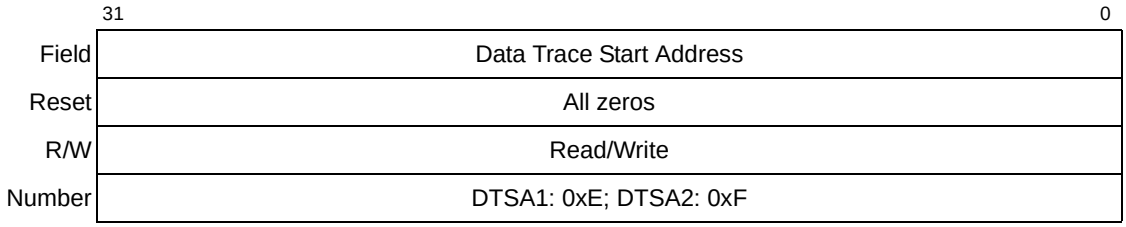


Figure 11-12. Data Trace Start Address Registers 1 and 2 (DTSA*n*)

11.4.11 Data Trace End Address Registers 1 and 2 (DTEA1 and DTEA2)

The data trace end address registers, shown in Figure 11-13, define the end addresses for each trace channel.

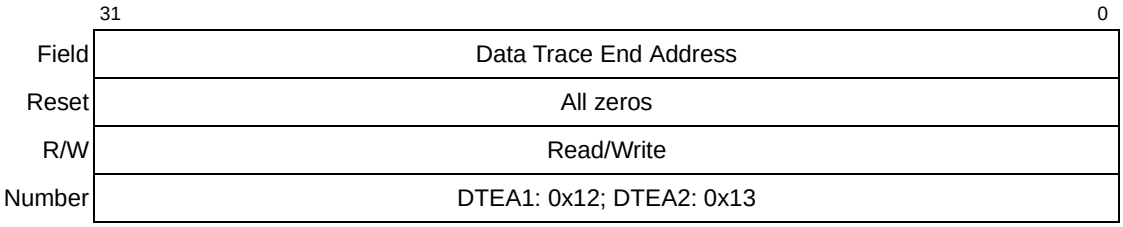


Figure 11-13. Data Trace End Address Registers 1 and 2 (DTEA*n*)

Table 11-17 illustrates the range that is selected for data trace for various cases of DTSA being less than, greater than, or equal to DTEA.

Table 11-17. Data Trace—Address Range Options

Programmed Values	Range Control Bit Value	Range Selected
DTSA < DTEA	0	The address range lies between the values specified by DTSA and DTEA. (DTSA -> <- DTEA)
	1	The address range lies outside the values specified by DTSA and DTEA. (<-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 exclusive of the DTSA and DTEA addresses.

11.5 Nexus3 Register Access through JTAG/OnCE

Access to Nexus3 register resources is enabled by loading a single instruction, NEXUS3-Access, into the JTAG instruction register/OnCE OCMD register. For the Nexus3 block, the OCMD value is 0b00_0111_1100.

Once the NEXUS3-Access instruction has been loaded, the JTAG/OnCE port allows tool/target communications with all Nexus3 registers according to the register map in Table 11-7.

Reading/writing of a Nexus3 register then requires two passes through the data-scan path of the JTAG state machine 12 (see Section 11.15, “IEEE 1149.1 (JTAG) RD/WR Sequences”).

1. The first pass through the DR selects the Nexus3 register to be accessed by providing an index (see Table 11-7), and the direction, read/write. This is achieved by loading an 8-bit value into the JTAG data register (DR). This register has the format shown in Figure 11-14.

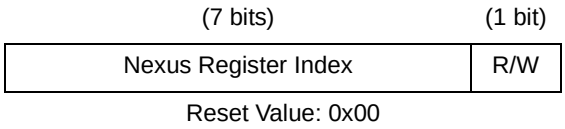


Figure 11-14. Nexus3 Register Access through JTAG/OnCE (Example)

Table 11-18. Nexus Register Example

Field	Description

Table 11-18. Nexus Register Example

Nexus Register Index	Selected from values in Table 11-7
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, least-significant bit first.
   a) During a read access, data is latched from the selected Nexus register when the JTAG state machine passes through the capture-DR state.
   b) During a write access, data is latched into the selected Nexus register when the JTAG state machine passes through the update-DR state.

11.6 Ownership Trace

This section details the ownership trace features of the Nexus3 module.

11.6.1 Overview

Ownership trace provides a macroscopic view, such as task flow reconstruction, when debugging software is 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.

11.6.2 Ownership Trace Messaging (OTM)

Ownership trace information is messaged by means of the auxiliary port using OTM. For e200z6 processors, there are two distinct methods for providing task/process ID data. Some e200 processors contain a BookE–defined process ID register within the CPU while others may not. Within Nexus, task/process ID data is handled in one of the following two ways in order to maintain IEEE-ISTO 5001 compliance.

1. If the process ID register exists, 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. See Section 2.14.5, “Process ID Register (PID0).”
2. If the process ID register does not exist, the user base address register (UBA) is implemented within Nexus. The UBA can be accessed by means of the JTAG/OnCE port and contains the address of the ownership trace register (OTR). The memory-mapped OTR is updated by the operating system software to provide task/process ID information.

NOTE

The e200z6 includes a process ID register (PID0), thus the UBA functionality is not implemented.

There are two conditions that cause an ownership trace message:

1. When new information is updated in the OTR register or process ID register by the e200z6 processor, the data is latched within Nexus and is messaged out through the auxiliary port, allowing development tools to trace ownership flow.
2. When the periodic OTM message counter expires after 255 queued messages without an OTM, an OTM is sent. The data is sent from either the latched OTR data or the latched process ID data. This allows processors using virtual memory to be regularly updated with the latest process ID.

Ownership trace information is messaged out in the format shown in Figure 11-15

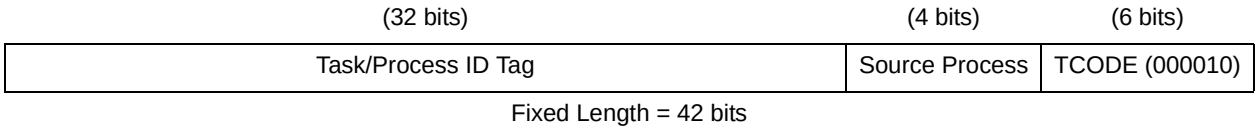


Figure 11-15. Ownership Trace Message Format

11.6.3 OTM Error Messages

An error message occurs when the message queue is full and a new message cannot be queued. The FIFO discards incoming messages until it has completely emptied the queue. Once the queue is emptied, an error message is queued. The error encoding indicates which types of messages attempted to be queued while the FIFO was being emptied.

If only an OTM message attempts to enter the queue while the queue is being emptied, the error message only incorporates the OTM error encoding 00000. If both OTM and either BTM or DTM (that is, OTM and BTM or OTM and DTM) messages attempt to enter the queue, the error message incorporates the OTM and program or data trace error encoding 00111. If a watchpoint also attempts to be queued while the FIFO is being emptied, then the error message incorporates error encoding 01000.

NOTE

DC1[OVC] can be set to delay the CPU in order to alleviate, but not eliminate, potential overrun situations.

Error information is messaged out in the format shown in Figure 11-16.

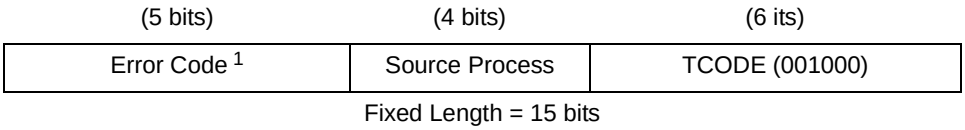


Figure 11-16. Error Message Format

[1] Must be one of 00000, 00111, or 01000

11.6.4 OTM Flow

Ownership trace messages are generated when the operating system writes to the e200z6 process ID register (PID0) or the memory-mapped ownership trace register (OTR).

The following flow describes the OTM process:

1. For the Nexus3 module, there are two different registers which can contain task/process ID data. Nexus uses one or the other for OTM.
   a) The process ID register is a system control register. It is internal to the e200z6 processor and can be accessed by using PowerPC instructions. PID0 contents are replicated on the pins of the processor and connected to Nexus. See Section 2.14.5, “Process ID Register (PID0),” for more information on PID0.
   b) The OTR is a memory-mapped register whose address is located in the UBA. The UBA is internal to the Nexus module and can be accessed by the IEEE-ISTO 5001 tool through the JTAG port.
2. If the UBA is implemented, only word writes to OTR are valid. Writes to PID0 pulse a write signal to Nexus. The data value written into the OTR or PID0 is latched and formed into the ownership trace message that is queued to be transmitted.
3. OTR or PID0 reads do not cause ownership trace messages to be transmitted by the Nexus3 module.
4. If the periodic OTM message counter expires after 255 queued messages without an OTM, an OTM is sent using the latched data from the previous OTR or PID0 write.

11.7 Program Trace

This section details the program trace mechanism supported by Nexus3 for the e200z6 processor. Program trace is implemented using branch trace messaging (BTM) as required by the class 3 *IEEE-ISTO 5001-2003* standard definition. Branch trace messaging for e200z6 processors is accomplished by snooping the e200z6 virtual address bus, between the CPU and MMU, attribute signals, and CPU status *p_pstat[0:5]*.

11.7.1 Branch Trace Messaging (BTM)

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. Direct or indirect branches not taken are counted as 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 history messaging facilitates program trace by providing the following information.

- Messaging for taken indirect branches and exceptions includes how many sequential instructions were executed since the last predicate instruction, taken indirect branch, or exception, the unique portion of the branch target address or exception vector address, and 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.

11.7.1.1 e200z6 Indirect Branch Message Instructions (Book E)

Table 11-19 shows the types of instructions and events which cause indirect branch messages or branch history messages to be encoded.

Table 11-19. Indirect Branch Message Sources

Source of Indirect Branch Message	Instructions
Taken branch relative to a register value	**bcctr**, **bcctrl**, **bclr**, **bclrl**
System call/trap exceptions taken	**sc**, **tw**, **twi**
Return from interrupts/exceptions	**rfi**, **rfci**, **rfdi**

11.7.1.2 e200z6 Direct Branch Message Instructions (Book E)

Table 11-20 shows the types of instructions that cause direct branch messages or will toggle a bit in the instruction history buffer to be messaged out in a resource full message or branch history message.

Table 11-20. Direct Branch Message Sources

Source of Direct Branch Message	Instructions
Taken direct branch instructions	**b**, **ba**, **bl**, **bla**, **bc**, **bca**, **bcl**, **bcla**
Instruction synchronize	**isync**

11.7.1.3 BTM using Branch History Messages

Traditional BTM 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. 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 direct 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.

11.7.1.4 BTM using Traditional Program Trace Messages

Program tracing can utilize either branch history messages (DC1[PTM] = 1) or traditional direct/indirect branch messages (DC1[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.

11.7.2 BTM Message Formats

The e200z6 Nexus3 block supports three types of traditional BTM messages: direct, indirect, and synchronized messages. It supports two types of branch history BTM messages: indirect branch history, and indirect branch history with synchronized messages. Debug status messages and error messages are also supported.

11.7.2.1 Indirect Branch Messages (History)

Indirect branches include all taken branches whose destination is determined at run time, interrupts, and exceptions. If DC1[PTM] is set, indirect branch information is messaged out in the format shown in Figure 11-17:

(1–32 bits)	(1–32 bits)	(1–8 bits)	(4 bits)	(6 bits)
Branch History	Relative Address	Sequence Count	Source Process	TCODE (011100)

Maximum length = 82 bit; Minimum length = 13 bits

Figure 11-17. Indirect Branch Message (History) Format

11.7.2.2 Indirect Branch Messages (Traditional)

If DC1[PTM] is cleared, indirect branch information is messaged out in the format shown in Figure 11-18:

(1–32 bits)	(1–8 bits)	(4 bits)	(6 bits)
Relative Address	Sequence Count	Source Process	TCODE (000100)

Maximum length = 50 bits; Minimum length = 12 bits

Figure 11-18. Indirect Branch Message Format

11.7.2.3 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 format shown in Figure 11-19:

(1–8 bits)	(4 bits)	(6 bits)
Sequence Count	Source Process	TCODE (000011)

Maximum Length = 18 bits; Minimum length = 11 bits

Figure 11-19. Direct Branch Message Format

NOTE

When DC1[PTM] is set, direct branch messages are not transmitted. Instead, each direct branch or predicated instruction toggles a bit in the history buffer.

11.7.2.4 Resource Full Messages

The resource full message is used in conjunction with the branch history messages. The resource full message is generated when the internal branch/predicate history buffer is full. If synchronization is needed at the time this message is generated, the synchronization is delayed until the next branch trace message that is not a resource full message.

The current value of the history buffer is transmitted as part of the resource full message. This information can be concatenated by the tool with the branch/predicate history information from subsequent messages to obtain the complete branch history for a

message. The internal history value is reset by this message and the I-CNT value is reset as a result of a bit being added to the history buffer.

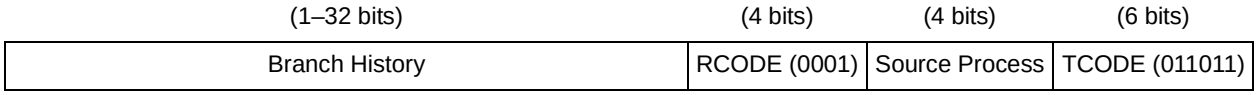


Figure 11-20. Resource Full Message Format

11.7.2.5 Debug Status Messages

Debug status messages report low-power mode and debug status. Entering/exiting debug mode as well as entering a low-power mode triggers a debug status message. Debug status information is sent out in the format shown in Figure 11-21:

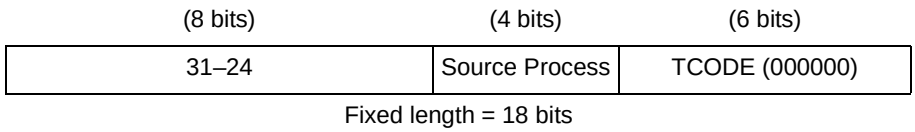


Figure 11-21. Debug Status Message Format

11.7.2.6 Program Correlation Messages

Program correlation messages are used to correlate events to the program flow that may not be associated with the instruction stream. In order to maintain accurate instruction tracing information when entering debug mode or a CPU low-power mode, where tracing may be disabled, this message is sent upon entry into one of these two modes and includes the instruction count and branch history. Program correlation is messaged out in the format shown in Figure 11-22:

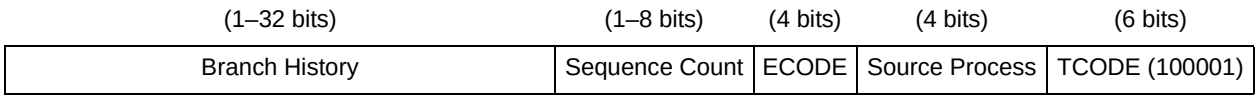


Figure 11-22. Program Correlation Message Format

11.7.2.7 BTM Overflow Error Messages

An error message occurs when to the message queue is full and a new message cannot be queued. The FIFO discards incoming messages until it has completely emptied the queue. Once emptied, an error message is queued. The error encoding indicates which types of messages attempted to be queued while the FIFO was being emptied.

If only a program trace message attempts to enter the queue while it is being emptied, the error message incorporates the program trace only error encoding, 00001. If both OTM and program trace messages attempt to enter the queue, the error message incorporates the

OTM and program trace error encoding, 00111. If a watchpoint also attempts to be queued while the FIFO is being emptied, the error message incorporates error encoding 01000.

NOTE

DC1[OVC] can be set to delay the CPU in order to alleviate, but not eliminate, potential overrun situations.

Error information is messaged out in the format shown in Figure 11-23:

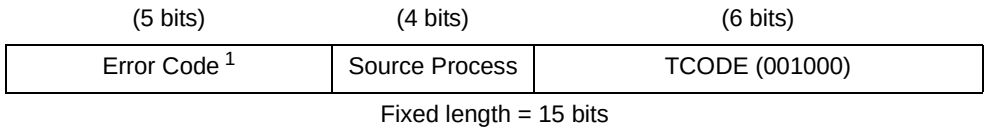


Figure 11-23. Error Message Format

1 Must be one of 00001, 00111, or 01000.

11.7.2.8 Program Trace Synchronization Messages

A program trace direct/indirect branch with synchronization message is messaged using the auxiliary port, provided program trace is enabled, for the following conditions (see Table 11-21):

- 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, which can be caused by any trace message
- Upon direct/indirect branch after the periodic program trace counter has expired, indicating 255 without-synchronization program trace messages have occurred since the last with-synchronization message occurred
- Upon direct/indirect branch after assertion of the event-in (*nex_evti_b*) signal, 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 between branches
- Upon direct/indirect branch after a BTM message was lost due to an attempted access to a secure memory location (for SOCs with security)
- Upon direct/indirect branch after a BTM message was lost due to a collision entering the FIFO between the BTM message and either a watchpoint message or an ownership trace message

If the Nexus3 module is enabled at reset, a *nex_evti_b* assertion initiates a program trace direct/indirect branch with synchronization message if program trace is enabled upon the

first direct/indirect branch. The format for program trace direct/indirect branch with synchronization messages is shown in Figure 11-24:

(1–32 bits)	(1–8 bits)	(4 bits)	(6 bits)
Full Target Address	Sequence Count	Source Process	TOCODE (001011 or 001100)

Maximum length = 50 bits; Minimum length = 12 bits

Figure 11-24. Direct/Indirect Branch with Synchronization Message Format

The formats for program trace direct/indirect branch with synchronized messages and indirect branch history with synchronized messages are shown in Figure 11-25:

(1–32 bits)	(1–32 bits)	(1–8 bits)	(4 bits)	(6 bits)
Branch History	Full Target Address	Sequence Count	Source Process	TCODE (011101)

Maximum length = 82 bit; Minimum length = 13 bits

Figure 11-25. Indirect Branch History with Synchronization Message Format

Exception conditions that result in program trace synchronization are summarized in Table 11-21.

Table 11-21. 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 Nexus3 module are reset. Upon the first branch out of system reset, if program trace is enabled, the first program trace message is a direct/indirect branch with synchronization message.
Program trace enabled	The first program trace message, after program trace has been enabled, is a synchronization message.
Exit from low power/debug	Upon exit from a low-power mode or debug mode the next direct/indirect branch is converted to a direct/indirect branch with synchronization message.
Queue overrun	An error message occurs when the message queue is full and a new message cannot be queued. The FIFO discards messages until it has completely emptied the queue. Once emptied, an error message is queued. The error encoding indicates which types of messages attempted to be queued while the FIFO was being emptied. The next BTM message in the queue is a direct/indirect branch with synchronization message.
Periodic program trace synchronization	A forced synchronization occurs periodically after 255 program trace messages have been queued. A direct/indirect branch with synchronization message is queued. The periodic program trace message counter then resets.
Event in	If the Nexus module is enabled, assorting *nex_evti_b* initiates a direct/indirect branch with synchronization 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	When 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 with synchronization message is queued upon execution of the next branch.

Table 11-21. Program Trace Exception Summary (continued)

Exception Condition	Exception Handling
Attempted access to secure memory	For SOCs which implement security, any attempted branch to secure memory locations temporarily disables program trace and cause the corresponding BTM to be lost. The following direct/indirect branch queues a direct/indirect branch with synchronization message. The count value within this message will be inaccurate since the re-enable of program trace is not necessarily aligned on an instruction boundary.
Collision priority	All messages have the following priority: WPM → OTM → BTM → DTM. A BTM message which attempts to enter the queue at the same time as a watchpoint message or ownership trace message will be lost. An error message is sent indicating the BTM was lost. The following direct/indirect branch queues a direct/indirect branch with synchronization message. The count value within this message reflects the number of sequential instructions executed after the last successful BTM message was generated. This count includes the branch which did not generate a message due to the collision.

11.7.3 BTM Operation

11.7.3.1 Enabling Program Trace

Both types of branch trace messaging can be enabled in one of two ways:

- Setting DC1[TM] to enable program trace
- Using WT[PTS] to enable program trace on watchpoint hits. e200z6 watchpoints are configured within the CPU.

11.7.3.2 Relative Addressing

The relative address feature is compliant with the *IEEE-ISTO 5001-2003* standard recommendations and is designed to reduce the number of bits transmitted for addresses of indirect branch messages.

The address transmitted is relative to the target address of the instruction which triggered the previous indirect branch or synchronized message. It is generated by XORing the new address with the previous address and then using only the results up to the most significant 1 bit in the result. To recreate this address, an XOR of the most-significant zero-padded message address with the previously decoded address gives the current address. For the example given in Figure 11-26, assume the previous address (A1) = 0x0003FC01, and the new address (A2) = 0x0003F365:

Message Generation	
A1	0000 0000 0000 0011 1111 1100 0000 0001
A2	0000 0000 0000 0011 1111 0011 0110 0101

Figure 11-26. Relative Address Generation and Re-Creation Example

A1 ⊕ A2	0000 0000 0000 0000 0000 1111 0110 0100
M1 (Address Message)	1111 0110 0100
Address Re-creation	
A1	0000 0000 0000 0011 1111 1100 0000 0001
M1	0000 0000 0000 0000 0000 1111 0110 0100
A1 ⊕ M1 (A2)	0000 0000 0000 0011 1111 0011 0110 0101

Figure 11-26. Relative Address Generation and Re-Creation Example

11.7.3.3 Branch/Predicate Instruction History (HIST)

If DC1[PTM] is set, BTM messaging uses the branch history format. The branch history (HIST) packet in these messages provides a history of direct branch execution used for reconstructing program flow. This packet is implemented as a left-shifting shift register. The register is always pre-loaded with a value of one. This bit acts as a stop bit so that the development tools can determine which bit is the end of the history information. 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 on a taken branch, conditional or unconditional, and on any instruction whose predicate condition executed as true. A value of zero is shifted into the history buffer on any instruction whose predicate condition executed as false, as well as on branches not taken. This includes indirect as well as direct branches not taken. 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.

11.7.3.4 Sequential Instruction Count (I-CNT)

The I-CNT packet is present in all BTM messages. For traditional branch messages, I-CNT represents the number of sequential instructions, or non-taken branches in between direct/indirect branch messages.

For branch history messages, I-CNT represents the number of instructions executed since the last taken/non-taken direct branch, last taken indirect branch, or exception. Not-taken indirect branches are considered sequential instructions and cause the instruction count to increment. I-CNT also represents the number of instructions executed since the last predicate instruction.

The sequential instruction counter overflows when its value reaches 255. The next BTM message is converted to a synchronization type message.

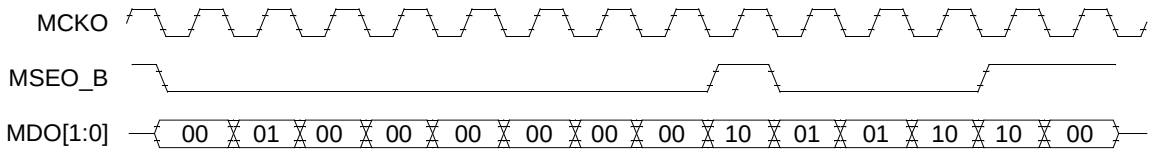
11.7.3.5 Program Trace Queueing

Nexus3 implements a programmable depth queue (a minimum of 32 entries is recommended) for queuing all messages. Messages that enter the queue are transmitted through the auxiliary pins in the order in which they are queued.

NOTE

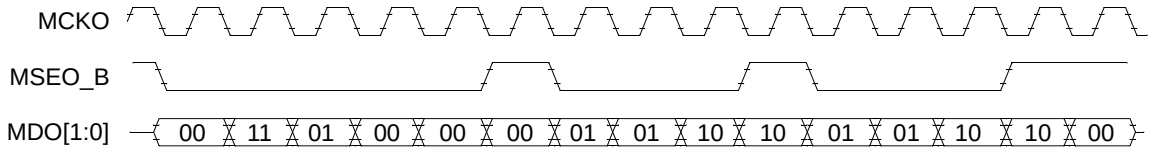
If multiple trace messages need to be queued at the same time, watchpoint messages have the highest priority:
(WPM → OTM → BTM → DTM).

11.7.4 Program Trace Timing Diagrams (2 MDO/1 MSEO Configuration)



TCODE = 4
Source processor = 0000
Number of sequential instructions = 128
Relative address = 0xA5

Figure 11-27. Program Trace—Indirect Branch Message (Traditional)



TCODE = 28
Source processor = 0000
Number of sequential instructions = 0
Relative address = 0xA5
Branch history = 010100101 (w/ stop)

Figure 11-28. Program Trace—Indirect Branch Message (History)

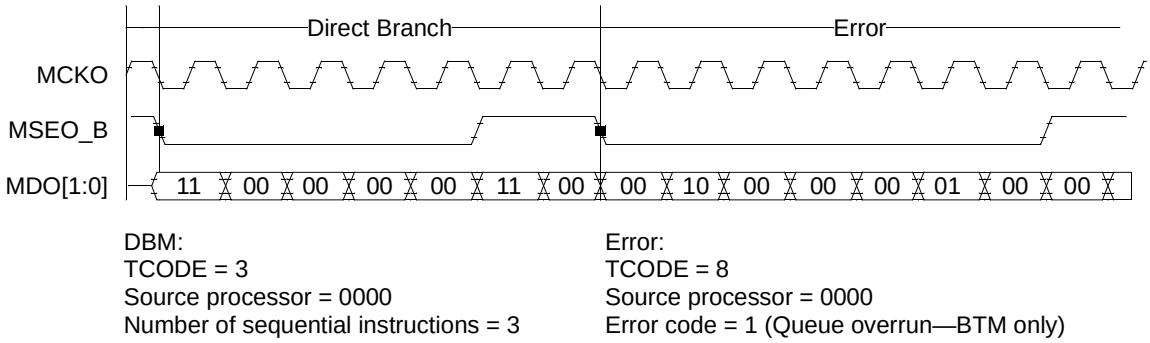


Figure 11-29. Program Trace—Direct Branch (Traditional) and Error Messages

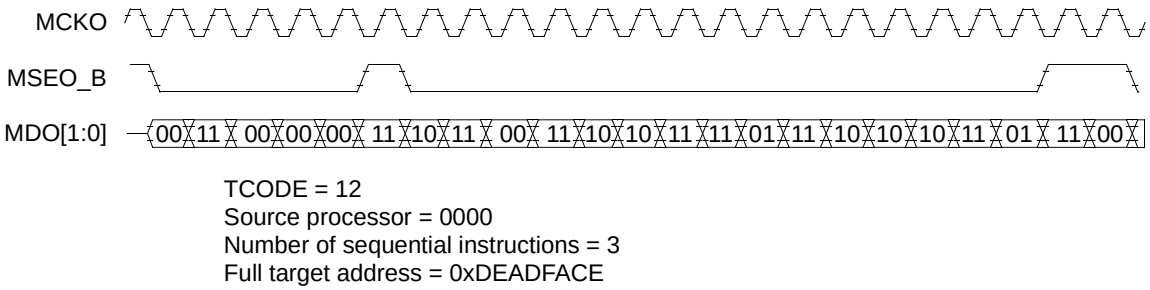


Figure 11-30. Program Trace—Indirect Branch with Synchronization Message

11.8 Data Trace

This section deals with the data trace mechanism supported by the Nexus3 module. Data trace is implemented by means of data write messaging (DWM) and data read messaging (DRM) in accordance with the *IEEE-ISTO 5001-2003* standard.

11.8.1 Data Trace Messaging (DTM)

Data trace messaging for the e200z6 is accomplished by snooping the e200z6 virtual data bus between the CPU and MMU, and storing the information for qualifying access, based on enabled features and matching target addresses. The Nexus3 module traces all data access that meet the selected range and attributes.

NOTE

Data trace is only performed on the e200z6 virtual data bus. This allows for data visibility for e200z6 processors which incorporate a data cache. Only e200z6 CPU-initiated accesses are traced. No DMA accesses to the AHB system bus are traced.

Data trace messaging can be enabled in one of two ways:

- Setting DC1[TM] to enable data trace.
- Using WT[DTS] to enable data trace on watchpoint hits. e200z6 watchpoints are configured within the Nexus1 module.

11.8.2 DTM Message Formats

The Nexus3 block supports five types of DTM messages: data write, data read, data write synchronization, data read synchronization, and error messages.

11.8.2.1 Data Write Messages

The data write message contains the data write value and the address of the write access, relative to the previous data trace message. Data write message information is messaged out in the format shown in Figure 11-31:

(1–64 bits)	(1–32 bits)	(3 bits)	(4 bits)	(6 bits)
Data Value(s)	Relative Address	Data Size	Source Process	TCODE (000101)

Maximum length = 109 bits; Minimum length = 15 bits

Figure 11-31. Data Write Message Format

11.8.2.2 Data Read Messages

The data read message contains the data read value and the address of the read access, relative to the previous data trace message. Data read message information is messaged out in the format shown in Figure 11-32:

(1–64 bits)	(1–32 bits)	(3 bits)	(4 bits)	(6 bits)
Data Value(s)	Relative Address	Data Size	Source Process	TCODE (000110)

Maximum length = 109 bits; Minimum length = 15 bits

Figure 11-32. Data Read Message Format

NOTE

For e200z6-based CPUs, the double-word encoding, *p_tsiz* = 0, indicates a double-word access and is sent out as a single data trace message with a single 64-bit data value.

The debug/development tool needs to distinguish the two cases based on the family of e200z6 processors.

11.8.2.3 DTM Overflow Error Messages

An error message occurs when the message queue is full and a new message cannot be queued. The FIFO discards incoming messages until it has completely emptied the queue. Once emptied, an error message is queued. The error encoding indicates which types of messages attempted to be queued while the FIFO was being emptied.

If only a DTM attempts to enter the queue while it is being emptied, the error message incorporates the data trace only error encoding, 00010. If both OTM and DTM attempt to enter the queue, the error message incorporates the OTM and data trace error encoding, 00111. If a watchpoint also attempts to be queued while the FIFO is being emptied, the error message incorporates error encoding, 01000.

NOTE

DC1[OVC] can be set to delay the CPU in order to alleviate, but not eliminate, potential overrun situations.

Error information is messaged out in the format shown in Figure 11-33:

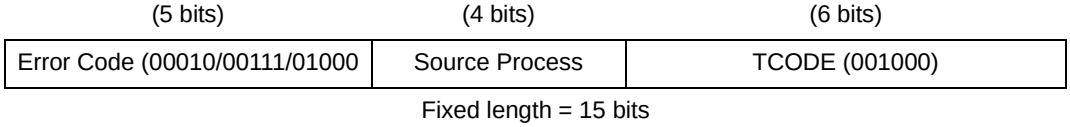


Figure 11-33. Error Message Format

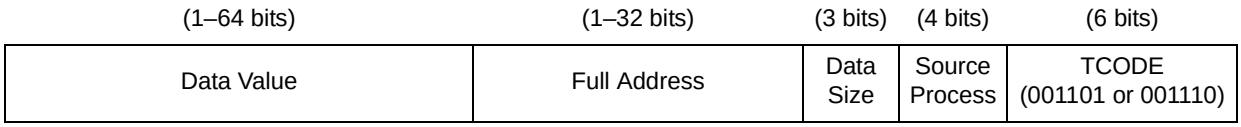
11.8.2.4 Data Trace Synchronization Messages

A data trace write/read with synchronization message is messaged through the auxiliary port, provided data trace is enabled, for the following conditions (see Table 11-22):

- 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 data trace messages have occurred without synchronization since the last with-synchronization message occurred
- Upon assertion of the event-in *nex_evti_b* pin, the first data trace message is a synchronization message if the EIC bits of the DC1 register have enabled this feature.
- Upon data trace write/read after the previous DTM message was lost due to an attempted access to a secure memory location (for SOC's 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 of the following:
  - watchpoint message
  - ownership trace message
  - branch 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 as follows:



Maximum length = 109 bit; Minimum length = 15 bits

Figure 11-34. Data Write/Read with Synchronization Message Format

Exception conditions that result in data trace synchronization are summarized in Table 11-22.

Table 11-22. 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 Nexus3 module are reset. If data trace is enabled, the first data trace message is a data write/read with synchronization message.
Data trace enabled	The first data trace message (after data trace has been enabled) is a synchronization message.
Exit from low power/debug	Upon exit from a low-power mode or debug mode the next data trace message is converted to a data write/read with synchronization message.

Table 11-22. Data Trace Exception Summary (continued)

Exception Condition	Exception Handling
Queue overrun	An error message occurs when a new message cannot be queued due to the message queue being full. The FIFO discards messages until it has completely emptied the queue. Once emptied, an error message is queued. The error encoding indicates which type(s) of messages attempted to be queued while the FIFO was being emptied. The next DTM message in the queue will be a data write/read with synchronization message.
Periodic data trace synchronization	A forced synchronization occurs periodically after 255 data trace messages have been queued. A data write/read with synchronization message is queued. The periodic data trace message counter then resets.
Event in	If the Nexus module is enabled, a *nex_evti_b* assertion initiates a data trace write/read with synchronization message upon the next data write/read (if data trace is enabled and the EIC bits of the DC1 register have enabled this feature).
Attempted access to secure memory	For SOCs which implement security, any attempted read or write to secure memory locations temporarily disables data trace and causes the corresponding DTM to be lost. A subsequent read/write queues a data trace read/write with synchronization message.
Collision priority	All messages have the following priority: WPM → OTM → BTM → DTM. A DTM message which attempts to enter the queue at the same time as a watchpoint message or ownership trace message or branch trace message will be lost. A subsequent read/write queues a data trace read/write with synchronization message.

11.8.3 DTM Operation

11.8.3.1 DTM Queueing

Nexus3 implements a programmable depth queue (a minimum of 32 entries is recommended) for queuing all messages. Messages that enter the queue are transmitted through the auxiliary pins in the order in which they are queued.

NOTE

If multiple trace messages need to be queued simultaneously, watchpoint messages have the highest priority:
WPM → OTM → BTM → DTM.

11.8.3.2 Relative Addressing

The relative address feature is compliant with the *IEEE-ISTO 5001-2003* standard recommendations and is designed to reduce the number of bits transmitted for addresses of data trace messages. Refer to Section 11.7.3.2, “Relative Addressing,” for details.

11.8.3.3 Data Trace Windowing

Data write/read messages are enabled by the RWT1*n* field in the data trace control register, DTC, for each DTM channel. Data trace windowing is achieved through the address range defined by the DTEA and DTSA registers and by DTC[RC1*n*]. All e200z6-initiated read/write accesses which fall inside or outside these address ranges, as programmed, are candidates to be traced.

11.8.3.4 Data Access/Instruction Access Data Tracing

The Nexus3 module is capable of tracing both instruction access data or data access data. Each trace window can be configured for either type of data trace by setting the DI1*n* field within the data trace control register for each DTM channel.

11.8.3.5 e200z6 Bus Cycle Special Cases

Table 11-23. e200z6 Bus Cycle Cases

Special Case	Action
e200z6 bus cycle aborted	Cycle ignored
e200z6 bus cycle with data error ($\overline{\text{TEA}}$)	Data trace message discarded
e200z6 bus cycle completed without error	Cycle captured and transmitted
e200z6 (AHB) bus cycle initiated by Nexus3	Cycle ignored
e200z6 bus cycle is an instruction fetch	Cycle ignored
e200z6 bus cycle accesses misaligned data (across 64-bit boundary)—both first and second transactions within data trace range	First and second cycle captured and two DTMs transmitted
e200z6 bus cycle accesses misaligned data (across 64-bit boundary)—first transaction within data trace range; second transaction out of data trace range	First cycle captured and transmitted; second cycle ignored
e200z6 bus cycle accesses misaligned data (across 64-bit boundary)—first transaction out of data trace range; second transaction within data trace range	First cycle ignored; second cycle captured and transmitted

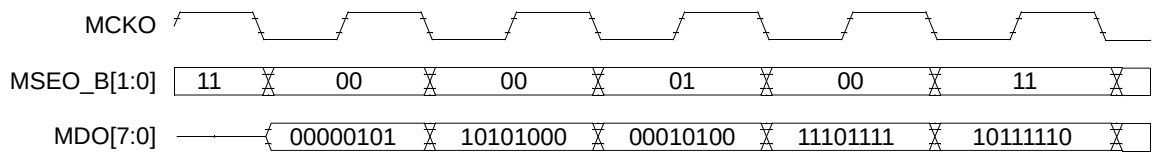
NOTE

For misaligned accesses, crossing 64-bit boundary, the access is broken into two accesses. If both accesses are within the data trace range, two DTMs are sent: one with a size encoding indicating the size of the original access, that is word, and one with a size encoding for the portion which crossed the boundary, that is 3-byte. See Table 3-10 for examples of misaligned accesses.

NOTE

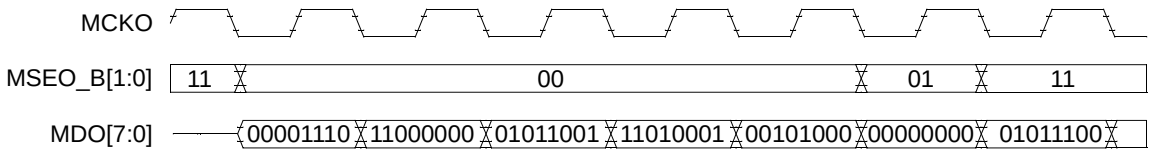
An STM (store) to the cache's store buffer within the data trace range initiates a DTM message. If the corresponding memory access causes an error, a checkstop condition occurs. The debug/development tool should use this indication to invalidate the previous DTM.

11.8.4 Data Trace Timing Diagrams (8 MDO/2 MSEO Configuration)



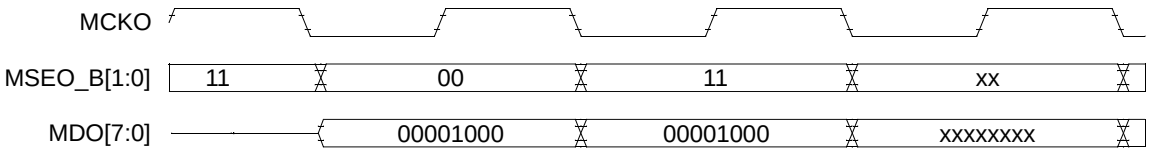
TCODE = 5
Source processor = 0000
Data size = 010 (half word)
Relative address = 0xA5
Write data = 0xBEEF

Figure 11-35. Data Trace—Data Write Message



TCODE = 14
Source processor = 0000
Data size = 000 (byte)
Full access address = 0x01468ACE
Write data = 0x5C

Figure 11-36. Data Trace—Data Read with Synchronization Message



TCODE = 8
Source processor = 0000
Error code = 2 (queue overrun - DTM only)

Figure 11-37. Error Message (Data Trace Only Encoded)

11.9 Watchpoint Support

This section details the watchpoint features of the Nexus3 module.

11.9.1 Overview

The Nexus3 module provides watchpoint messaging by means of the auxiliary pins, as defined by the *IEEE-ISTO 5001-2003* standard.

Nexus3 is not compliant with class 4 breakpoint/watchpoint requirements defined in the standard. The breakpoint/watchpoint control register is not implemented.

11.9.2 Watchpoint Messaging

Enabling watchpoint messaging is done by setting the watchpoint enable bit in the DC1 register. Setting the individual watchpoint sources is supported through the e200z6 Nexus1 module. The e200z6 Nexus1 module is capable of setting multiple address and/or data watchpoints. Please refer to Chapter 10, “Debug Support,” for details on watchpoint initialization.

When these watchpoints occur, a watchpoint event signal from the Nexus1 module causes a message to be sent to the queue to be messaged out. This message includes the watchpoint number indicating which watchpoint caused the message.

The occurrence of any of the e200z6-defined watchpoints can be programmed to assert the event out, *nex_evto_b*, pin for one period of the output clock, *nex_mcko*; see Table 11-28 for details on *nex_evto_b*.

Watchpoint information is messaged out in the format shown in Figure 11-38:

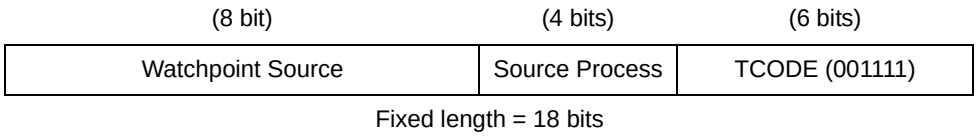


Figure 11-38. Watchpoint Message Format.

Table 11-24. Watchpoint Source Encoding

Watchpoint Source (8-Bits)	Watchpoint Description
0000_0001	e200z6 watchpoint #0 (IAC1 from Nexus1)
0000_0010	e200z6 watchpoint #1 (IAC2 from Nexus1)
0000_0100	e200z6 watchpoint #2 (IAC3 from Nexus1)
0000_1000	e200z6 watchpoint #3 (IAC4 from Nexus1)
0001_0000	e200z6 watchpoint #4 (DAC1 from Nexus1)

Table 11-24. Watchpoint Source Encoding (continued)

Watchpoint Source (8-Bits)	Watchpoint Description
0010_0000	e200z6 watchpoint #5 (DAC2 from Nexus1)
0100_0000	e200z6 watchpoint #6 (DCNT1 from Nexus1)
1000_0000	e200z6 watchpoint #7 (DCNT2 from Nexus1)

11.9.3 Watchpoint Error Message

An error message occurs when the message queue is full and a new message cannot be queued. The FIFO discards messages until it has completely emptied the queue. Once emptied, an error message is queued. The error encoding indicates which types of messages attempted to be queued while the FIFO was being emptied.

If only a watchpoint message attempts to enter the queue while it is being emptied, the error message incorporates the watchpoint-only error encoding, 00110. If an OTM and/or program trace and/or data trace message also attempts to enter the queue while it is being emptied, the error message incorporates error encoding 01000.

NOTE

DC1[OVC] can be set to delay the CPU in order to alleviate, but not eliminate, potential overrun situations.

Error information is messaged out in the format, shown in Figure 11-39:

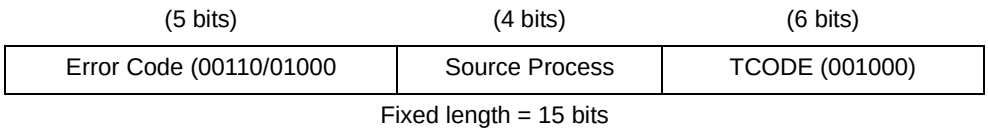


Figure 11-39. Error Message Format

11.9.4 Watchpoint Timing Diagram (2 MDO/1 MSEO Configuration)

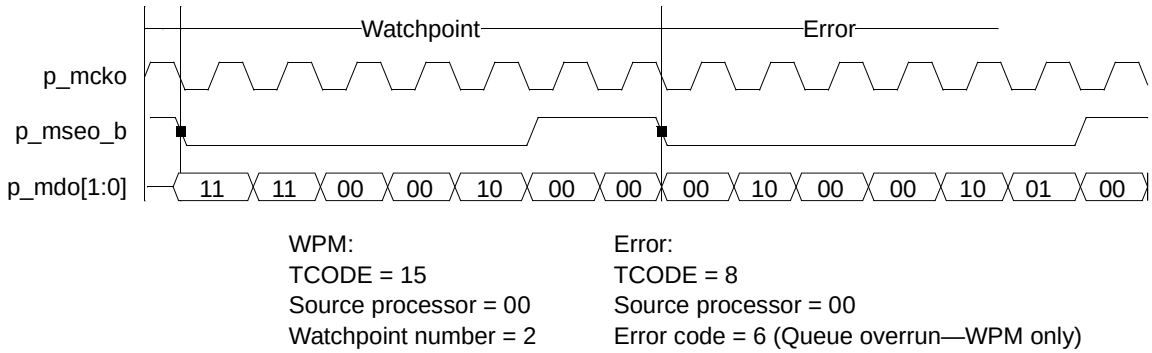


Figure 11-40. Watchpoint Message and Watchpoint Error Message

11.10 Nexus3 Read/Write Access to Memory-Mapped Resources

The read/write access feature allows access to memory-mapped resources through the JTAG/OnCE port. The read/write mechanism supports single as well as block reads and writes to e200z6 AHB resources.

The Nexus3 module is capable of accessing resources on the e200z6 system bus, AHB, with multiple configurable priority levels. Memory-mapped registers and other non-cached memory can be accessed through the standard memory map settings.

All accesses are set up and initiated by the read/write access control/status register, RWCS, as well as RWA and RWD.

Using RWCS, RWA and RWD, memory-mapped e200z6 AHB resources can be accessed through Nexus3. The following sections describe the steps which are required to access memory-mapped resources.

NOTE

Read/write access can only access memory-mapped resources when system reset is cleared. Misaligned accesses are not supported in the e200z6 Nexus3 module.

11.10.1 Single Write Access

NOTE

In the first three steps, the registers are initialized using the access method outlined in Section 11.5, “Nexus3 Register Access through JTAG/OnCE.”

1. Initialize RWA using the Nexus register index of 0x9; see Table 11-7. Configure as shown below:
   — Write address = 0x*nnnn_nnnn* (write address)
2. Initialize RWCS using the Nexus register index of 0x7; see Table 11-7. Configure the fields as shown in Table 11-25:

Table 11-25. Single Write Access Field Settings

Field	Setting
AC (Access control)	1 (indicates start access)
MAP (Map select)	000 (primary memory map)
PR (Access priority)	00 (lowest priority)
RW (Read/write)	1 (write access)
SZ (Word size)	0*nn* (32-bit, 16-bit, 8-bit)
CNT (Access count)	0x0000 or 0x0001 (single access)

NOTE

Access count (CNT) of 0x0000 or 0x0001 performs a single access.

3. Initialize RWD using the Nexus register index of 0xA; see Table 11-7. Configure as shown below:
   — Write data = 0x*nnnn_nnnn* (write data)
4. The Nexus block then arbitrates for the AHB system bus and transfers the data value from the RWD register to the memory-mapped address in RWA. When the access has completed without error (ERR=0), Nexus asserts the *nex_rdy_b* signal (see Table 11-28 for detail on *nex_rdy_b*) and clears RWCS[DV]. This indicates that the device is ready for the next access.

NOTE

Only the *nex_rdy_b* signal and the DV and ERR fields within RWCS provide read/write access status to the external development tool.

11.10.2 Block Write Access (Non-Burst Mode)

1. For a non-burst block write access, follow Steps 1, 2, and 3 outlined in Section 11.10.1, “Single Write Access,” to initialize the registers, but use a value greater than one (0x0001) for RWCS[CNT].
2. The Nexus block then arbitrates for the AHB system bus and transfers the first data value from the RWD register to the memory mapped address in RWA. When the transfer has completed without error (ERR = 0), the address from the RWA register is incremented to the next word size (specified in RWCS[SZ]) and the number from the CNT field is decremented. Nexus then asserts the *nex_rdy_b* pin. This indicates that the device is ready for the next access.
3. Repeat step 3 in Section 11.10.1, “Single Write Access,” until the internal CNT value is zero. When this occurs, RWCS[DV] is cleared to indicate the end of the block write access.

11.10.3 Block Write Access (Burst Mode)

1. For a burst block write access, follow steps 1 and 2 outlined in Section 11.10.1, “Single Write Access” to initialize the registers, using a value of four (double word) for RWCS[CNT] and an RWCS[SZ] value of 0b011, indicating 64-bit access.
2. Initialize the burst data buffer (RWD register) through the access method outlined in Section 11.5, “Nexus3 Register Access through JTAG/OnCE,” using the Nexus register index of 0xA; see Table 11-7.
3. Repeat step 2 until all double-word values are written to the buffer.

NOTE

The data values must be shifted in 32 bits at a time, least-significant bit first (that is, double-word write = two word writes to RWD).

4. The Nexus block then arbitrates for the AHB system bus and transfers the burst data values from the data buffer to the AHB beginning from the memory mapped address in RWA. For each access within the burst, the address from the RWA register is incremented to the next double-word size (as specified in RWCS[SZ]), modulo the length of the burst, and the number from the CNT field is decremented.
5. When the entire burst transfer has completed without error (ERR=0), Nexus3 then asserts the *nex_rdy_b* pin, and RWCS[DV] is cleared to indicate the end of the block write access.

NOTE

The actual RWA and RWCS[CNT] values are not changed when executing a block write access, burst or non-burst. The original values can be read by the external development tool at any time.

11.10.4 Single Read Access

1. Initialize RWA with the access method outlined in Section 11.5, “Nexus3 Register Access through JTAG/OnCE,” using the Nexus register index of 0x9; see Table 11-7. Configure as shown below:
   — Read address = 0x*nnnn_nnnn* (read address)
2. Initialize RWCS with the access method outlined in Section 11.5, “Nexus3 Register Access through JTAG/OnCE,” using the Nexus register index of 0x7, see Table 11-7. Configure the bits as shown in Table 11-26:

Table 11-26. Single Read Access Parameter Settings

Parameter	Settings
Access control (AC)	1 (to indicate start access)
Map select (MAP)	000 (primary memory map)
Access priority (PR)	00 (lowest priority)
Read/write (RW)	0 (read access)
Word size (SZ)	0*nn* (32-bit, 16-bit, 8-bit)
Access count (CNT)	0x0000 or 0x0001(single access)

NOTE

Access count (CNT) of 0x0000 or 0x0001 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=0), Nexus asserts the *nex_rdy_b* pin (see Table 11-28 for details on *nex_rdy_b*) and sets RWCS[DV]. This indicates that the device is ready for the next access.
4. The data can then be read from RWD with the access method outlined in Section 11.5, “Nexus3 Register Access through JTAG/OnCE,” using the Nexus register index of 0xA; see Table 11-7.

NOTE

Only the *nex_rdy_b* signal and the DV and ERR bits within RWCS provide read/write access status to the external development tool.

11.10.5 Block Read Access (Non-Burst Mode)

1. For a non-burst block read access, follow steps 1 and 2 outlined in Section 11.10.4, “Single Read Access” to initialize the registers, but using a value greater than one (0x0001) for RWCS[CNT].
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 = 0), the address from RWA is incremented to the next word size (specified in the SZ field) and the number from the CNT field is decremented. Nexus then asserts the *nex_rdy_b* pin. This indicates that the device is ready for the next access.
3. The data can then be read from RWD with the access method outlined in Section 11.5, “Nexus3 Register Access through JTAG/OnCE,” using the Nexus register index of 0xA, see Table 11-7.
4. Repeat steps 3 and 4 in Section 11.10.4, “Single Read Access,” until the CNT value is zero. When this occurs, RWCS[DV] is set to indicate the end of the block read access.

11.10.6 Block Read Access (Burst Mode)

1. For a burst block read access, follow steps 1 and 2 outlined in Section 11.10.4, “Single Read Access,” to initialize the registers, using a value of four (double-words) for the CNT field and a SZ field indicating 64-bit access in RWCS.
2. The Nexus block then arbitrates for the AHB system bus and the burst read data is transferred from the AHB to the data buffer (RWD register). For each access within the burst, the address from the RWA register is incremented to the next double-word, specified in the SZ field, and the number from the CNT field is decremented.

3. When the entire burst transfer has completed without error (ERR=0), Nexus then asserts the *nex_rdy_b* pin and RWCS[DV] is set to indicate the end of the block read access.
4. The data can then be read from the burst data buffer (RWD register) with the access method outlined in Section 11.5, “Nexus3 Register Access through JTAG/OnCE,” using the Nexus register index of 0xA, see Table 11-7.
5. Repeat step 3 until all double-word values are read from the buffer.

NOTE

The data values must be shifted out 32-bits at a time, least significant bit first, that is double-word read = two word reads from RWD.

NOTE

The actual RWA and CNT values within RWCS are not changed when executing a block read access, burst or non-burst. The original values can be read by the external development tool at any time.

11.10.7 Error Handling

The Nexus3 module handles various error conditions as described in the following sections.

11.10.7.1 AHB Read/Write Error

All address and data errors that occur on read/write accesses to the e200z6 AHB system bus return a transfer error encoding on the *p_hresp[1:0]* signals. If this occurs, the following steps are taken:

1. The access is terminated without retrying, and RWCS[AC] is cleared.
2. RWCS[ERR] is set.
3. The error message is sent, TCODE = 8, indicating read/write error.

11.10.7.2 Access Termination

The following cases are defined for sequences of the read/write protocol that differ from those described in the above sections.

1. If RWCS[AC] is set to start read/write accesses and invalid values are loaded into RWD or RWA, an AHB access error may occur. This is handled as described above.

2. If a block access is in progress, all cycles are not completed, and the RWCS register is written, the original block access is terminated at the boundary of the nearest completed access.
   a) If RWCS[AC] is set, the next read/write access begins and the RWD can be written to/read from.
   b) If RWCS[AC] is cleared, the read/write access is terminated at the nearest completed access. This method can be used to break/early terminate block accesses.

11.10.7.3 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 in the format shown in Figure 11-41:

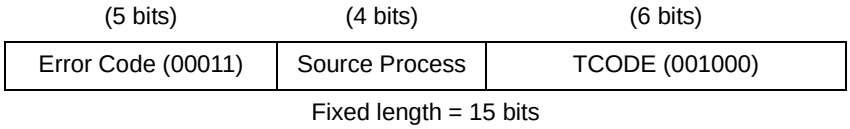


Figure 11-41. Error Message Format

11.11 Nexus3 Pin Interface

This section details the Nexus3 pins and pin protocol.

The Nexus3 pin interface provides the function of transmitting messages from the messages queues to the external tools. It is also responsible for handshaking with the message queues.

11.11.1 Pins Implemented

The Nexus3 module implements one *nex_evti_b* and either one *nex_mseo_b* or two *nex_mseo_b[1:0]*. It also implements a configurable number of *nex_mdo[n:0]* pins, *nex_rdy_b* pin, *nex_evto_b* pin, and one clock output pin, *nex_mcko*. The output pins are synchronized to the Nexus3 output clock, *nex_mcko*.

All Nexus3 input functionality is controlled through the JTAG/OnCE port, in compliance with IEEE 1149.1. (See Section 11.5, “Nexus3 Register Access through JTAG/OnCE,” for details.) The JTAG pins are incorporated as I/O to the e200z6 processor and are further described in Section 10.5.2, “JTAG/OnCE Signals.”

Table 11-27. JTAG Pins for Nexus3

JTAG Pin	I/O	Description of JTAG Pins (included in e200z6 Nexus1)
j_tdo	O	Test data output. *j_tdo* is the serial output for test instructions and data. It is three-stateable and is actively driven in the shift-IR and shift-DR controller states. It changes on the falling edge of *j_tclk*.
j_tdi	I	test data input. *j_tdi* receives serial test instruction and data. TDI is sampled on the rising edge of *j_tclk*.
j_tms	I	Test mode select. Input pin used to sequence the OnCE controller state machine. *j_tms* is sampled on the rising edge of *j_tclk*.
j_tclk	I	Test clock. Input pin used to synchronize the test logic and control register access through the JTAG/OnCE port.
j_trst_b	I	Test reset. Input pin used to asynchronously initialize the JTAG/OnCE controller.

The auxiliary pins are used to send and receive messages and are described in Table 11-28.

Table 11-28. Nexus3 Auxiliary Pins

Auxiliary Pin	I/O	Description of Auxiliary Pins
nex_mcko	O	Message clock out. A free running output clock to development tools for timing of *nex_mdo[n:0]* and *nex_mseo_b[1:0]* pin functions. *nex_mcko* is programmable through the DC1 register.
nex_mdo[n–0]	O	Message data out. Used for OTM, BTM, and DTM. External latching of *nex_mdo[n:0]* occurs on the rising edge of the Nexus3 clock (*nex_mcko*).
nex_mseo_b[1–0]	O	Message start/end out. 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]* occurs on the rising edge of the Nexus3 clock (*nex_mcko*). One- or two-pin MSEO functionality is determined at integration time according to the SOC implementation
nex_rdy_b	O	Ready. Used to indicate to the external tool that the Nexus block is ready for the next read/write access. If Nexus is enabled, this signal is asserted upon successful completion (without error) of an AHB system bus transfer (Nexus read or write) and is held asserted until the JTAG/OnCE state machine reaches the capture_dr state. Upon exit from system reset or if Nexus is disabled, *nex_rdy_b* remains de-asserted
nex_evto_b	O	Event out. An output whose assertion indicates that one of two events has occurred based on the bits in DC1[EOC]. *nex_evto_b* is held asserted for 1 cycle of *nex_mcko*: • One (or more) watchpoints has occurred (from Nexus1) and EOC = 00 • Debug mode was entered (*jd_debug_b* asserted from Nexus1) and EOC = 01
nex_evti_b	I	Event in. An input whose assertion initiates one of two events based on DC1[EIC] (if the Nexus module is enabled at reset): • Program trace and data trace synchronization messages (provided program trace and data trace are enabled and EIC = 00). • Debug request to e200z6 Nexus1 module (provided EIC = 01 and this feature is implemented).

The Nexus auxiliary port arbitration pins are used when the Nexus3 module is implemented in a multiple 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 11.13 for details on Nexus port arbitration.

Table 11-29. Nexus Port Arbitration Signals

Nexus Port Arbitration Pins	Input/ Output	Description of Arbitration Pins
nex_aux_req[1:0]	O	Nexus auxiliary request. Output signals indicating to an SoC level Nexus arbiter a request for access to the shared Nexus auxiliary port in a multiple Nexus implementation. The priority encodings are determined by how many messages are currently in the message queues, see Table 11-31).
nex_aux_busy	O	Nexus auxiliary busy. An output signal to an SoC level Nexus arbiter indicating that the Nexus3 module is currently transmitting its message after being granted the Nexus auxiliary port.
npc_aux_grant	I	Nexus auxiliary grant. An input from the SoC level Nexus port controller (NPC) indicating that the auxiliary port has been granted to the Nexus3 module to transmit its message.
ext_multi_nex_sel	I	Multiple Nexus select. A static signal indicating that the Nexus3 module is implemented within a multiple Nexus environment. If set, port control and arbitration is controlled by the SoC-level arbitration module (NPC).

11.11.2 Pin Protocol

The protocol for the e200z6 processor transmitting messages through the auxiliary pins is accomplished with the MSEO pin function outlined in Table 11-30. 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 Nexus3 clock, *nex_mcko*.

Table 11-30. 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 ones)	00 (or 01)–11–(more ones)
End of variable length packet	0–1–0	00–01
Message transmission	0s	00s
Idle (no message)	1s	11s

Figure 11-42 illustrates the state diagram for single pin MSEO transfers.

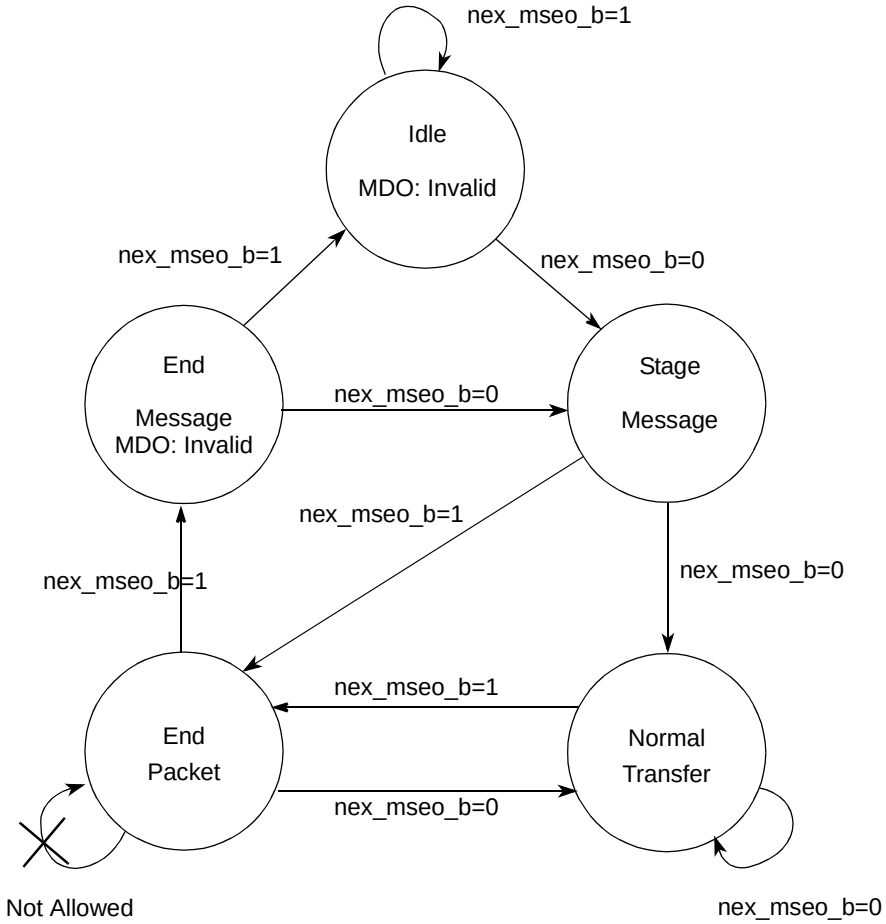


Figure 11-42. Single-Pin MSEO Transfers

Note that the end message state does not contain valid data on *nex_mdo[n:0]*. Also, it is not possible to have two consecutive end packet messages. This implies the minimum packet size for a variable length packet is 2x the number of *nex_mdo[n:0]* pins. This ensures that a false end-of-message state is not entered by emitting two consecutive 1s on *nex_mseo_b* before the actual end of message.

Figure 11-43 illustrates the state diagram for dual-pin MSEO transfers.

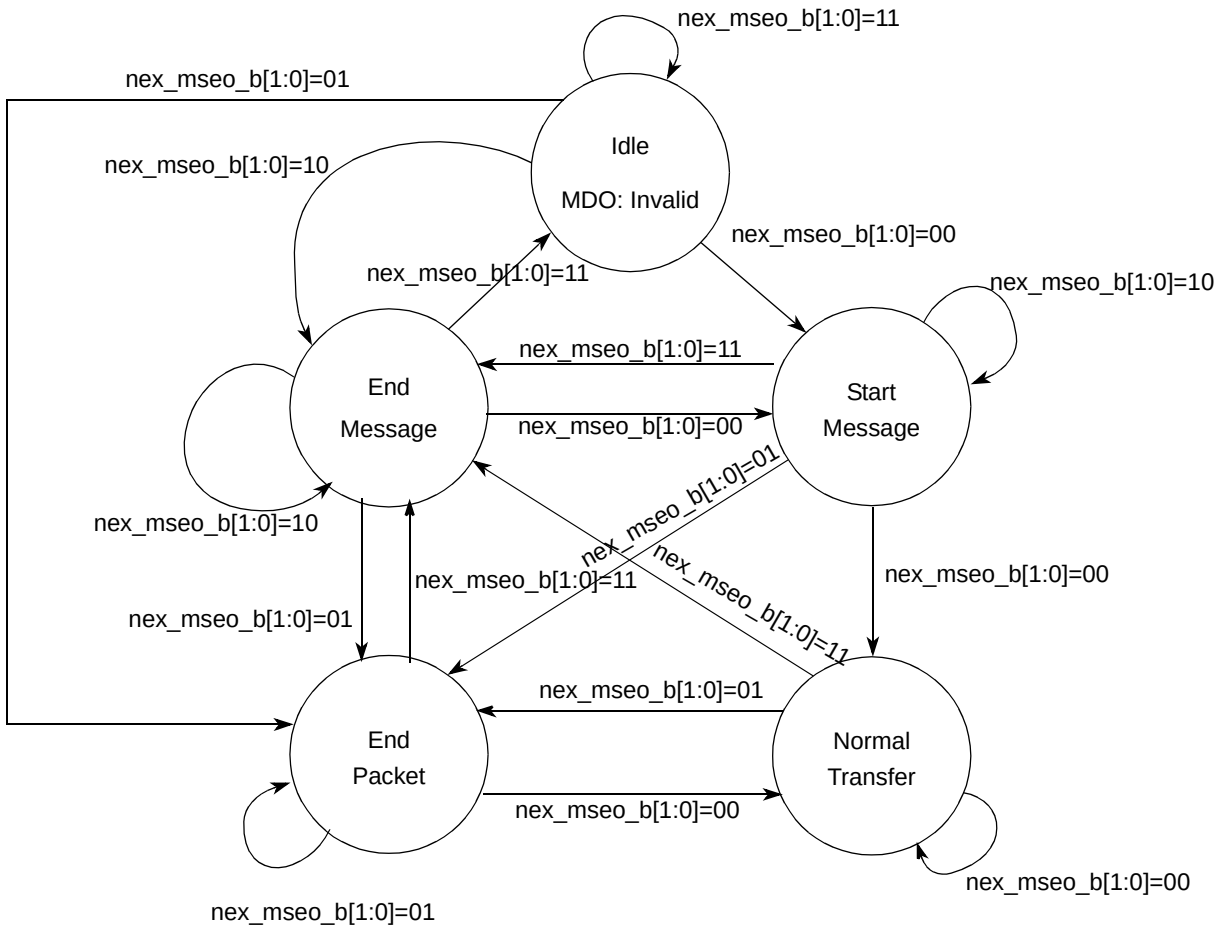


Figure 11-43. Dual-Pin MSEO Transfers

The dual-pin MSEO option is more robust than the single-pin option. Termination of the current message may immediately be followed by the start of the next message on 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.

11.12 Rules for Output Messages

e200z6-based class 3–compliant embedded processors must provide messages through the auxiliary port in a consistent manner as described below:

- A variable-length packet within a message must end on a port boundary.
- A variable-length packet may start within a port boundary only when following a fixed-length packet. If two variable-length 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, the remaining 1 bit of *nex_mdo[n:0]* must be packed with a zero.

11.13 Auxiliary Port Arbitration

In a multiple Nexus environment, the Nexus3 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 11-31 below. The Nexus3 module receives a 1-bit grant signal (*npc_aux_grant*) from the SoC level arbiter. When a grant is received, the Nexus3 module begins transmitting its message following the protocol outlined in Section 11.11.2, “Pin Protocol.” The Nexus3 module maintains control of the port, by asserting the *nex_aux_busy* signal, until the MSEO state machine reaches the end message state.

Table 11-31. 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 half full
—	10	Reserved
High priority	11	Message queue at least half full

11.14 Examples

The following are examples of program trace and data trace messages.

Table 11-32 illustrates an example indirect branch message with 2 MDO/1 MSEO configuration. Table 11-33 illustrates the same example with an 8 MDO/2 MSEO configuration.

Table 11-32. Indirect Branch Message Example (2 MDO/1 MSEO) [1]

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	0	Normal transfer
7	I3	I2	0	Normal transfer
8	I5	I4	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 **Note:** During clock 12, the *nex_mdo[n:0]* pins are ignored in the single-MSEO case.
13	0	0	1	End message
14	T1	T0	0	Start message

1 T0 and S0 are the least significant bits where: Tx = TCODE number (fixed); Sx = Source processor (fixed); Ix = Number of instructions (variable); Ax = Unique portion of the address (variable).

Table 11-33. Indirect Branch Message Example (8 MDO/2 MSEO) [1]

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	I5	I4	I3	I2	I1	I0	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

1 T0 and S0 are the least significant bits where: Tx = TCODE number (fixed); Sx = Source processor (fixed); Ix = Number of instructions (variable); Ax = Unique portion of the address (variable).

Table 11-35 illustrates examples of direct branch messages: one with 2 MDO/1 MSEO, and one with 8 MDO/2 MSEO.

Table 11-34. Direct Branch Message Example (2 MDO/1 MSEO) [1]

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

1 T0 and I0 are the least-significant bits where: Tx = TCODE number (fixed); Sx = Source processor (fixed); Ix = Number of instructions (variable); Ax = Unique portion of the address (variable).

Table 11-35. Direct Branch Message Example (8 MDO / 2 MSEO) [1]

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

1 T0 and I0 are the least-significant bits where: Tx = TCODE number (fixed); Sx = Source processor (fixed); Ix = Number of instructions (variable); Ax = Unique portion of the address (variable).

Table 11-36 illustrates an example data write message with 8 MDO/1 MSEO configuration, and Table 11-37 illustrates the same DWM with 8 MDO/2 MSEO configuration

Table 11-36. Data Write Message Example (8 MDO/1 MSEO) [1]

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	A2	A1	A0	Z2	Z1	Z0	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

1 T0, A0, D0 are the least-significant bits where: Tx = TCODE number (fixed); Sx = Source processor (fixed); Zx = Data size (fixed); Ax = Unique portion of the address (variable); Dx = Write data (variable-8, 16 or 32-bit).

Table 11-37. Data Write Message Example (8 MDO/2 MSEO) [1]

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	A2	A1	A0	Z2	Z1	Z0	S3	S2	0	1	End packet
3	D7	D6	D5	D4	D3	D2	D1	D0	1	1	End packet/end message

[1] T0, A0, D0 are the least-significant bits where: Tx = TCODE number (fixed); Sx = Source processor (fixed); Zx = Data size (fixed); Ax = Unique portion of the address (variable); Dx = Write data (variable - 8, 16 or 32-bit).

11.15 IEEE 1149.1 (JTAG) RD/WR Sequences

This section contains example JTAG/OnCE sequences used to access resources.

11.15.1 JTAG Sequence for Accessing Internal Nexus Registers

Table 11-38. Accessing Internal Nexus3 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 register address
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 register select state)

11.15.2 JTAG Sequence for Read Access of Memory-Mapped Resources

Table 11-39. 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

11.15.3 JTAG Sequence for Write Access of Memory-Mapped Resources

Table 11-40. 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

Index

D

E

F

G

H

I

N

O

P

R

S

T

U

W

X