e200z759n3 Core Reference Manual

Supports:
e200z759n3

e200z759n3CRM
Rev. 2
January 2015

Chapter 1
e200z759n3 Overview

Chapter 2
Register Model

Chapter 3 Instruction Model

Chapter 4 Instruction Pipeline and Execution Timing

Chapter 14
External Core Complex Interfaces

Chapter 1
e200z759n3 Overview

1.1 Overview of the e200z759n3

The e200z759n3 processor family is a set of CPU cores that implement low-cost versions of the *PowerISA 2.06* architecture.

The e200z759n3 is a dual-issue, 32-bit *PowerISA 2.06* compliant design with 64-bit general purpose registers (GPRs). *PowerISA 2.06* floating-point instructions are not supported by e200z759n3 in hardware, but are trapped and may be emulated by software.

An Embedded Floating-point (EFPU2) APU is provided to support real-time single-precision embedded numerics operations using the general-purpose registers.

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

In addition to the base *PowerISA 2.06* instruction set support, the e200z759n3 core also implements the VLE (variable-length encoding) technology, providing improved code density. The VLE technology is further documented in “PowerPC VLE Definition, Version 1.03", a separate document.

The e200z759n3 processor integrates a pair of integer execution units, a branch control unit, instruction fetch unit and load/store unit, and a multi-ported register file capable of sustaining six read and three write operations per clock. Most integer instructions execute in a single clock cycle. Branch target prefetching is performed by the branch unit to allow single-cycle branches in many cases.

The e200z759n3 contains a 16 KB instruction cache (ICache), a 16 KB Data Cache, as well as a Memory Management Unit. A Nexus Class 3+ module is also integrated.

1.1.1 Features

The following is a list of some of the key features of the e200z759n3:

- Dual issue, 32-bit *PowerISA 2.06* compliant CPU
- Implements the VLE APU for reduced code footprint
- In-order execution and retirement
- Precise exception handling
- Branch processing unit
  - Dedicated branch address calculation adder
  - Branch target prefetching using BTB
  - Return Address Stack

- Load/store unit
  - 3 cycle load latency
  - Fully pipelined
  - Big and Little endian support
  - Misaligned access support
- 64-bit General Purpose Register file
- Dual AHB 2.v6 64-bit System buses
- Memory Management Unit (MMU) with 32-entry fully-associative TLB and multiple page size support
- 16Kbyte, 4-Way Set Associative Harvard I and D Caches
- Embedded Floating-point APU (EFPU2) supporting scalar and SIMD single-precision floating-point operations
- Signal Processing Extension (SPE1.1) APU supporting SIMD fixed-point operations, using the 64-bit General Purpose Register file.
- Performance Monitor APU supporting execution profiling
- Nexus Class 3-plus Real-time Development Unit
- Power management
  - Low power design - extensive clock gating
  - Power saving modes: doze, nap, sleep, wait
  - Dynamic power management of execution units, caches and MMUs
- Testability
  - Synthesizeable, MuxD scan design
  - Optional ABIST/MBIST for arrays
  - Built-in Parallel Signature Unit

1.1.2 Microarchitecture summary

The e200z759n3 processor utilizes a ten stage instruction pipeline, with four stages for execution. The Instruction Fetch 0, Instruction Fetch 1, Instruction Fetch 2, Instruction Decode0, Instruction Decode 1/Register file Read/ EA Calc, Execute 0/ Memory Access0, Execute1/Memory Access1, Execute2/Memory Access2, Execute 3, and Register Writeback stages operate in an overlapped fashion, allowing single clock instruction execution for most instructions.

The integer execution units each 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 32x32 Hardware Multiplier array, and result feed-forward hardware. Integer EU1 also supports hardware division.

Most arithmetic and logical operations are executed in a single cycle with the exception of multiply, which is implemented with a 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 dedicated Branch Address adders to minimize delays during change of flow operations. Sequential prefetching is performed to ensure a supply of instructions into the execution pipeline. Branch target prefetching is performed to accelerate taken branches. Prefetched instructions are placed into an instruction buffer.

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

Memory load and store operations are provided for byte, halfword, word (32-bit), and doubleword data with automatic zero or sign extension of byte and halfword 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 condition register consists of eight 4-bit fields that reflect the results of certain operations, such as move, integer and floating-point compare, arithmetic, and logical instructions, 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 SPE1.1 APU supports vector instructions operating on 16 and 32-bit fixed-point data types. The EFPU2 APU supports 32-bit IEEE-754 single-precision floating-point formats, and supports scalar and vector single-precision floating-point operations in a pipelined fashion. The 64-bit general purpose register file is used for source and destination operands, and there is a unified storage model for scalar single-precision floating-point data types of 32-bits and the normal integer type. Low latency fixed-point and floating-point add, subtract, mixed add/subtract, sum, diff, min, max, multiply, multiply-add, multiply-sub, divide, square root, compare, and conversion operations are provided, and most operations can be pipelined.

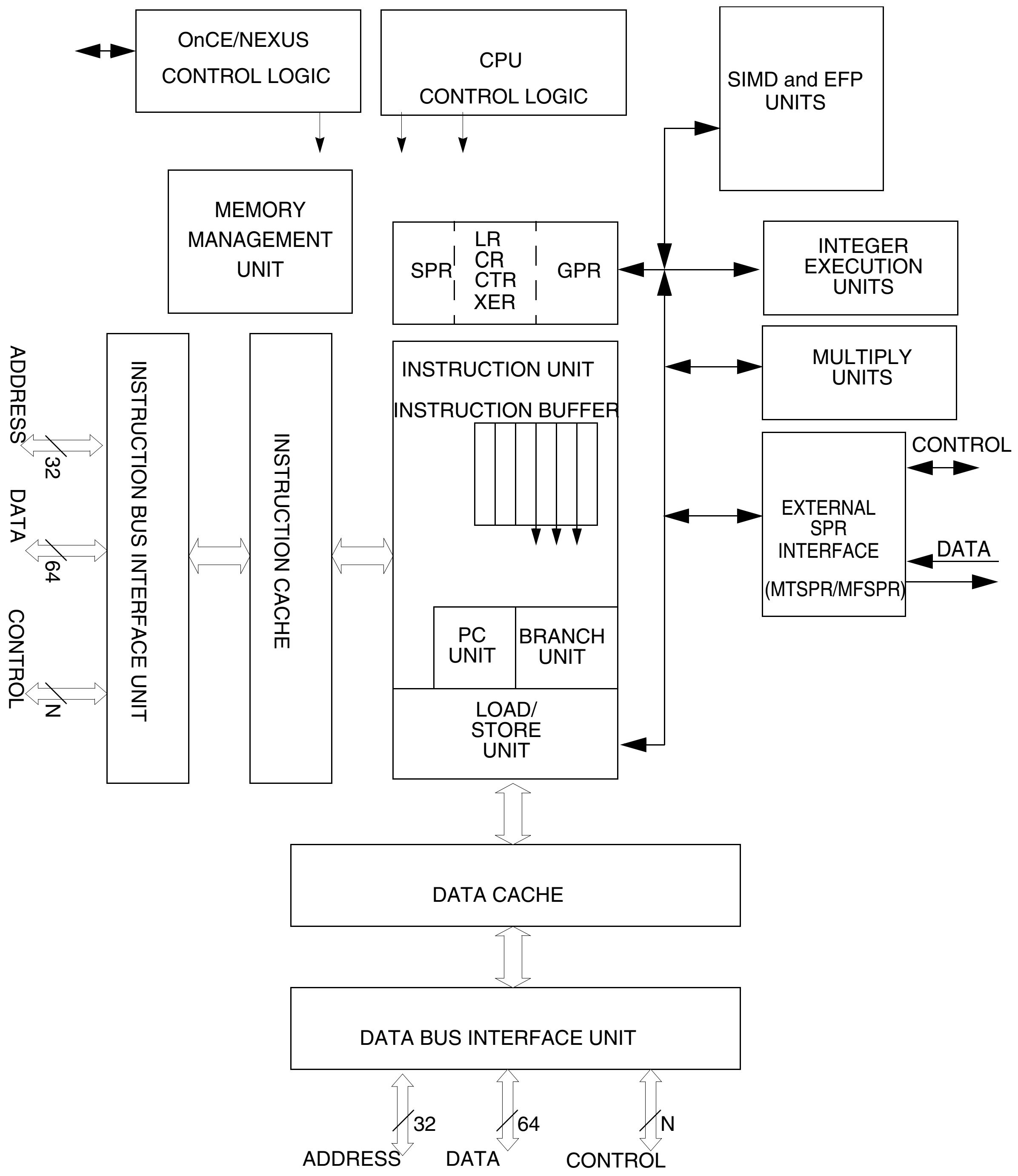


Figure 1-1. e200z759n3 block diagram

1.1.2.1 Instruction unit features

The features of the e200z759n3 Instruction unit are:

- 64-bit path to cache supports fetching of two 32-bit instruction per clock
- Instruction buffer holds up to 10 32-bit instructions
- Dedicated PC incrementer supporting instruction prefetches

- Branch unit with dedicated branch address adder, and branch lookahead logic (BTB) supporting single cycle execution of successfully predicted branches

1.1.2.2 Integer unit features

The e200z759n3 integer units support 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 zero's function
- 32-bit single cycle barrel shifter for static shifts and rotates
- 32-bit mask unit for data masking and insertion
- Divider logic for signed and unsigned divide in 4-15 clocks with minimized execution timing (EU1 only)
- Pipelined 32x32 hardware multiplier array supports $32 \times 32 \rightarrow 32$ multiply with 3 clock latency, 1 clock throughput

1.1.2.3 Load/store unit features

The e200z759n3 load/store unit 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

1.1.2.4 Cache features

The features of the cache are as follows:

- Separate 16 KB, 4-way set-associative instruction and data caches (Harvard architecture)
- Copyback and Writethrough Support
- 8-entry store buffer
- Push buffer
- Linefill buffer
- 32-bit address bus plus attributes and control
- Separate uni-directional 64-bit read data bus and 64-bit write data bus
- Support for cache line locking
- Support for way allocation
- Support for write allocation policies
- Support for tag and data parity
- Support for multi-bit EDC for the ICache
- Correction/auto-invalidation capability for the I and D caches
- Hardware cache coherency support for the data cache

1.1.2.5 MMU Features

The features of the MMU are as follows:

- Virtual memory support
- 32-bit virtual and physical addresses
- 8-bit process identifier
- 32-entry fully-associative TLB
- Multiple page size support from 1 KB to 4 GB
- Entry flush protection

1.1.2.6 e200z759n3 system bus features

The features of the e200z759n3 system bus interface are as follows:

- Independent Instruction and Data interfaces
- AMBA AHB2.v6 protocol
- 32-bit address bus, 64-bit data bus, plus attributes and control
- Data interface provides separate uni-directional 64-bit read and write data buses
- Support for HCLK running at a slower rate than CPU clock

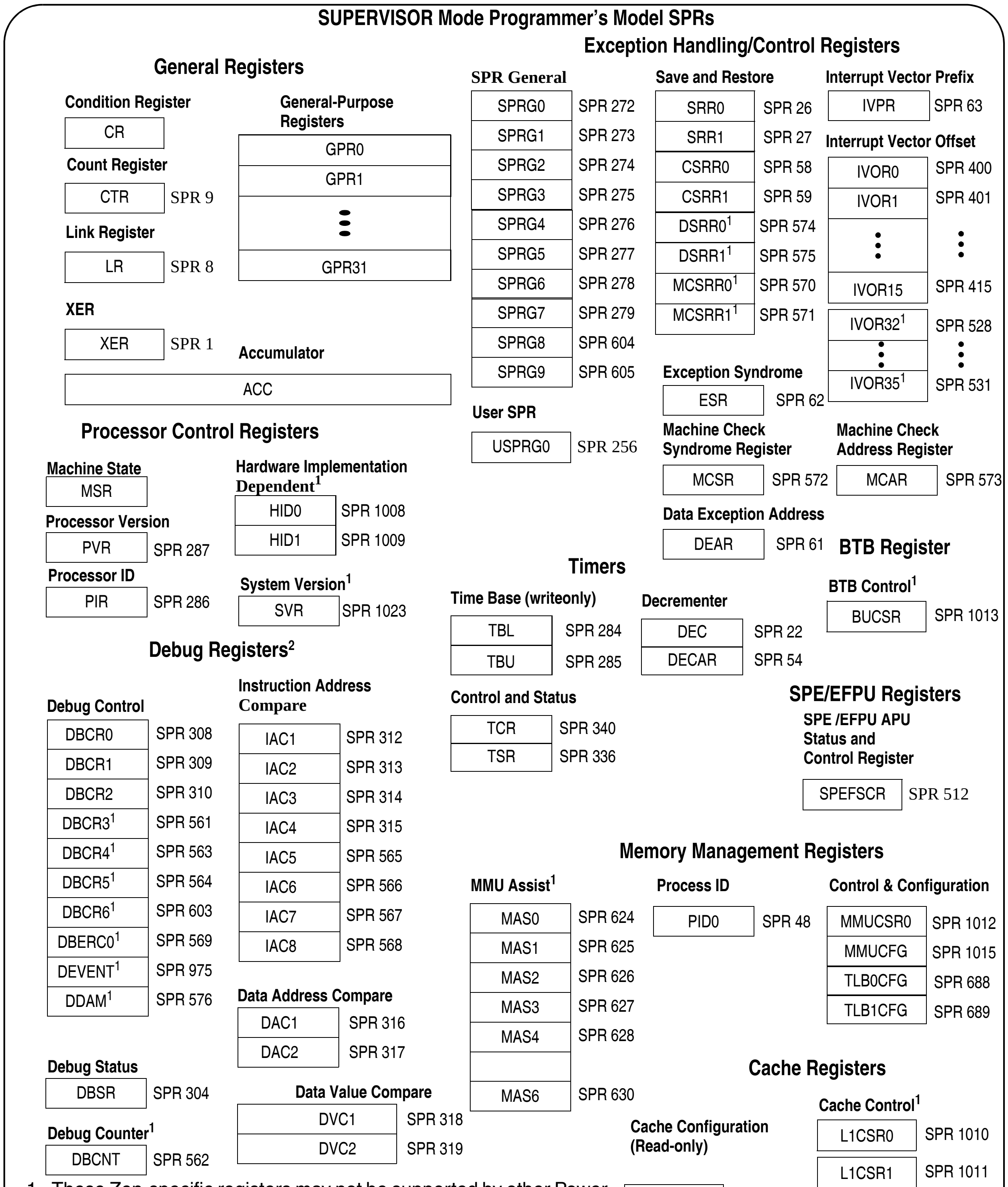
Chapter 2
Register Model

This section describes the registers implemented in the e200z759n3 core. It includes an overview of registers defined by the Power Architecture Book E architecture, highlighting differences in how these registers are implemented in the e200z759n3 core, and provides a detailed description of e200z759n3-specific registers. Full descriptions of the architecture-defined register set are provided in *Book E: Enhanced PowerPC^{tm} Architecture*.

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

e200z759n3 extends the General Purpose Registers to 64-bits for supporting SPE and EFPU 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 1 and Figure 3 show the complete e200z759n3 register set. Figure 1 shows the registers that are accessible while in supervisor mode, and Figure 3 shows the set of registers that are accessible while in user mode. The number to the right of the special-purpose registers (SPRs) is the decimal number used in the instruction syntax to access the register (for example, the integer exception register (XER) is SPR 1).



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

2 - Optional registers defined by the Power Architecture Book-E architecture

3 - Read-only registers

Figure 1. e200z759n3 supervisor mode programmer's model SPRs

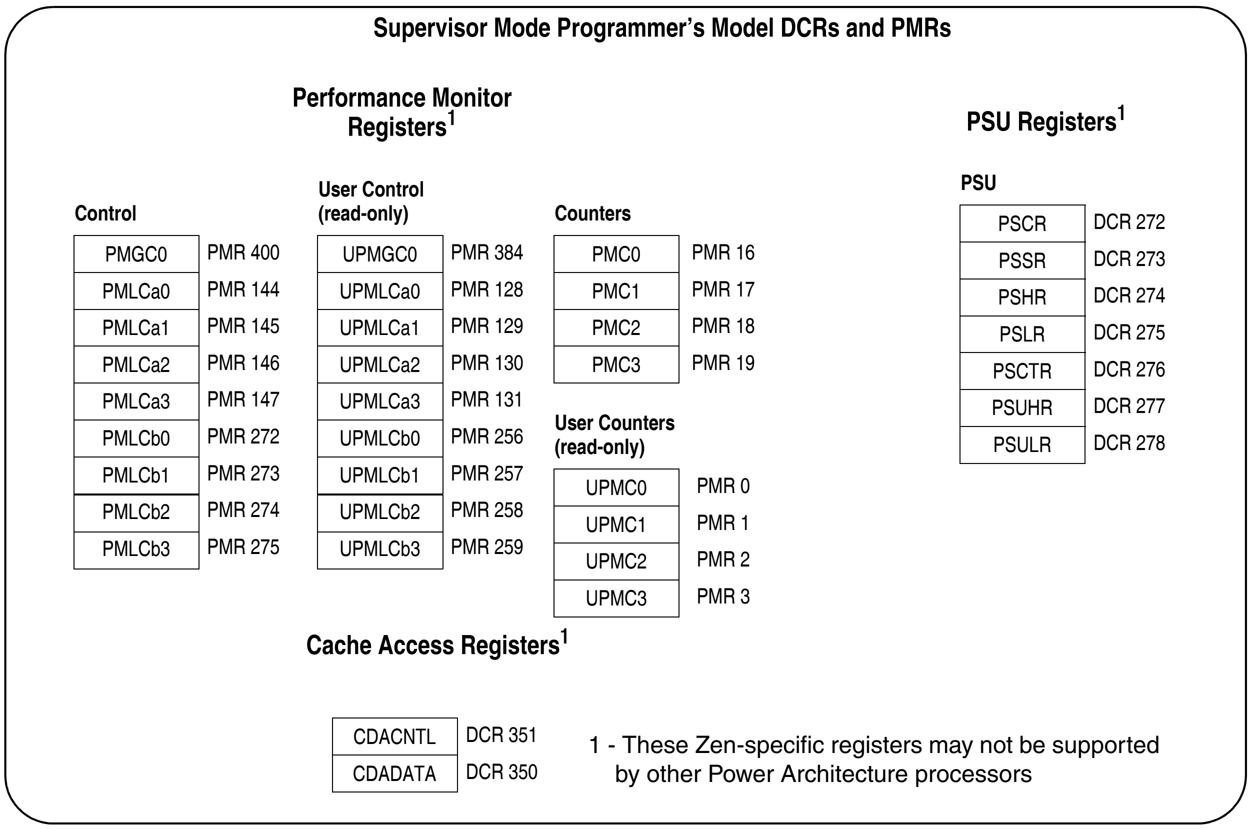


Figure 2. e200z759n3 supervisor mode programmer's model DCRs and PMRs

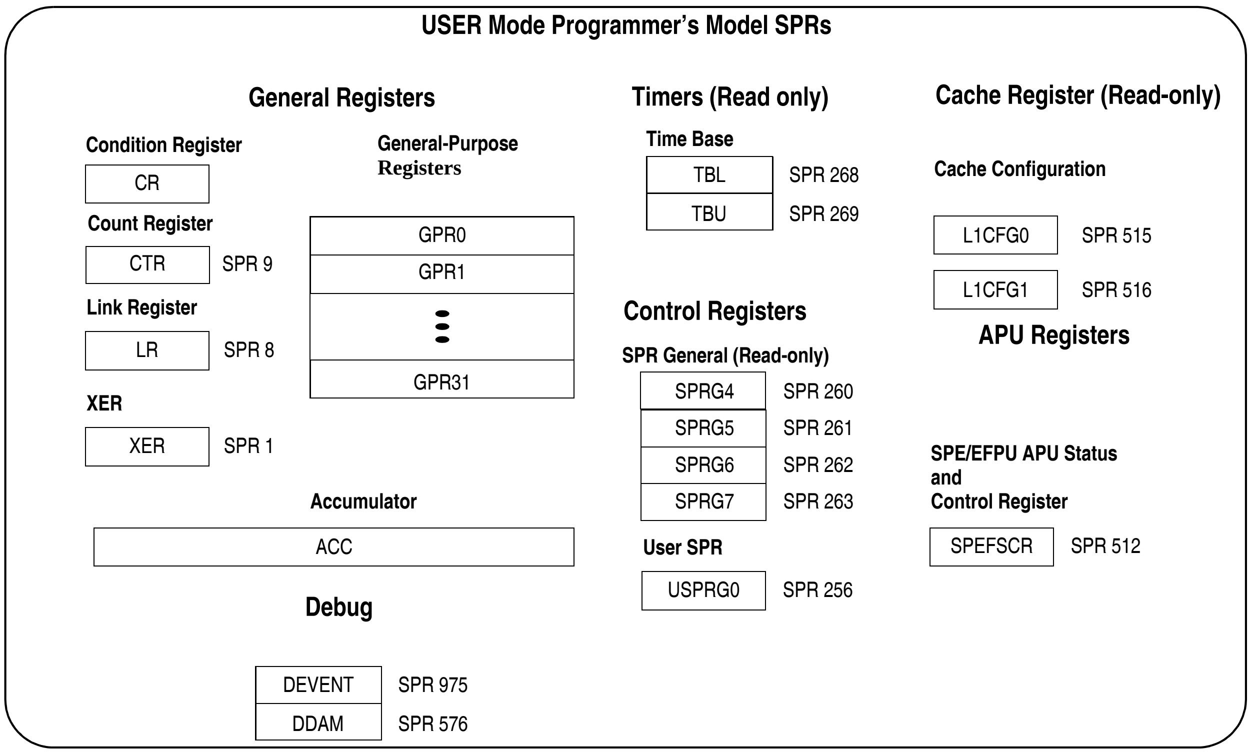


Figure 3. e200z759n3 user mode programmer's model SPRs

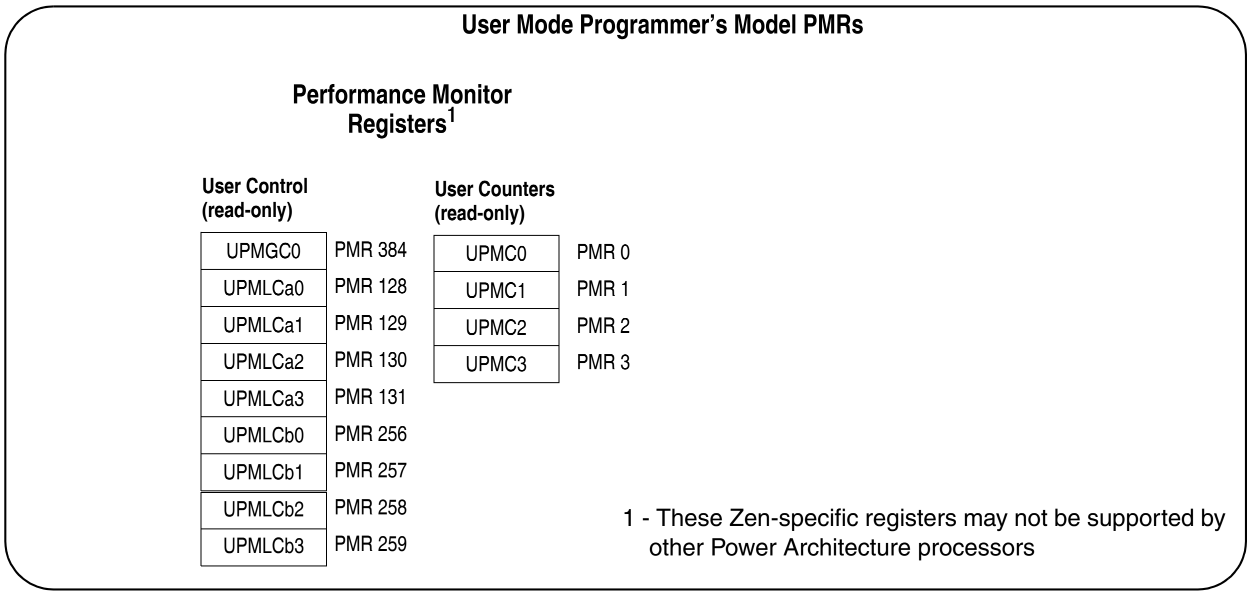


Figure 4. e200z759n3 user mode programmer's model PMRs

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

2.1 PowerPC Book E registers

e200z759n3 supports most of the registers defined by *Book E: Enhanced PowerPC^tm Architecture*. Notable exceptions are the Floating Point registers FPR0-FPR31 and FPSCR. e200z759n3 does not support the Book E Floating Point Architecture in hardware. The General Purpose registers have been extended to 64-bits. The Zen supported Power Architecture Book E registers are described as follows (Zen-specific registers are described in the next sub-section):

- **User-level registers** —The user-level registers can be accessed by all software with either user or supervisor privileges. They include the following:
  - — General-purpose registers (GPRs). The thirty-two 64-bit GPRs (GPR0–GPR31) serve as data source or destination registers for integer instructions and provide data for generating addresses. *PowerPC Book E* instructions affect only the lower 32 bits of the GPRs. SPE and EFP APU instructions are provided, which operate on the entire 64-bit register.
  - — Condition register (CR). The 32-bit CR consists of eight 4-bit fields, CR0–CR7, that reflect results of certain arithmetic operations and provide a mechanism for testing and branching. See "Condition Register (CR)," in Chapter 3, "Branch and Condition Register Operations, *Book E: Enhanced PowerPC^tm Architecture*.

  The remaining user-level registers are SPRs. Note that the Power Architecture architecture provides the **mtspr** and **mfspr** instructions for accessing SPRs.
  - — Integer exception register (XER). The XER indicates overflow and carries for integer operations. See "XER Register (XER)," in Chapter 4, "Integer Operations" of *Book E: Enhanced PowerPC^tm Architecture* for more information.
  - — Link register (LR). The LR provides the branch target address for the Branch [Conditional] to Link Register (**bclr**, **bclrl, se_blr**, **se_blrl**) instructions, and is used to hold the address of the instruction that follows a branch and link instruction, typically used for linking to subroutines. See "Link Register (LR)", in Chapter 3, "Branch and Condition Register Operations" of *Book E: Enhanced PowerPC^tm Architecture*.
  - — Count register (CTR). The CTR holds a loop count that can be decremented during execution of appropriately coded branch instructions. The CTR also provides the branch target address for the Branch [Conditional] to Count Register (**bcctr**, **bcctrl, se_bctr**, **se_bctrl**) instructions. See "Count Register (CTR)", in Chapter 3, "Branch and Condition Register Operations" of *Book E: Enhanced PowerPC^tm Architecture*.
  - — The Time Base facility (TB) consists of two 32-bit registers—Time Base Upper (TBU) and Time Base Lower (TBL). These two registers are accessible in a read-only fashion to user-level software. See "Time Base", in Chapter 8, "Timer Facilities" of *Book E: Enhanced PowerPC^tm Architecture*.
  - — SPRG4-SPRG7. The *PowerPC Book E* architecture defines Software-Use Special Purpose Registers (SPRGs). SPRG4 through SPRG7 are accessible in a read-only fashion by user-level software. Zen does not allow user mode access to the SPRG3 register (defined as implementation dependent by Book E).
  - — USPRG0. The Power Architecture Book E architecture defines User Software-Use Special Purpose Register USPRG0, which is accessible in a read-write fashion by user-level software.

- Supervisor-level registers — In addition to the registers accessible in user mode, Supervisor-level software has access to additional control and status registers used for configuration, exception handling, and other operating system functions. The Power Architecture Book E architecture defines the following supervisor-level registers:
  - **Processor Control registers**
    - Machine State Register (MSR). The MSR defines the state of the processor. The MSR can be modified by the Move to Machine State Register (**mtmsr**), System Call (**sc, se_sc)**, and Return from Exception (**rfi, rfci, rfdi, rfmci, se_rfi, se_rfci, se_rfdi, se_rfmci)** instructions. It can be read by the Move from Machine State Register (**mfmsr)** instruction. When an interrupt occurs, the contents of the MSR are saved to one of the machine state save/restore registers (SRR1, CSRR1, DSRR1, MCSRR1).
    - Processor version register (PVR). This register is a read-only register that identifies the version (model) and revision level of the Power Architecture processor.
    - Processor Identification Register (PIR). This read/write register is provided to distinguish the processor from other processors in the system.
- **Storage Control register**
    - Process ID Register (PID, also referred to as PID0). This register is provided to indicate the current process or task identifier. It is used by the MMU as an extension to the effective address, and by external Nexus 2/3/4 modules for Ownership Trace message generation. *PowerPC Book E* allows for multiple PIDs; e200z759n3 implements only one.
  - **Interrupt Registers**
    - Data Exception Address Register (DEAR). After most Data Storage Interrupts (DSI), or on an Alignment Interrupt or Data TLB Miss Interrupt, the DEAR is set to the effective address (EA) generated by the faulting instruction.
    - SPRG0–SPRG7, USPRG0. The SPRG0–SPRG7 and USPRG0 registers are provided for operating system use. Zen does not allow user mode access to the SPRG3 register (defined as implementation dependent by Book E).
    - Exception Syndrome Register (ESR). The ESR register 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 Vector Offset Registers (IVOR0-IVOR15, IVOR32-IVOR35). These registers together provide the address of the interrupt handler for different classes of interrupts.
    - Save/Restore Register 0 (SRR0). The SRR0 register is used to save machine state on a non-critical interrupt, and contains the address of the instruction at which execution resumes when an **rfi** or **se_rfi** instruction is executed at the end of a non-critical class interrupt handler routine.
    - Critical Save/Restore register 0 (CSRR0). The CSRR0 register is used to save machine state on a critical interrupt, and contains the address of the instruction at which execution resumes when an **rfci** or **se_rfci** instruction is executed at the end of a critical class interrupt handler routine.
    - Save/Restore register 1 (SRR1). The SRR1 register is used to save machine state from the MSR on non-critical interrupts, and to restore machine state when an **rfi** or **se_rfi** executes.

    - Critical Save/Restore register 1 (CSRR1). The CSRR1 register is used to save machine state from the MSR on critical interrupts, and to restore machine state when **rfci** or **se_rfci** executes.
  - **Debug facility registers**
    - Debug Control Registers (DBCR0-DBCR2). These registers provide control for enabling and configuring debug events.
    - Debug Status Register (DBSR). This register contains debug event status.
    - Instruction Address Compare registers (IAC1-IAC4). These registers contain addresses and/or masks that specify Instruction Address Compare debug events.
    - Data address compare registers (DAC1-2). These registers contain addresses and/or masks that specify Data Address Compare debug events.
    - Data value compare registers (DVC1-2). These registers contain data values that specify Data Value Compare debug events.
  - **Timer Registers**
    - Time base (TB). The TB is a 64-bit structure provided for maintaining the time of day and operating interval timers. The TB consists of two 32-bit registers, Time Base Upper (TBU) and Time Base Lower (TBL). The Time Base registers can be written to only by supervisor-level software, but can be read by both user and supervisor-level software.
    - Decrementer register (DEC). This register is a 32-bit decrementing counter that provides a mechanism for causing a decrementer exception after a programmable delay.
    - Decrementer Auto-Reload (DECAR). This register is provided to support the auto-reload feature of the Decrementer.
    - Timer Control Register (TCR). This register controls Decrementer, Fixed-Interval Timer, and Watchdog Timer options.
    - Timer Status Register (TSR). This register contains status on timer events and the most recent Watchdog Timer-initiated processor reset.

2.2 Zen-specific special purpose registers

The Power Architecture Book E architecture allows implementation-specific special purpose registers. Those incorporated in the Zen core are as follows:

- **User-level registers** —The user-level registers can be accessed by all software with either user or supervisor privileges. They include the following:
  - Signal Processing Extension / Embedded Floating-point APU status and control register (SPEFSCR). The SPEFSCR contains all fixed-point and floating-point exception signal bits, exception summary bits, exception enable bits, and rounding control bits needed for compliance with the IEEE 754 standard. See Section 6.2.1, SPE Status and Control Register (SPEFSCR), in Chapter 6, Signal Processing Extension APU (SPE APU)
  - The L1 Cache Configuration registers (L1CFG0, L1CGF1). These read-only registers allows software to query the configuration of the L1 Harvard caches.
- **Supervisor-level registers** — The following supervisor-level registers are defined in Zen in addition to the Power Architecture Book E registers described above:

- Configuration Registers
  - Hardware implementation-dependent register 0 (HID0). This register controls various processor and system functions.
  - Hardware implementation-dependent register 1 (HID1). This register controls various processor and system functions.
- Exception Handling and Control Registers
  - Machine Check Save/Restore register 0 (MCSRR0). The MCSRR0 register is used to save machine state on a Machine Check interrupt, and contains the address of the instruction at which execution resumes when an **rfmci** or **se_rfmci** instruction is executed.
  - Machine Check Save/Restore register 1 (MCSRR1). The MCSRR1 register is used to save machine state from the MSR on Machine Check interrupts, and to restore machine state when an **rfmci** or **se_rfmci** instruction is executed.
  - Machine Check Syndrome register (MCSR). This register provides a syndrome to differentiate between the different kinds of conditions that can generate a Machine Check.
  - Machine Check Address register (MCAR). This register provides an address associated with certain Machine Checks.
  - Debug Save/Restore register 0 (DSRR0). When enabled, the DSRR0 register is used to save the address of the instruction at which execution continues when an **rfdi** or **se_rfdi** instruction executes at the end of a debug interrupt handler routine.
  - Debug Save/Restore register 1 (DSRR1). When enabled, the DSRR1 register is used to save machine status on debug interrupts and to restore machine status when an **rfdi** or **se_rfdi** instruction executes.
  - SPRG8, SPRG9. The SPRG8 and SPRG9 registers are provided for operating system use for the Machine check and Debug APUs.
- Debug Facility Registers
  - Instruction Address Compare registers (IAC5–IAC8). These registers contain addresses and/or masks that are used to specify Instruction Address Compare debug events.
  - Debug Control Register 3–6 (DBCR3, DBCR4, DBCR5, DBCR6)—These registers provides control for debug functions not described in Power Architecture Book E architecture.
  - Debug External Resource Control Register 0 (DBERC0)—This register provides control for debug functions not described in Power Architecture Book E architecture.
  - Debug Counter Register (DBCNT)—This register provides counter capability for debug functions.
- Branch Unit Control and Status Register (BUCSR) controls operation of the BTB
- Cache Registers
  - L1 Cache Configuration Registers (L1CFG0, L1CFG1) is a read-only register that allows software to query the configuration of the L1 Caches.
  - L1 Cache Control and Status Registers (L1CSR0, L1CSR1) control the operation of the L1 Caches such as cache enabling, cache invalidation, cache locking, etc.
  - L1 Cache Flush and Invalidate Registers (L1FINV0, L1FINV1) controls software flushing

and invalidation of the L1 Caches.

- — Memory Management Unit Registers
  - – MMU Configuration Register (MMUCFG) is a read-only register that allows software to query the configuration of the MMU.
  - – MMU Assist (MAS0-MAS4, MAS6) registers. These registers provide the interface to the Zen core from the Memory Management Unit.
  - – MMU Control and Status Register (MMUCSR0) controls invalidation of the MMU.
  - – TLB Configuration Registers (TLB0CFG, TLB1CFG) are read-only registers that allow software to query the configuration of the TLBs.
- — System version register (SVR). This register is a read-only register that identifies the version (model) and revision level of the System that includes a Zen Power Architecture processor.

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

All Zen SPR definitions are compliant with the *Freescale EIS* definitions.

2.3 Zen-specific device control registers

In addition to the SPRs described above, implementations may also choose to implement one or more Device Control Registers (DCRs). e200z759n3 implements a set of device control registers to perform a parallel signature capability in the Parallel Signature Unit (PSU). These registers are described in Section 12.9, Parallel Signature unit.

2.4 Special-purpose register descriptions

2.4.1 Machine State Register (MSR)

A complete description of the Machine State Register (MSR) begins on pg. 37 of *Book E: Enhanced PowerPC^{tm} Architecture v0.99*. The Machine State Register defines the state of the processor. Chapter 7, Interrupts and Exceptions, describes how the MSR is affected when Interrupts occur. The Zen MSR is shown in Figure 5.

0					UCLE	SPE	0						WE	CE	0	EE	PR	FP	ME	FE0	0	DE	FE1	0		IS	DS	0	PMM	RI	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Read/ Write; Reset - 0x0

Figure 5. Machine State Register (MSR)

The MSR bits are defined in Table 2.

Table 2. MSR field descriptions

Bit(s)	Name	Description
0:4 (32:36)	—	Reserved[1]
5 (37)	UCLE	User Cache Lock Enable 0 Execution of the cache locking instructions in user mode (MSR_{PR}=1) disabled; DSI exception taken instead, and ILK or DLK set in ESR. 1 Execution of the cache lock instructions in user mode enabled.
6 (38)	SPE	SPE/EFPU Available 0 Execution of SPE and EFPU APU vector instructions is disabled; SPE/EFPU Unavailable exception taken instead, and SPE bit is set in ESR. 1 Execution of SPE and EFPU APU vector instructions is enabled.
7:12 (39:44)	—	Reserved[1]
13 (45)	WE	Wait State (Power management) enable. 0 Power management is disabled. 1 Power management is enabled. The processor can enter a power-saving mode when additional conditions are present. The mode chosen is determined by the DOZE, NAP, and SLEEP bits in the HID0 register, described in Section 2.4.11, Hardware Implementation Dependent Register 0 (HID0).
14 (46)	CE	Critical Interrupt Enable 0 Critical Input and Watchdog Timer interrupts are disabled. 1 Critical Input and Watchdog Timer interrupts are enabled.
15 (47)	—	Preserved[1]
16 (48)	EE	External Interrupt Enable 0 External Input, Decrementer, and Fixed-Interval Timer interrupts are disabled. 1 External Input, Decrementer, and Fixed-Interval Timer interrupts are enabled.
17 (49)	PR	Problem State 0 The processor is in supervisor mode, can execute any instruction, and can access any resource (e.g. GPRs, SPRs, MSR, etc.). 1 The processor is in user mode, cannot execute any privileged instruction, and cannot access any privileged resource.
18 (50)	FP	Floating-Point Available 0 Floating point unit is unavailable. The processor cannot execute floating-point instructions, including floating-point loads, stores, and moves. 1 Floating Point unit is available. The processor can execute floating-point instructions. **Note:** For e200z759n3, the floating point unit is not supported in hardware, and an Illegal Instruction exception will be generated for attempted execution of *PowerPC Book E* floating point instructions regardless of the setting of FP. FP is ignored, but cleared on exceptions.
19 (51)	ME	Machine Check Enable 0 Asynchronous Machine Check interrupts are disabled. 1 Asynchronous Machine Check interrupts are enabled.
20 (52)	FE0	Floating-point exception mode 0 (not used by Zen)
21 (53)	—	Reserved[1]

Table 2. MSR field descriptions (continued)

Bit(s)	Name	Description
22 (54)	DE	Debug Interrupt Enable 0 Debug interrupts are disabled. 1 Debug interrupts are enabled.
23 (55)	FE1	Floating-point exception mode 1 (not used by Zen)
24 (56)	—	Reserved[1]
25 (57)	—	Preserved[1]
26 (58)	IS	Instruction Address Space 0 The processor directs all instruction fetches to address space 0 (TS=0 in the relevant TLB entry). 1 - The processor directs all instruction fetches to address space 1 (TS=1 in the relevant TLB entry).
27 (59)	DS	Data Address Space 0 The processor directs all data storage accesses to address space 0 (TS=0 in the relevant TLB entry). 1 The processor directs all data storage accesses to address space 1 (TS=1 in the relevant TLB entry).
28 (60)	—	Reserved[1]
29 (61)	PMM	PMM Performance monitor mark bit. System software can set PMM when a marked process is running to enable statistics to be gathered only during the execution of the marked process. MSR_{PR} and MSR_{PMM} together define a state that the processor (supervisor or user) and the process (marked or unmarked) may be in at any time. If this state matches an individual state specified in the Performance Monitor registers PMLCa n, the state for which monitoring is enabled, counting is enabled.
30 (62)	RI	Recoverable Interrupt - This bit is provided for software use to detect nested exception conditions. This bit is cleared by hardware when a Machine Check interrupt is taken.
31 (63)	—	Preserved[1]

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

2.4.2 Processor ID Register (PIR)

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

ID																															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

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

Figure 6. Processor ID Register (PIR)

The PIR fields are defined in Table 3.

Table 3. PIR field descriptions

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

2.4.3 Processor Version Register (PVR)

The Processor Version Register (PVR) contains the processor version number for the CPU core.

1	0	0	0	0	0	0	1	0	1	1	0	Version				MBG Use				Minor Rev				Major Rev				MBG ID			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

SPR - 287; Read-only

Figure 7. Processor Version Register (PVR)

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

Table 4. PVR field descriptions

Bits	Name	Description
0:3	Manuf. ID	These bits identify the Manufacturer ID. Freescale is 4`b1000.
4:5	—	These bits are reserved (00)
6:11	Type	These bits identify the processor type. Zen Z7 is 6`b010110.
12:15	Version	These bits identify the version of the processor and inclusion of optional elements.For e200z759n3, these are tied to 4`b1001.

Table 4. PVR field descriptions (continued)

Bits	Name	Description
16:19	MBG Use	These bits are allocated for use by Freescale Business Groups to distinguish different system variants, and are provided by the **p_pvrin[16:19]** input signals.
20:23	Minor Rev	These bits distinguish between implementations of the version, and are provided by the **p_pvrin[20:23]** input signals.
24:27	Major Rev	These bits distinguish between implementations of the version, and are provided by the **p_pvrin[24:27]** input signals.
28:31	MBG ID	These bits identify the Freescale Business Group responsible for a particular mask set, and are provided by the **p_pvrin[28:31]** input signals. MBG value of 4`b0000 is reserved.

2.4.4 System Version Register (SVR)

The System Version Register (SVR) contains system version information for a Zen-based SoC.

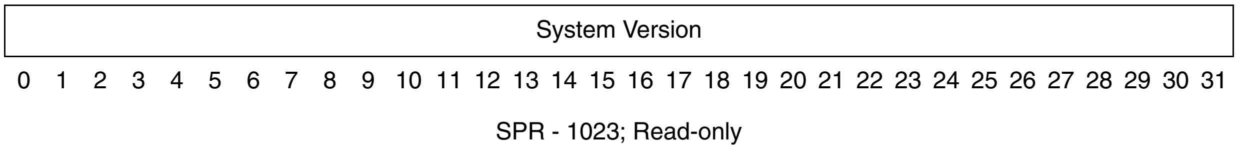


Figure 8. System Version Register (SVR)

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

Table 5. SVR field descriptions

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

2.4.5 Integer Exception Register (XER)

A complete description of the Integer Exception Register (XER) begins on pg. 51 of *Book E: Enhanced PowerPC^{tm} Architecture v0.99*.The XER bit assignments are shown in Figure 9.

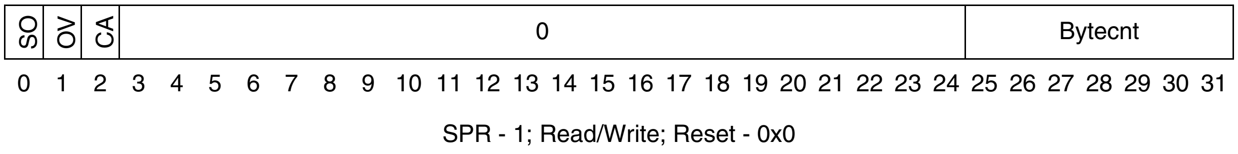


Figure 9. Integer Exception Register (XER)

The XER fields are defined in Table 6.

Table 6. XER field descriptions

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

NOTES:
[1] These bits are not implemented, will be read as zero, and writes are ignored.
[2] These bits are implemented to support emulation of the string instructions.

2.4.6 Exception Syndrome Register

A complete description of the Exception Syndrome Register (ESR) begins on pg. 142 of *Book E: Enhanced PowerPC*^{tm} *Architecture v0.99*. The Exception Syndrome Register (ESR) provides a *syndrome* to differentiate between exceptions that can generate the same interrupt type. Zen adds some implementation specific bits to this register, as seen in Figure 10

0				PIL	PPR	PTR	FP	ST	0	DLK	ILK	AP	PUO	BO	PIE	0								SPE	0	VLEMI	0			MIF	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

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

Figure 10. Exception Syndrome Register (ESR)

The ESR fields are defined in Table 7.

Table 7. ESR field descriptions

Bits	Name	Description	Associated interrupt type
0:3 (32:35)	—	Allocated[1]	—
4 (36)	PIL	Illegal Instruction exception (For e200z759n3, PIL used for all illegal/unimps)	Program
5 (37)	PPR	Privileged Instruction exception	Program
6 (38)	PTR	Trap exception	Program

Table 7. ESR field descriptions (continued)

Bits	Name	Description	Associated interrupt type
7 (39)	FP	Floating-point operation	Alignment Data Storage Data TLB Program
8 (40)	ST	Store operation	Alignment Data Storage Data TLB
9 (41)	—	Reserved[2]	—
10 (42)	DLK	Data Cache Locking	Data Storage
11 (43)	ILK	Instruction Cache Locking	Data Storage
12 (44)	AP	Auxiliary Processor operation (Currently unused in Zen)	Alignment Data Storage Data TLB Program
13 (45)	PUO	Unimplemented Operation exception (Not used by e200z759n3, PIL used for all illegal/unimps)	Program
14 (46)	BO	Byte Ordering exception Mismatched Instruction Storage exception	Data Storage Instruction Storage
15 (47)	PIE	Program Imprecise exception (Reserved)	Currently unused in Zen
16:23 (48:55)	—	Reserved[2]	—
24 (56)	SPE	SPE/EFPU APU Operation	SPE/EFPU Unavailable EFPU Floating-point Data Exception EFPU Floating-point Round Exception Alignment Data Storage Data TLB
25 (57)	—	Allocated[1]	—

Table 7. ESR field descriptions (continued)

Bits	Name	Description	Associated interrupt type
26 (58)	VLEMI	VLE Mode Instruction	SPE/EFPU Unavailable EFPU Floating-point Data Exception EFPU Floating-point Round Exception Data Storage Data TLB Instruction Storage Alignment Program System Call
27:29 (59:61)	—	Allocated[1]	—
30 (62)	MIF	Misaligned Instruction Fetch	Instruction Storage Instruction TLB
31 (63)	—	Allocated[1]	—

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

2.4.6.1 PowerPC VLE mode instruction syndrome

The ESR_{VLEMI} bit is provided to indicate that an interrupt was caused by a PowerPC VLE instruction. This syndrome bit is set on an exception associated with execution or attempted execution of a PowerPC VLE instruction. This bit is updated for the interrupt types indicated in Table 7.

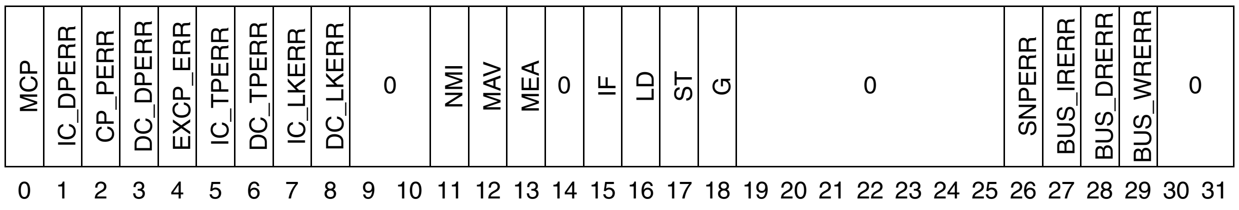
2.4.6.2 Misaligned instruction fetch syndrome

The ESR_{MIF} bit is provided to indicate that an Instruction Storage Interrupt was caused by an attempt to fetch an instruction from a BookE page that was not aligned on a word boundary. The fetch may have been caused by execution of a Branch class instruction from a VLE page to a non-VLE page, a Branch to LR instruction with LR[62]=1, a Branch to CTR instruction with CTR[62]=1, execution of an ***rfi*** or ***se_rfi*** instruction with SRR0[62]=1, execution of an ***rfci*** or ***se_rfci*** instruction with CSRR0[62]=1, execution of an ***rfdi*** or ***se_rfdi*** instruction with DSRR0[62]=1, or execution of an ***rfmci*** or ***se_rfmci*** instruction with MCSRR0[62]=1, where the destination address corresponds to an instruction page that is not marked as a PowerPC VLE page.

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

2.4.7 Machine Check Syndrome Register (MCSR)

When the core complex takes a machine check interrupt, it updates the Machine Check Syndrome register (MCSR) to differentiate between machine check conditions. The MCSR is shown in Figure 11.



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

Figure 11. Machine Check Syndrome Register (MCSR)

Table 8 describes MCSR fields. The MCSR indicates the source of a machine check condition. When an "Async Mchk" or "Error Report" syndrome bit in the MCSR is set, the core complex asserts **p_mcp_out** for system information. Note that the bits in the MCSR are implemented as "write '1' to clear", so software must write ones into those bit positions it wishes to clear, typically by writing back what was originally read. See Section 7.7.2, Machine Check interrupt (IVOR1), for more details of the MCSR settings.

Table 8. MCSR field descriptions

Bit	Name	Description	Exception type[1]	Recoverable
0 (32)	MCP	Machine check input pin	Async Mchk	Maybe
1 (33)	IC_DPERR	Instruction Cache data array parity error	Async Mchk	Precise
2 (34)	CP_PERR	Data Cache push parity error	Async Mchk	Unlikely
3 (35)	DC_DPERR	Data Cache data array parity error	Async Mchk	Maybe
4 (36)	EXCP_ERR	ISI, ITLB, or Bus Error on first instruction fetch for an exception handler	Async Mchk	Precise
5 (37)	IC_TPERR	Instruction Cache Tag parity error	Async Mchk	Precise
6 (38)	DC_TPERR	Data Cache Tag parity error	Async Mchk	Maybe
7 (39)	IC_LKERR	Instruction Cache Lock error Indicates a cache control operation or invalidation operation invalidated one or more locked lines in the ICache.	Status	—
8 (40)	DC_LKERR	Data Cache Lock error Indicates a cache control operation or instruction invalidation operation invalidated one or more locked lines in the DCache.	Status	—

Table 8. MCSR field descriptions (continued)

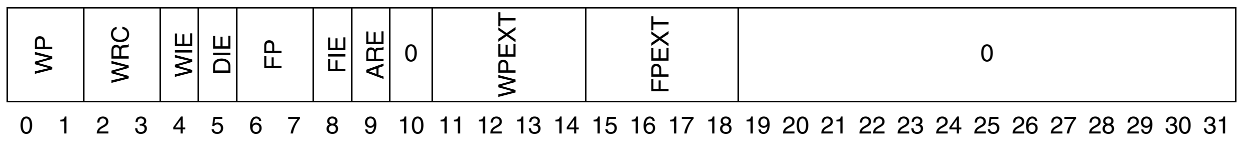
Bit	Name	Description	Exception type[1]	Recoverable
9:10 (41:42)	—	Reserved, should be cleared.		—
11 (43)	NMI	NMI input pin	NMI	—
12 (44)	MAV	MCAR Address Valid Indicates that the address contained in the MCAR was updated by hardware to correspond to the first detected Async Mchk error condition	Status	—
13 (45)	MEA	MCAR holds Effective Address If MAV=1,MEA=1 indicates that the MCAR contains an effective address and MEA=0 indicates that the MCAR contains a physical address	Status	—
14 (46)	—	Reserved, should be cleared.		—
15 (47)	IF	Instruction Fetch Error Report An error occurred during the attempt to fetch an instruction. MCSRR0 contains the instruction address.	Error Report	Precise
16 (48)	LD	Load type instruction Error Report An error occurred during the attempt to execute the load type instruction located at the address stored in MCSRR0.	Error Report	Precise
17 (49)	ST	Store type instruction Error Report An error occurred during the attempt to execute the store type instruction located at the address stored in MCSRR0.	Error Report	Precise
18 (50)	G	Guarded instruction Error Report An error occurred during the attempt to execute the load or store type instruction located at the address stored in MCSRR0 and the access was guarded and encountered an error on the external bus.	Error Report	Precise
19:25 (51:57)	—	Reserved, should be cleared.		—
26 (58)	SNPERR	Snoop Lookup Error An error occurred during certain snoop operations. This is typically due to a data cache tag parity error, in which case DC_TPERR will also be set.	Async Mchk	Unlikely
27 (59)	BUS_IRERR	Read bus error on Instruction fetch or linefill	Async Mchk	Precise if data used
28 (60)	BUS_DRERR	Read bus error on data load or linefill	Async Mchk	Precise if data used
29 (61)	BUS_WRERR	Write bus error on store or cache line push	Async Mchk	Unlikely
30:31 (62:63)	—	Reserved, should be cleared.		—

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

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

2.4.8 Timer Control Register (TCR)

The Timer Control Register (TCR) provides control information for the CPU timer facilities. A complete description of the TCR begins on pg. 182 of *Book E: Enhanced PowerPC^{tm} Architecture v0.99*. The TCR_{WRC} field functions are defined to be implementation-dependent and are described below. In addition, the Zen core implements two fields not specified in Book E, TCR_{WPEXT} and TCR_{FPEXT}. The TCR is shown in Figure 12.



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

Figure 12. Timer Control Register (TCR)

The TCR fields are defined in Table 9.

Table 9. TCR field descriptions

Bits	Name	Description
0:1 (32:33)	WP	Watchdog Timer Period When concatenated with WPEXT, specifies one of 64 bit locations of the time base used to signal a watchdog timer exception on a transition from 0 to 1. TCRwpext[0–3],TCRwp[0–1] == 6'b000000 selects TBU[0] TCRwpext[0–3],TCRwp[0–1] == 6'b111111 selects TBL[31]

Table 9. TCR field descriptions (continued)

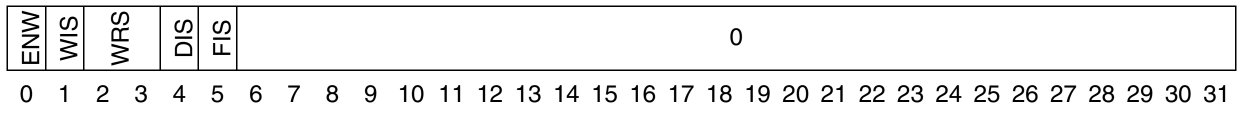
Bits	Name	Description
2:3 (34:35)	WRC	Watchdog Timer Reset Control 00 No Watchdog Timer reset will occur 01 Assert watchdog reset status output 1 (**p_wrs[1]**) on second time-out of Watchdog Timer 10 Assert watchdog reset status output 0 (**p_wrs[0]**) on second time-out of Watchdog Timer 11 Assert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

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

2.4.9 Timer Status Register (TSR)

The Timer Status Register (TSR) provides status information for the CPU timer facilities. A complete description of the TSR begins on pg. 184 of *Book E: Enhanced PowerPC^{tm} Architecture v0.99*. The

TSR$_{WRS}$ field is defined to be implementation-dependent and is described below. The TSR is shown in Figure 13.



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

Figure 13. Timer Status Register (TSR)

The TSR fields are defined in Table 10.

Table 10. TSR field descriptions

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

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

NOTE

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

2.4.10 Debug registers

The Debug facility registers are described in Chapter 12, Debug Support.

2.4.11 Hardware Implementation Dependent Register 0 (HID0)

The HID0 register is a Zen implementation dependent register used for various configuration and control functions.The HID0 register is shown in Figure 14.

EMCP	0	DOZE	NAP	SLEEP	0	ICR	NHR	0	TBEN	SEL_TBCLK	DCLREE	DCLRCE	CICLRDE	MCCLRDE	DAPUEN	0	NOPTI
0	1 2 3 4 5 6 7	8	9	10	11 12 13	14	15	16	17	18	19	20	21	22	23	24 25 26 27 28 29 30	31

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

Figure 14. Hardware Implementation Dependent Register 0 (HID0)

The HID0 fields are defined in Table 11.

Table 11. HID0 fiels descriptions

Bits	Name	Description
0 [32]	EMCP	Enable machine check pin (**p_mcp_b**) 0 **p_mcp_b** pin is disabled. 1 **p_mcp_b** pin is enabled. Asserting **p_mcp_b** causes a machine check interrupt to be reported.
1:7 [33:39]	—	Reserved[1]
8 [40]	DOZE	Configure for Doze power management mode 0 Doze mode is disabled. 1 Doze mode is enabled. Doze mode is invoked by setting MSR_{WE} while this bit is set.
9 [41]	NAP	Configure for Nap power management mode 0 Nap mode is disabled. 1 Nap mode is enabled. Nap mode is invoked by setting MSR_{WE} while this bit is set.
10 [42]	SLEEP	Configure for Sleep power management mode 0 Sleep mode is disabled. 1 Sleep mode is enabled. Sleep mode is invoked by setting MSR_{WE} while this bit is set. Only one of DOZE, NAP, or SLEEP should be set for proper operation.
11:13 [43:45]	—	Reserved[1]
14 [46]	ICR	Interrupt Inputs Clear Reservation 0 External Input, Critical Input, and Non-Maskable Interrupts do not affect reservation status. 1 External Input, Critical Input, and Non-Maskable Interrupts clear an outstanding reservation.

Table 11. HID0 fiels descriptions (continued)

Bits	Name	Description
15 [47]	NHR	Not hardware reset 0 Indicates to a reset exception handler that a reset occurred if software had previously set this bit. 1 Indicates to a reset exception handler that no reset occurred if software had previously set this bit. Provided for software useset anytime by software, cleared by reset.
16 [48]	—	Reserved[1]
17 [49]	TBEN	TimeBase Enable 0 TimeBase is disabled. 1 TimeBase is enabled.
18 [50]	SEL_TBCLK	Select TimeBase Clock 0 TimeBase is based on processor clock. 1 TimeBase is based on **p_tbclk** input. This bit controls the clock source for the TimeBase. Altering this bit must be done while the time base is disabled to preclude glitching of the counter. Timer interrupts should be disabled prior to alteration, and the TBL and TBU registers re-initialized following a change of TimeBase clock source.
19 [51]	DCLREE	Debug Interrupt Clears MSR_{EE} 0 MSR_{EE}unaffected by Debug Interrupt. 1 MSR_{EE} cleared by Debug Interrupt. This bit controls whether Debug interrupts force External Input interrupts to be disabled, or whether they remain unaffected.
20 [52]	DCLRCE	Debug Interrupt Clears MSR_{CE} 0 MSR_{CE} unaffected by Debug Interrupt. 1 MSR_{CE} cleared by Debug Interrupt. This bit controls whether Debug interrupts force Critical interrupts to be disabled, or whether they remain unaffected.
21 [53]	CICLRDE	Critical Interrupt Clears MSR_{DE} 0 MSR_{DE} unaffected by Critical class interrupt. 1 MSR_{DE} cleared by Critical class interrupt. This bit controls whether certain Critical interrupts (Critical Input, Watchdog Timer) force Debug interrupts to be disabled, or whether they remain unaffected. Machine Check interrupts have a separate control bit. 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 will be executed first. In this case, $DSRR0_{DE}$ will have been cleared, such that after returning from the debug handler, the Critical interrupt handler will not be run with MSR_{DE} enabled.
22 [54]	MCCLRDE	Machine Check Interrupt Clears MSR_{DE} 0 MSR_{DE} unaffected by Machine Check interrupt. 1 MSR_{DE} cleared by Machine Check interrupt. This bit controls whether Machine Check interrupts force Debug interrupts to be disabled, or whether they remain unaffected.

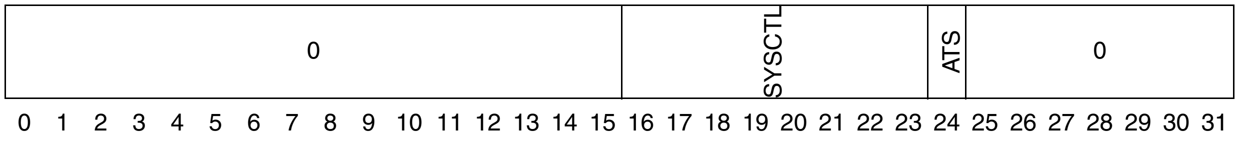
Table 11. HID0 fiels descriptions (continued)

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

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

2.4.12 Hardware Implementation Dependent Register 1 (HID1)

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



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

Figure 15. Hardware Implementation Dependent Register 1 (HID1)

The HID1 fields are defined in Table 12.

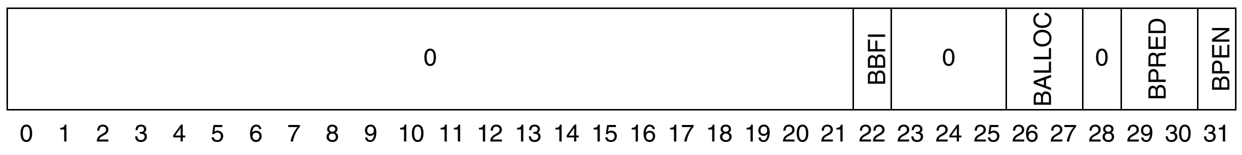
Table 12. HID 1 field descriptions

Bits	Name	Description
0:15 [32:47]	—	Reserved[1]
16:23 [48:56]	SYSCTL	System Control These bits are reflected on the outputs of the p_hid1_sysctl[0:7] output signals for use in controlling the system. They may need external synchronization.
24 [56]	ATS	Atomic status (read-only) Indicates state of the reservation bit in the load/store unit. See Section 3.5, Memory synchronization and reservation instructions for more detail.
25:31 [57:63]	—	Reserved[1]

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

2.4.13 Branch Unit Control and Status Register (BUCSR)

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



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

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

The BUCSR fields are defined in Table 13.

Table 13. BUCSR field descriptions

Bits	Name	Description
0:21 [32:53]	—	Reserved[1]
22 [54]	BBFI	Branch target buffer flash invalidate. When written to a '1', BBFI flash clears the valid bit of all entries in the branch buffer; clearing occurs regardless of the value of the enable bit (BPEN). BBFI is always read as 0.
23:25 [55:57]	—	Reserved[1]

Table 13. BUCSR field descriptions (continued)

Bits	Name	Description
26:27 [58:59]	BALLOC	Branch Target Buffer Allocation Control 00 - Branch Target Buffer allocation for all branches is enabled. 01 - Branch Target Buffer allocation is disabled for backward branches. 10 - Branch Target Buffer allocation is disabled for forward branches. 11 - Branch Target Buffer allocation is disabled for both branch directions. This field controls BTB allocation for branch acceleration when BPEN = 1. BTB hits are not affected by the settings of this field. For branches with “AA’ = ‘1’, the MSB of the displacement field is still used to indicate forward/backward, even though the branch is absolute.
28 [60]	—	Reserved[1]
29:30 [61:62]	BPRED	Branch Prediction Control (Static) 00 - Branch predicted taken on BTB miss for all branches. 01 - Branch predicted taken on BTB miss only for forward branches. 10 - Branch predicted taken on BTB miss only for backward branches. 11 - Branch predicted not taken on BTB miss for both branch directions. This field controls operation of static prediction mechanism on a BTB miss. Unless disabled, fetching of the predicted target location will be performed for branch acceleration. BPRED operates independently of BPEN, and with a BPEN setting of 0, will be used to perform static prediction of all unresolved branches. BTB hits are not affected by the settings of this field. For certain applications, setting BPRED to a non-default value may result in improved performance.
31 [63]	BPEN	Branch target buffer prediction enable. 0 Branch target buffer prediction disabled 1 Branch target buffer prediction enabled (enables BTB to predict branches) When the BPEN bit is cleared, no hits will be generated from the BTB, and no new entries will be allocated. Entries are not automatically invalidated when BPEN is cleared; the BBFI bit controls entry invalidation. BPEN operates independently of BPRED, and will be used even with a BPRED setting of 00.

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

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

The L1CSR0 and L1CSR1 registers are used for general control and status of the L1 caches. A description of the L1CSR0 and L1CSR1 registers can be found in Chapter 11, L1 Cache.

2.4.15 L1 Cache Configuration registers (L1CFG0, L1CFG1)

The L1CFG0 and L1CGF1 registers provide configuration information for the L1 caches supplied with this version of the Zen CPU core. A description of the L1CFG0 and L1CGF1 registers can be found in Chapter 11, L1 Cache.

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

The L1FINV0 and L1FINV1 registers provide software-based flush and invalidation control for the L1 caches supplied with this version of the Zen CPU core. A description of the L1FINV0 and L1FINV1 registers can be found in Chapter 11, L1 Cache.

2.4.17 MMU Control and Status Register (MMUCSR0)

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

2.4.18 MMU Configuration register (MMUCFG)

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

2.4.19 TLB Configuration registers (TLB0CFG, TLB1CFG)

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

2.5 SPR register access

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

2.5.1 Invalid SPR references

System behavior when an invalid SPR is referenced depends on the apparent privilege level of the register. The register privilege level is determined by bit 5 in the SPR address. If the invalid SPR is accessible in user mode, then an illegal exception is generated. If the invalid SPR is accessible only in supervisor mode and the CPU core is in supervisor mode (MSR_{PR} = 0), then an illegal exception is generated. If the invalid SPR address is accessible only in supervisor mode and the CPU is not in supervisor mode (MSR_{PR} = 1), then a privilege exception is generated.

NOTE

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

Table 14. System response to invalid SPR reference

SPR address bit 5	Mode	MSR_{PR}	Response
0	—	—	Illegal exception
1	Supervisor	0	Illegal exception
1	user	1	Privilege exception

2.5.2 Synchronization requirements for SPRs

With the exception of the following registers, there are no synchronization requirements for accessing SPRs beyond those stated in *PowerPC Book E*. A complete description of Synchronization requirements are contained in Chapter 11 of *Book E: Enhanced PowerPC^{tm} Architecture v0.99* beginning on page 219. Software requirements for synchronization before/after accessing these registers are shown in Table 15. The notation CSI in the table refers to a Context Synchronizing instruction, which includes **sc**, **isync**, **rfi**, **rfci**, and **rfdi**.

Table 15. Additional synchronization requirements for SPRs

Context altering event or instruction		Required before	Required after	Notes
mtmsr[UCLE]		none	CSI	—
mtmsr[SPE]		none	CSI	—
mtmsr[PMM]		none	CSI	—
mfspr				
DBCNT	Debug Counter register	msync	none	1
DBSR	Debug Status register	msync	none	—
HID0	Hardware implementation dependent reg 0	none	none	—
HID1	Hardware implementation dependent reg 1	msync	none	—
L1CSR0, L1CSR1	L1 cache control and status registers 0,1	msync	none	—
L1FINV0, L1FINV1	L1 cache flush and invalidate control registers 0,1	msync	none	—
MMUCSR	MMU control and status register 0	CSI	none	—
mtspr				
BUCSR	Branch Unit Control and Status Register	none	CSI	—
DBCNT	Debug Counter register	none	CSI	1
DBCR0-6	Debug Control Register 0-6	none	CSI	—
DBSR	Debug Status Register	msync	none	—
HID0	Hardware implementation dependent reg 0	CSI	isync	—
HID1	Hardware implementation dependent reg 1	msync, isync	CSI	—
L1CSR0	L1 cache control and status register 0	msync, isync	CSI	—
L1CSR1	L1 cache control and status registers 1	none	CSI	—

Table 15. Additional synchronization requirements for SPRs (continued)

Context altering event or instruction		Required before	Required after	Notes
L1FINV0, L1FINV1	L1 cache flush and invalidate control registers 0,1	msync	CSI	—
MASx	MMU MAS registers	none	CSI	—
MMUCSR	MMU control and status register 0	CSI	CSI	—
PID	PID0 register	none	CSI	—
SPEFSCR	SPEFSCR register	none	CSI[2]	—

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

2.5.3 Special purpose register summary

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

Table 16. Special purpose registers

Mnemonic	Name	SPR number	Access	Privileged	Zen-specific
BUCSR	Branch Unit Control and Status Register	1013	R/W	Yes	Yes
CSRR0	Critical Save/Restore Register 0	58	R/W	Yes	No
CSRR1	Critical Save/Restore Register 1	59	R/W	Yes	No
CTR	Count Register	9	R/W	No	No
DAC1	Data Address Compare 1	316	R/W	Yes	No
DAC2	Data Address Compare 2	317	R/W	Yes	No
DBCNT	Debug Counter register	562	R/W	Yes	Yes
DBCR0	Debug Control Register 0	308	R/W	Yes	No
DBCR1	Debug Control Register 1	309	R/W	Yes	No
DBCR2	Debug Control Register 2	310	R/W	Yes	No
DBCR3	Debug control register 3	561	R/W	Yes	Yes
DBCR4	Debug control register 4	563	R/W	Yes	Yes
DBCR5	Debug control register 5	564	R/W	Yes	Yes
DBCR6	Debug control register 5	603	R/W	Yes	Yes
DBERC0	Debug external resource control register 0	569	Read-only	Yes	Yes
DBSR	Debug Status Register	304	Read/Clear[1]	Yes	No

Table 16. Special purpose registers (continued)

Mnemonic	Name	SPR number	Access	Privileged	Zen-specific
DDAM	Debug Data Acquisition Messaging register	576	R/W	No	Yes
DEAR	Data Exception Address Register	61	R/W	Yes	No
DEC	Decrementer	22	R/W	Yes	No
DECAR	Decrementer Auto-Reload	54	R/W	Yes	No
DEVENT	Debug Event register	975	R/W	No	Yes
DSRR0	Debug save/restore register 0	574	R/W	Yes	Yes
DSRR1	Debug save/restore register 1	575	R/W	Yes	Yes
DVC1	Data Value Compare 1	318	R/W	Yes	No
DVC2	Data Value Compare 2	319	R/W	Yes	No
ESR	Exception Syndrome Register	62	R/W	Yes	No
HID0	Hardware implementation dependent reg 0	1008	R/W	Yes	Yes
HID1	Hardware implementation dependent reg 1	1009	R/W	Yes	Yes
IAC1	Instruction Address Compare 1	312	R/W	Yes	No
IAC2	Instruction Address Compare 2	313	R/W	Yes	No
IAC3	Instruction Address Compare 3	314	R/W	Yes	No
IAC4	Instruction Address Compare 4	315	R/W	Yes	No
IAC5	Instruction Address Compare 5	565	R/W	Yes	Yes
IAC6	Instruction Address Compare 6	566	R/W	Yes	Yes
IAC7	Instruction Address Compare 7	567	R/W	Yes	Yes
IAC8	Instruction Address Compare 8	568	R/W	Yes	Yes
IVOR0	Interrupt Vector Offset Register 0	400	R/W	Yes	No
IVOR1	Interrupt Vector Offset Register 1	401	R/W	Yes	No
IVOR2	Interrupt Vector Offset Register 2	402	R/W	Yes	No
IVOR3	Interrupt Vector Offset Register 3	403	R/W	Yes	No
IVOR4	Interrupt Vector Offset Register 4	404	R/W	Yes	No
IVOR5	Interrupt Vector Offset Register 5	405	R/W	Yes	No
IVOR6	Interrupt Vector Offset Register 6	406	R/W	Yes	No
IVOR7	Interrupt Vector Offset Register 7	407	R/W	Yes	No
IVOR8	Interrupt Vector Offset Register 8	408	R/W	Yes	No
IVOR9	Interrupt Vector Offset Register 9	409	R/W	Yes	No
IVOR10	Interrupt Vector Offset Register 10	410	R/W	Yes	No
IVOR11	Interrupt Vector Offset Register 11	411	R/W	Yes	No
IVOR12	Interrupt Vector Offset Register 12	412	R/W	Yes	No
IVOR13	Interrupt Vector Offset Register 13	413	R/W	Yes	No

Table 16. Special purpose registers (continued)

Mnemonic	Name	SPR number	Access	Privileged	Zen-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
IVOR35	Interrupt vector offset register 35	531	R/W	Yes	Yes
IVPR	Interrupt Vector Prefix Register	63	R/W	Yes	No
LR	Link Register	8	R/W	No	No
L1CFG0	L1 cache config register 0	515	Read-only	No	Yes
L1CFG1	L1 cache config register 1	516	Read-only	No	Yes
L1CSR0	L1 cache control and status register 0	1010	R/W	Yes	Yes
L1CSR1	L1 cache control and status register 1	1011	R/W	Yes	Yes
L1FINV0	L1 cache flush and invalidate control register 0	1016	R/W	Yes	Yes
L1FINV1	L1 cache flush and invalidate control register 0	959	R/W	Yes	Yes
MAS0	MMU assist register 0	624	R/W	Yes	Yes
MAS1	MMU assist register 1	625	R/W	Yes	Yes
MAS2	MMU assist register 2	626	R/W	Yes	Yes
MAS3	MMU assist register 3	627	R/W	Yes	Yes
MAS4	MMU assist register 4	628	R/W	Yes	Yes
MAS6	MMU assist register 6	630	R/W	Yes	Yes
MCAR	Machine Check Address Register	573	R/W	Yes	Yes
MCSR	Machine Check Syndrome Register	572	R/Clear[2]	Yes	Yes
MCSRR0	Machine Check Save/Restore Register 0	570	R/W	Yes	Yes
MCSRR1	Machine Check Save/Restore Register 1	571	R/W	Yes	Yes
MMUCFG	MMU configuration register	1015	Read-only	Yes	Yes
MMUCSR	MMU control and status register 0	1012	R/W	Yes	Yes
PID0	Process ID Register	48	R/W	Yes	No
PIR	Processor ID Register	286	R/W	Yes	No
PVR	Processor Version Register	287	Read-only	Yes	No
SPEFSCR	SPE 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

Table 16. Special purpose registers (continued)

Mnemonic	Name	SPR number	Access	Privileged	Zen-specific
SPRG4	SPR General 4	260	Read-only	No	No
		276	R/W	Yes	No
SPRG5	SPR General 5	261	Read-only	No	No
		277	R/W	Yes	No
SPRG6	SPR General 6	262	Read-only	No	No
		278	R/W	Yes	No
SPRG7	SPR General 7	263	Read-only	No	No
		279	R/W	Yes	No
SPRG8	SPR General 8	604	R/W	Yes	Yes
SPRG9	SPR General 9	605	R/W	Yes	Yes
SRR0	Save/Restore Register 0	26	R/W	Yes	No
SRR1	Save/Restore Register 1	27	R/W	Yes	No
SVR	System Version Register	1023	Read-only	Yes	Yes
TBL	Time Base Lower	268	Read-only	No	No
		284	Write-only	Yes	No
TBU	Time Base Upper	269	Read-only	No	No
		285	Write-only	Yes	No
TCR	Timer Control Register	340	R/W	Yes	No
TLB0CFG	TLB0 configuration register	688	Read-only	Yes	Yes
TLB1CFG	TLB1 configuration register	689	Read-only	Yes	Yes
TSR	Timer Status Register	336	Read/Clear[3]	Yes	No
USPRG0	User SPR General 0	256	R/W	No	No
XER	Integer Exception Register	1	R/W	No	No

NOTES:

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

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

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

2.6 Reset settings

Table 17 shows the state of the *PowerPC Book E* architected registers and other optional resources immediately following a system reset.

Table 17. Reset settings for Zen resources

Resource	System reset setting
Program Counter	p_rstbase[0:29] ll 2’b00
GPRs	Unaffected[1]
CR	Unaffected[1]
BUCSR	0x0000_0000
CSRR0	Unaffected[1]
CSRR1	Unaffected[1]
CTR	Unaffected[1]
DAC1	0x0000_0000[2]
DAC2	0x0000_0000[2]
DBCNT	Unaffected[1]
DBCR0	0x0000_0000[2]
DBCR1	0x0000_0000[2]
DBCR2	0x0000_0000[2]
DBCR3	0x0000_0000[2]
DBCR4	0x0000_0000[2]
DBCR5	0x0000_0000[2]
DBCR6	0x0000_0000[2]
DBSR	0x1000_0000[2]
DDAM	0x0000_0000[2]
DEAR	Unaffected[1]
DEC	Unaffected[1]
DECAR	Unaffected[1]
DEVENT	0x0000_0000[2]
DSRR0	Unaffected[1]
DSRR1	Unaffected[1]
DVC1	Unaffected[1]
DVC2	Unaffected[1]
ESR	0x0000_0000
HID0	0x0000_0000
HID1	0x0000_0000
IAC1	0x0000_0000[2]
IAC2	0x0000_0000[2]
IAC3	0x0000_0000[2]
IAC4	0x0000_0000[2]
IAC5	0x0000_0000[2]

Table 17. Reset settings for Zen resources (continued)

Resource	System reset setting
IAC6	0x0000_0000[2]
IAC7	0x0000_0000[2]
IAC8	0x0000_0000[2]
IVORxx	Unaffected[1]
IVPR	Unaffected[1]
LR	Unaffected[1]
L1CFG0, L1CFG1[3]	—
L1CSR0, 1	0x0000_0000
L1FINV0, 1	0x0000_0000
MAS0	Unaffected[1]
MAS1	Unaffected[1]
MAS2	Unaffected[1]
MAS3	Unaffected[1]
MAS4	Unaffected[1]
MAS6	Unaffected[1]
MCAR	Unaffected[1]
MCSR	0x0000_0000
MCSRR0	Unaffected[1]
MCSRR1	Unaffected[1]
MMUCFG[3]	—
MSR	0x0000_0000
PID0	0x0000_0000
PIR	0x0000_00 || p_cpuid[0:7]
PVR[3]	—
SPEFSCR	0x0000_0000
SPRG0	Unaffected[1]
SPRG1	Unaffected[1]
SPRG2	Unaffected[1]
SPRG3	Unaffected[1]
SPRG4	Unaffected[1]
SPRG5	Unaffected[1]
SPRG6	Unaffected[1]
SPRG7	Unaffected[1]
SPRG8	Unaffected[1]
SPRG9	Unaffected[1]

Table 17. Reset settings for Zen resources (continued)

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

NOTES:
[1] Undefined on **m_por** assertion, unchanged on **p_reset_b** assertion
[2] Reset by processor reset **p_reset_b** if DBCR0[EDM]=0, as well as unconditionally by **m_por**.
[3] Read-only registers

Chapter 3 Instruction Model

This chapter provides additional information about the Book E Power Architecture architecture as it relates specifically to e200z759n3.

The e200z759n3 is a 32-bit implementation of the Book E Power Architecture architecture as defined in *Book E: Enhanced PowerPC^{tm} Architecture*. This architecture specification includes a recognition that different processor implementations may require clarifications, extensions or deviations from the architectural descriptions. The *PowerPC Book E* instruction set is described in Chapter 12 “Instruction Set” of *Book E: Enhanced PowerPC^{tm} Architecture v0.99* beginning on page 223.

3.1 Unsupported instructions and instruction forms

Because e200z759n3 is a 32-bit *PowerPC Book E* core, all of the instructions defined for 64-bit implementations of the *PowerPC Book E* architecture are illegal on Zen. See Appendix A of *Book E: Enhanced PowerPC^{tm} Architecture* for more information on 64-bit instructions. Zen takes an illegal instruction exception type program interrupt upon encountering a 64-bit *PowerPC Book E* instruction.

The e200z759n3 core does not support the instructions listed in Table 18. An illegal instruction exception is generated if the processor attempts to execute one of these instructions.

Table 18. List of unsupported instructions

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

3.2 Implementation-specific instructions

Several *PowerPC Book E* defined instructions are implementation-specific. Table 19 summarizes the Zen implementation-specific instructions.

Table 19. Implementation-specific instruction summary

Mnemonic	Implementation Details
mfapidi	unimplemented instructions (treated as illegal on e200z759n3)
mfdcrx	
mtdcrx	
stbcx., sthcx., stwcx.	address match with prior lbarx, lharx, or lwarx not req’d for store to be performed
mfdcr, mtdcr[1]	optionally supported instructions

NOTES:
[1] The Zen CPU will take an illegal instruction exception for unsupported DCR values

3.3 Book E instruction extensions

This section describes the various extensions to Book E instructions to support the PowerPC VLE APU.

rfci, ***rfdi***, ***rfi, rfmci*** - no longer mask bit 62 of CSRR0, DSRR0, or SRR0 respectively. The destination address is [D,C, MC]SRR0[32:62] || 0b0.

bclr, ***bclrl***, ***bcctr***, ***bcctrl*** - no longer mask bit 62 of the LR or CTR respectively. The destination address is [LR,CTR][32:62] || 0b0.

3.4 Memory access alignment support

The Zen core provides hardware support for unaligned memory accesses; however, there is a 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 that are misaligned across a 64-bit (8 byte) boundary can be translated at a rate of 2 cycles per store. Frequent use of unaligned memory accesses is discouraged because of the impact on performance.

NOTE

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

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

3.5 Memory synchronization and reservation instructions

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

In addition, the **msync** instruction, and the **mbar** w/MO=0, or 1 instructions handshake with the system to ensure that all accesses initiated by this CPU have been "performed" with respect to all other processors and mechanisms prior to completion of the instruction. Refer to Section 14.2.10, Memory synchronization control signals for further detail on the hardware handshake sequence.

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

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

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

The Zen core implements an internal status flag ($HID1_{ATS}$) representing reservation status. This flag is set when a load-type reservation instruction is executed and completes without error, and remains set until it is cleared by one of the following mechanisms:

1. Execution of a store-type reservation instruction is completed without error, or
2. The Zen core **p_rsrv_clr** input signal is asserted, or
3. The reservation is invalidated when an external input, critical input, or non-maskable interrupt is signaled and the $HID0_{ICR}$ bit is set.

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

The Zen core treats reservation accesses as though they were both cache inhibited and guarded, regardless of storage attributes. A hit to a cache line corresponding to the address of a reservation access will be flushed to memory if dirty, prior to the reservation access being issued to the bus. The access will be performed externally, regardless of a cache hit. This is done to allow external reservation logic to be built that properly signals a reservation failure.

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

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

3.6 Branch prediction

The e200z759n3 instruction fetching mechanism uses a branch target buffer (BTB) that holds branch target addresses combined with a 2-bit saturating up-down counter scheme for branch prediction. Branch paths are predicted by either the branch target buffer (BTB hit) or a selectable static prediction algorithm (BTB miss) and subsequently checked to see if the prediction was correct. This enables operation beyond a conditional branch without waiting for the branch to be decoded and resolved. The instruction fetch unit predicts the direction of the branch as follows:

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

3.7 Interruption of instructions by interrupt requests

In general, the e200z759n3 core samples pending non-maskable interrupts, external input, and critical input interrupt requests at instruction boundaries. However, in order to reduce interrupt latency, long running instructions may be interrupted prior to completion. Instructions in this class include divides (**divw[uo][.]**, **efsdiv**, **evfsdiv**, **evdivw[su]**), load multiple word (**lmw**, **e_lmw**), and store multiple word (**stmw**, **e_stmw**). In addition, the **e_lmvgprw**, **e_stmvgprw**, **e_lmvsprw**, and **e_stmvsprw** Volatile Context Save/Restore APU instructions may also be interrupted prior to completion. When interrupted prior to completion, the value saved in SRR0/CSRR0/MCSRR0 will be the address of the interrupted instruction. The instruction will be restarted from the beginning after returning to it from the interrupt handler.

3.8 New Zen instructions and APUs

The e200z759n3 core implements the following *Freescale EIS* APUs that extend the *PowerPC Book E* instruction set:

- The ISEL APU, which is described in Section 3.9, ISEL APU
- The Enhanced Debug APU and the Debug Notify Halt instructions, described in Section 3.10, Debug APU
- The Machine Check APU, which is described in Section 3.11, Machine Check APU
- The WAIT APU, which is described in Section 3.12, WAIT APU
- The Volatile Context Save/Restore APU, which is described in Section 3.14, Volatile Context Save/Restore APU
- The Embedded Floating-Point APU version 2, described along with supporting instructions in Chapter 5, Embedded Floating-Point APU (EFPU2).
- The Signal Processing Extension (SPE) APU version 1, described along with supporting instructions in Chapter 6, Signal Processing Extension APU (SPE APU).
- The Performance Monitor APU, which is described in Chapter 8, Performance Monitor

- The Cache Line-locking APU, which is described in Section 11.12, Cache line locking/unlocking APU
- The Enhanced Reservations APU, which is described in Section 3.13, Enhanced reservations APU

3.9 ISEL APU

The ISEL APU defines the **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. This instruction can be used to eliminate branches in software and in many cases improve performance. This instruction can also increase program execution time determinism by eliminating the need to predict the target and direction of the branches replaced by the integer select function. The instruction form and definition is as follows:

isel isel

Integer Select

isel RT, RA, RB, crb

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

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

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

Other registers altered:

- None

3.10 Debug APU

e200z759n3 implements the *Freescale EIS* Debug APU to support the capability to handle the Debug interrupt as an additional interrupt level. To support this interrupt level, a new ‘return from debug interrupt’ (**rfdi, se_rfdi**) instruction is defined as part of the Debug APU, along with a new pair of save/restore registers, DSRR0, and DSRR1.

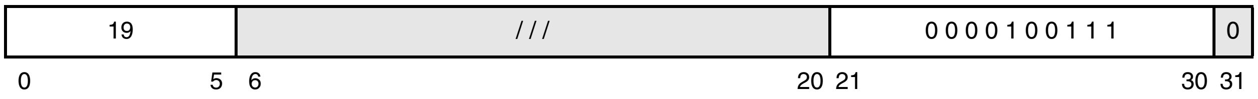
When the Debug APU is enabled ($HID0_{DAPUEN}$ = 1), the **rfdi** or **se_rfdi** instruction provides a means to return from a debug interrupt. See Section 2.4.11, Hardware Implementation Dependent Register 0 (HID0) for more information about enabling the Debug APU.

The instruction form and definition is as follows:

rfdi rfdi

Return From Debug Interrupt

rfdi



```
MSR ←DSRR1
PC ←DSRR0$_{0:30}$ || $^{1}$0
```

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

The contents of Debug Save/Restore Register 1 are place into the Machine State Register. If the new Machine State Register value does not enable any pending exceptions, then the next instruction is fetched, under control of the new Machine State Register value from the address DSRR0$_{0:30}$|| 1’b0. If the new Machine State Register value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into Save/Restore Register 0 or Critical Save/Restore Register 0 by the interrupt processing mechanism is the address of the instruction that would have been executed next had the interrupt not occurred (i.e. the address in Debug Save/Restore Register 0 at the time of the execution of the **rfdi**).

Execution of this instruction is privileged and context synchronizing.

Special Registers Altered:

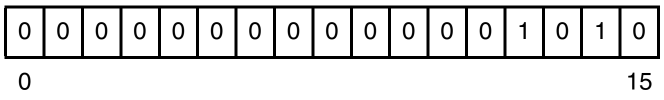
- MSR

When the Debug APU is disabled (HID0$_{\text{DAPUEN}}$=0), this instruction is treated as an illegal instruction.

se_rfdi se_rfdi

Return From Debug Interrupt

se_rfdi



```
MSR ←DSRR1
PC ←DSRR0$_{32:62}$ || 0b0
```

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

The contents of Debug Save/Restore Register 1 are place into the Machine State Register. If the new Machine State Register value does not enable any pending exceptions, then the next instruction is fetched, under control of the new Machine State Register value from the address DSRR0$_{32:62}$|| 0b0. If the new Machine State Register value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into Save/Restore Register 0 or Critical Save/Restore Register 0 by the interrupt processing mechanism is the address of the instruction that would have been executed next had the interrupt not occurred (i.e. the address in Debug Save/Restore Register 0 at the time of the execution of the **rfdi** or **se_rfdi**).

Execution of this instruction is privileged and context synchronizing.

Special Registers Altered:

- MSR

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

3.10.1 Debug notify halt instructions

The **dnh, e_dnh,** and **se_dnh** instructions provide a bridge between the execution of instructions on the core in a non-halted mode, and an external debug facility. **dnh**, **e_dnh**, and **se_dnh** allows software to transition the core from a running state to a debug halted state if enabled by an external debugger, and **dnh** provides the external debugger with bits reserved in the instruction itself to pass additional information. For e200z759n3, when the CPU enters a debug halted state due to a **dnh**, **e_dnh**, or **se_dnh** instruction, the instruction will be stored in the CPUSCR[IR] portion, and the CPUSCR[PC] value will point to the instruction. The external debugger should update the CPUSCR prior to exiting the debug halted state to point past the **dnh**, **e_dnh**, or **se_dnh** instruction.

Note that the **dnh** instruction is only available in BookE instruction pages, and the **e_dnh** and **se_dnh** instructions are only available in VLE instruction pages.

dnh

dnh

Debugger Notify Halt

dnh **dui, duis**

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

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

Execution of the **dnh** instruction causes the processor to halt if the external debug facility has enabled such action by previously setting the EDBCR$_{DNH\_EN}$ bit. If the processor is halted, the contents of the dui field are provided to the external debug facility to identify the reason for the halt.

If EDBCR$_{DNH\_EN}$ has not been previously set by the external debug facility, executing the **dnh** instruction produces an illegal instruction exception.

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

The **dnh** instruction is not privileged, and executes the same regardless of the state of MSR$_{PR}$.

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

Other registers altered:

- None.

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

e_dnh e_dnh

Debugger Notify Halt

e_dnh dui, duis

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

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

Execution of the **e_dnh** instruction causes the processor to halt if the external debug facility has enabled such action by previously setting the EDBCR$_{DNH\_EN}$ bit. If the processor is halted, the contents of the dui field are provided to the external debug facility to identify the reason for the halt.

If EDBCR$_{DNH\_EN}$ has not been previously set by the external debug facility, executing the **e_dnh** instruction produces an illegal instruction exception.

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

The **e_dnh** instruction is not privileged, and executes the same regardless of the state of MSR$_{PR}$.

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

Other registers altered:

- None.

se_dnh se_dnh

Debugger Notify Halt

se_dnh

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

```
if EDBCR$_{DNH\_EN}$ = 1 then

    halt processor
else
    illegal instruction exception
```

Execution of the **se_dnh** instruction causes the processor to halt if the external debug facility has enabled such action by previously setting the EDBCR$_{DNH\_EN}$ bit.

If EDBCR$_{DNH\_EN}$ has not been previously set by the external debug facility, executing the **se_dnh** instruction produces an illegal instruction exception.

The **se_dnh** instruction is not privileged, and executes the same regardless of the state of MSR$_{PR}$.

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

Other registers altered:

- None.

3.11 Machine Check APU

e200z759n3 implements the *Freescale EIS* Machine Check APU to support the capability to handle the Machine Check interrupt as an additional interrupt level. To support this interrupt level, a new ‘return from Machine Check interrupt’ (**rfmci, se_rfmci**) instruction is defined as part of the Machine Check APU, along with a new pair of save/restore registers, MCSRR0, and MCSRR1, a machine check syndrome register MCSR, and a machine check address register MCAR.

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

rfmci rfmci

Return From Machine Check Interrupt

rfmci

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

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

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

The contents of Machine Check Save/Restore Register 1 are place into the Machine State Register. If the new Machine State Register value does not enable any pending exceptions, then the next instruction is fetched, under control of the new Machine State Register value from the address MCSRR0$_{0:30}$|| 1’b0. If the new Machine State Register value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into the appropriate Save/Restore Register 0 by the interrupt processing mechanism is the address of the instruction that would have been executed next had the interrupt not occurred (i.e. the address in Machine Check Save/Restore Register 0 at the time of the execution of the **rfmci**).

Execution of this instruction is privileged and context synchronizing.

Special Registers Altered:

- MSR

NOTE

This instruction is only available in Book E instruction pages, it is not available in VLE instruction pages.

se_rfmci se_rfmci

Return From Machine Check Interrupt

se_rfmci

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

```
MSR ←MCSRR1
```

$\mathtt{PC} \leftarrow \mathtt{MCSRR0}_{0:30} \,\|\, {}^{1}0$

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

The contents of Machine Check Save/Restore Register 1 are place into the Machine State Register. If the new Machine State Register value does not enable any pending exceptions, then the next instruction is fetched, under control of the new Machine State Register value from the address $\text{MCSRR0}_{0:30}\|$ 1'b0. If the new Machine State Register value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into the appropriate Save/Restore Register 0 by the interrupt processing mechanism is the address of the instruction that would have been executed next had the interrupt not occurred (i.e. the address in Machine Check Save/Restore Register 0 at the time of the execution of the **se_rfmci**).

Execution of this instruction is privileged and context synchronizing.

Special Registers Altered:

- MSR

NOTE

This instruction is only available in VLE instruction pages, it is not available in BookE instruction pages.

3.12 WAIT APU

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

wait wait

Wait for Interrupt

wait

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

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

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

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

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

NOTE

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

3.13 Enhanced reservations APU

Zen implements the EIS enhanced reservations APU, which extends the load and reserve and store conditional instructions to support byte and halfword data types. These instructions operate in the same manner as the **lwarx** and **stwcx.** instructions, except for the size of the access.

lbarx lbarx

Load Byte And Reserve Indexed

lbarx RT,RA,RB (X-mode)

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

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

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

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

The byte in storage addressed by EA is loaded into GPR(RT)$_{56:63}$. GPR(RT)$_{0:55}$ are set to 0.

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

Special Registers Altered:

- None

lharx lharx

Load Halfword And Reserve Indexed

lharx RT,RA,RB (X-mode)

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

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

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

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

The halfword in storage addressed by EA is loaded into GPR(RT)$_{48:63}$. GPR(RT)$_{0:47}$ are set to 0.

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

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

Special Registers Altered:

- None

stbcx. stbcx.

Store Byte Conditional Indexed

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

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

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

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

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

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

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

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

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

```
CR0$_{LT\ GT\ EQ\ SO}$ = 0b00 || store_performed || XER$_{SO}$
```

Special Registers Altered:

- CR0

sthcx. sthcx.

Store Halfword Conditional Indexed

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

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

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

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

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

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

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

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

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

```
CR0$_{LT\ GT\ EQ\ SO}$ = 0b00 || store_performed || XER$_{SO}$
```

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

Special Registers Altered:

- CR0

3.14 Volatile Context Save/Restore APU

Zen implements the EIS Volatile Context Save/Restore APU to support the capability to quickly save and restore volatile register context on entry into an interrupt handler. To support this functionality, a new set of instructions is defined as part of the APU.

- e_lmvgprw, e_stmvgprw — load/store multiple volatile gprs (r0, r3:r12)
- e_lmvsprw, e_stmvsprw — load/store multiple volatile sprs (CR, LR, CTR, and XER)
- e_lmvsrrw, e_stmvsrrw — load/store multiple volatile srrs (SRR0, SRR1)
- e_lmvcsrrw, e_stmvcsrrw — load/store multiple volatile csrrs (CSRR0, CSRR1)
- e_lmvdsrrw, e_stmvdsrrw — load/store multiple volatile dsrrs (DSRR0, DSRR1)
- e_lmvmcsrrw, e_stmvmcsrrw — load/store multiple volatile mcsrrs (MCSRR0, MCSRR1)

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

e_lmvgprw e_lmvgprw

Load Multiple Volatile GPR Word

e_lmvgprw D8(RA) (D8-mode)

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

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

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

r ← 3
do while r $\le$ 12
```

```
            GPR(r)$_{32:63}$ ← MEM(EA,4)
    EA ← (EA+4)
    r  ← r + 1
```

Let the effective address (EA) be the sum of the contents of GPR(RA), or 0 if RA=0, and the sign-extended value of the D8 instruction field.

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

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

Special Registers Altered:

- None

e_stmvgprw e_stmvgprw

Store Multiple Volatile GPR Word

e_stmvgprw D8(RA) (D8-mode)

0 1 2 3 4 5	6 7 8 9 10	11–15	16 17 18 19 20 21 22 23	24–31
0 0 0 1 1 0	0 0 0 0 0	RA	0 0 0 1 0 0 0 1	D8

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

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

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

Let the effective address (EA) be the sum of the contents of GPR(RA), or 0 if RA=0, and the sign-extended value of the D8 instruction field.

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

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

Special Registers Altered:

- None

e_lmvsprw e_lmvsprw

Load Multiple Volatile SPR Word

e_lmvsprw D8(RA) (D8-mode)

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

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

$LR_{32:63}$ ← MEM(EA,4)
EA ← (EA+4)

$CTR_{32:63}$ ← MEM(EA,4)
EA ← (EA+4)

$XER_{32:63}$ ← MEM(EA,4)
```

Let the effective address (EA) be the sum of the contents of GPR(RA), or 0 if RA=0, and the sign-extended value of the D8 instruction field.

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

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

Special Registers Altered:

- CR
- LR
- CTR
- XER

e_stmvsprw e_stmvsprw

Store Multiple Volatile SPR Word

e_stmvsprw D8(RA) (D8-mode)

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

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

MEM(EA,4) ← $CR_{32:63}$
```

```
EA ← (EA+4)

MEM(EA,4) ← LR$_{32:63}$
EA ← (EA+4)

MEM(EA,4) ← CTR$_{32:63}$
EA ← (EA+4)

MEM(EA,4) ← XER$_{32:63}$
```

Let the effective address (EA) be the sum of the contents of GPR(RA), or 0 if RA=0, and the sign-extended value of the D8 instruction field.

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

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

Special Registers Altered:

- None

e_lmvsrrw e_lmvsrrw

Load Multiple Volatile SRR Word

e_lmvsrrw D8(RA) (D8-mode)

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

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

SRR0$_{32:63}$ ← MEM(EA,4)
EA ← (EA+4)
SRR1$_{32:63}$ ← MEM(EA,4)
```

Let the effective address (EA) be the sum of the contents of GPR(RA), or 0 if RA=0, and the sign-extended value of the D8 instruction field.

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

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

Special Registers Altered:

- SRR0
- SRR1

e_stmvsrrw e_stmvsrrw

Store Multiple Volatile SRR Word

e_stmvsrrw D8(RA) (D8-mode)

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

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

MEM(EA,4) ← SRR0$_{32:63}$
EA ← (EA+4)
MEM(EA,4) ← SRR1$_{32:63}$
```

Let the effective address (EA) be the sum of the contents of GPR(RA), or 0 if RA=0, and the sign-extended value of the D8 instruction field.

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

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

Special Registers Altered:

- None

e_lmvcsrrw e_lmvcsrrw

Load Multiple Volatile CSRR Word

e_lmvcsrrw D8(RA) (D8-mode)

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

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

CSRR0$_{32:63}$ ← MEM(EA,4)
EA ← (EA+4)
CSRR1$_{32:63}$ ← MEM(EA,4)
```

Let the effective address (EA) be the sum of the contents of GPR(RA), or 0 if RA=0, and the sign-extended value of the D8 instruction field.

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

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

Special Registers Altered:

- CSRR0
- CSRR1

e_stmvcsrrw e_stmvcsrrw

Store Multiple Volatile CSRR Word

e_stmvcsrrw D8(RA) (D8-mode)

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

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

MEM(EA,4) ← CSRR0$_{32:63}$
EA ← (EA+4)
MEM(EA,4) ← CSRR1$_{32:63}$
```

Let the effective address (EA) be the sum of the contents of GPR(RA), or 0 if RA=0, and the sign-extended value of the D8 instruction field.

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

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

Special Registers Altered:

- None

e_lmvdsrrw e_lmvdsrrw

Load Multiple Volatile DSRR Word

e_lmvdsrrw D8(RA) (D8-mode)

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

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

DSRR0$_{32:63}$ ← MEM(EA,4)
```

```
EA ← (EA+4)
DSRR1$_{32:63}$ ← MEM(EA,4)
```

Let the effective address (EA) be the sum of the contents of GPR(RA), or 0 if RA=0, and the sign-extended value of the D8 instruction field.

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

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

Special Registers Altered:

- DSRR0
- DSRR1

e_stmvdsrrw e_stmvdsrrw

Store Multiple Volatile DSRR Word

e_stmvdsrrw D8(RA) (D8-mode)

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

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

MEM(EA,4) ← DSRR0$_{32:63}$
EA ← (EA+4)
MEM(EA,4) ← DSRR1$_{32:63}$
```

Let the effective address (EA) be the sum of the contents of GPR(RA), or 0 if RA=0, and the sign-extended value of the D8 instruction field.

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

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

Special Registers Altered:

- None

e_lmvmcsrrw e_lmvmcsrrw

Load Multiple Volatile MCSRR Word

e_lmvmcsrrw D8(RA) (D8-mode)

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

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

MCSRR0$_{32:63}$ ← MEM(EA,4)
EA ← (EA+4)
MCSRR1$_{32:63}$ ← MEM(EA,4)
```

Let the effective address (EA) be the sum of the contents of GPR(RA), or 0 if RA=0, and the sign-extended value of the D8 instruction field.

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

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

Special Registers Altered:

- MCSRR0
- MCSRR1

e_stmvmcsrrw e_stmvmcsrrw

Store Multiple Volatile MCSRR Word

e_stmvmcsrrw D8(RA) (D8-mode)

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

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

MEM(EA,4) ← MCSRR0$_{32:63}$
EA ← (EA+4)
MEM(EA,4) ← MCSRR1$_{32:63}$
```

Let the effective address (EA) be the sum of the contents of GPR(RA), or 0 if RA=0, and the sign-extended value of the D8 instruction field.

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

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

Special Registers Altered:

- None

3.15 Unimplemented SPRs and read-only SPRs

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

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

3.16 Invalid forms of instructions

3.16.1 Load and store with update instructions

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

3.16.2 Load multiple word (lmw, e_lmw) instruction

PowerPC Book E defines as invalid any form of the **lmw** or **e_lmw** instruction in which RA is in the range of registers to be loaded, including the case in which RA=0. On Zen, invalid forms of the **lmw** or **e_lmw** instruction will be executed as follows:

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

3.16.3 Branch conditional to count register instructions

PowerPC Book E defines as invalid any **bcctr** or **bcctrl** instruction that specifies the ‘decrement and test CTR’ (BO_2=0) option. For these invalid forms of instructions Zen will execute the instruction by decrementing the CTR and branch to the location specified by the pre-decremented CTR value if all CR and CTR conditions are met as specified by the other BO field settings.

3.16.4 Instructions with reserved fields non-zero

PowerPC Book E defines certain bit fields in various instructions as reserved and specifies that these fields be set to zero. Per the Book E recommendation, Zen ignores the value of the reserved field (bit 31) in X-form integer load and store instructions. Zen ignores the value of the reserved ‘z’ bits in the BO field of branch instructions. For all other instructions, Zen will generate an illegal instruction exception if a reserved field is non-zero.

3.17 Instruction summary

Table 20 and Table 21 list all 32-bit instructions in *PowerPC Book E*, as well as certain Zen specific instructions, sorted by mnemonic. Format, Opcode, Mnemonic, Instruction name, and page number in *Book E: Enhanced PowerPC^{tm} Architecture v0.99* are included in the table. For Zen specific instructions, page number is not shown. Entries with a are unsupported by the Zen core, and will signal an illegal instruction exception. Implementation dependent instructions are noted with a footnote. Instructions that are optionally supported (when an optional function is added to the base core) are shown with shaded entries.

Note that specific APUs are not included in the table below:

- Cache Maintenance APU
- SPE APU
- VLE APU
- WAIT APU
- Enhanced Reservation APU
- Volatile Context Save/Restore APU

3.17.1 Instruction index sorted by mnemonic

Table 20. Instructions sorted by mnemonic

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
X	011111	01000 01010 0	**add**	Add	223
X	011111	01000 01010 1	**add.**	Add & record CR	223
X	011111	00000 01010 0	**addc**	Add Carrying	224
X	011111	00000 01010 1	**addc.**	Add Carrying & record CR	224
X	011111	10000 01010 0	**addco**	Add Carrying & record OV	224
X	011111	10000 01010 1	**addco.**	Add Carrying & record OV & CR	224
X	011111	00100 01010 0	**adde**	Add Extended with CA	225
X	011111	00100 01010 1	**adde.**	Add Extended with CA & record CR	225
X	011111	10100 01010 0	**addeo**	Add Extended with CA & record OV	225
X	011111	10100 01010 1	**addeo.**	Add Extended with CA & record OV & CR	225
D	001110	----- ----- -	**addi**	Add Immediate	226
D	001100	----- ----- -	**addic**	Add Immediate Carrying	227
D	001101	----- ----- -	**addic.**	Add Immediate Carrying & record CR	227
D	001111	----- ----- -	**addis**	Add Immediate Shifted	226
X	011111	00111 01010 0	**addme**	Add to Minus One Extended with CA	228
X	011111	00111 01010 1	**addme.**	Add to Minus One Extended with CA & record CR	228
X	011111	10111 01010 0	**addmeo**	Add to Minus One Extended with CA & record OV	228
X	011111	10111 01010 1	**addmeo.**	Add to Minus One Extended with CA & record OV & CR	228
X	011111	11000 01010 0	**addo**	Add & record OV	223
X	011111	11000 01010 1	**addo.**	Add & record OV & CR	223
X	011111	00110 01010 0	**addze**	Add to Zero Extended with CA	229
X	011111	00110 01010 1	**addze.**	Add to Zero Extended with CA & record CR	229

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’s Manual for the implementation

Table 20. Instructions sorted by mnemonic (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 page
	Primary ($\text{Inst}_{0:5}$)	Extended ($\text{Inst}_{21:31}$)			
X	011111	10110 01010 0	**addzeo**	Add to Zero Extended with CA & record OV	229
X	011111	10110 01010 1	**addzeo.**	Add to Zero Extended with CA & record OV & CR	229
X	011111	00000 11100 0	**and**	AND	230
X	011111	00000 11100 1	**and.**	AND & record CR	230
X	011111	00001 11100 0	**andc**	AND with Complement	230
X	011111	00001 11100 1	**andc.**	AND with Complement & record CR	230
D	011100	----- ----- -	**andi.**	AND Immediate & record CR	230
D	011101	----- ----- -	**andis.**	AND Immediate Shifted & record CR	230
I	010010	----- ----0 0	**b**	Branch	231
I	010010	----- ----1 0	**ba**	Branch Absolute	231
B	010000	----- ----0 0	**bc**	Branch Conditional	232
B	010000	----- ----1 0	**bca**	Branch Conditional Absolute	232
XL	010011	10000 10000 0	**bcctr**	Branch Conditional to Count Register	233
XL	010011	10000 10000 1	**bcctrl**	Branch Conditional to Count Register & Link	233
B	010000	----- ----0 1	**bcl**	Branch Conditional & Link	232
B	010000	----- ----1 1	**bcla**	Branch Conditional & Link Absolute	232
XL	010011	00000 10000 0	**bclr**	Branch Conditional to Link Register	234
XL	010011	00000 10000 1	**bclrl**	Branch Conditional to Link Register & Link	234
I	010010	----- ----0 1	**bl**	Branch & Link	231
I	010010	----- ----1 1	**bla**	Branch & Link Absolute	231
X	011111	00000 00000 /	**cmp**	Compare	235
D	001011	----- ----- -	**cmpi**	Compare Immediate	235
X	011111	00001 00000 /	**cmpl**	Compare Logical	236

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’s Manual for the implementation

Table 20. Instructions sorted by mnemonic (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 page
	Primary (Inst$_{0:5}$)	Extended (Inst$_{21:31}$)			
D	001010	----- ----- -	**cmpli**	Compare Logical Immediate	236
X	011111	00000 11010 0	**cntlzw**	Count Leading Zeros Word	237
X	011111	00000 11010 1	**cntlzw.**	Count Leading Zeros Word & record CR	237
XL	010011	01000 00001 /	**crand**	Condition Register AND	238
XL	010011	00100 00001 /	**crandc**	Condition Register AND with Complement	238
XL	010011	01001 00001 /	**creqv**	Condition Register Equivalent	238
XL	010011	00111 00001 /	**crnand**	Condition Register NAND	239
XL	010011	00001 00001 /	**crnor**	Condition Register NOR	239
XL	010011	01110 00001 /	**cror**	Condition Register OR	239
XL	010011	01101 00001 /	**crorc**	Condition Register OR with Complement	240
XL	010011	00110 00001 /	**crxor**	Condition Register XOR	240
X	011111	10111 10110 /	**dcba**	Data Cache Block Allocate	241
X	011111	00010 10110 /	**dcbf**	Data Cache Block Flush	242
X	011111	01110 10110 /	**dcbi**	Data Cache Block Invalidate	243
X	011111	01100 00110 /	**dcblc**[1]	Data Cache Block Lock Clear	—
X	011111	00001 10110 /	**dcbst**	Data Cache Block Store	245
X	011111	01000 10110 /	**dcbt**	Data Cache Block Touch	246
X	011111	00101 00110 /	**dcbtls**[1]	Data Cache Block Touch and Lock Set	—
X	011111	00111 10110 /	**dcbtst**	Data Cache Block Touch for Store	247
X	011111	00100 00110 /	**dcbtstls**[1]	Data Cache Block Touch for Store and Lock Set	—
X	011111	11111 10110 /	**dcbz**	Data Cache Block set to Zero	248
X	011111	01111 01011 0	**divw**	Divide Word	251
X	011111	01111 01011 1	**divw.**	Divide Word & record CR	251

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's Manual for the implementation

Table 20. Instructions sorted by mnemonic (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 page
	Primary (Inst$_{0:5}$)	Extended (Inst$_{21:31}$)			
X	011111	11111 01011 0	**divwo**	Divide Word & record OV	251
X	011111	11111 01011 1	**divwo.**	Divide Word & record OV & CR	251
X	011111	01110 01011 0	**divwu**	Divide Word Unsigned	252
X	011111	01110 01011 1	**divwu.**	Divide Word Unsigned & record CR	252
X	011111	11110 01011 0	**divwuo**	Divide Word Unsigned & record OV	252
X	011111	11110 01011 1	**divwuo.**	Divide Word Unsigned & record OV & CR	252
X	011111	01000 11100 0	**eqv**	Equivalent	253
X	011111	01000 11100 1	**eqv.**	Equivalent & record CR	253
X	011111	11101 11010 0	**extsb**	Extend Sign Byte	254
X	011111	11101 11010 1	**extsb.**	Extend Sign Byte & record CR	254
X	011111	11100 11010 0	**extsh**	Extend Sign Halfword	254
X	011111	11100 11010 1	**extsh.**	Extend Sign Halfword & record CR	254
X	111111	01000 01000 0	2		255
X	111111	01000 01000 1	2		255
A	111111	----- 10101 0	2		256
A	111111	----- 10101 1	2		256
A	111011	----- 10101 0	2		256
A	111011	----- 10101 1	2		256
X	111111	11010 01110 /	2		257
X	111111	00001 00000 /	2		259
X	111111	00000 00000 /	2		259
X	111111	11001 01110 /	2		260
X	111111	11001 01111 /	2		260

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's Manual for the implementation

Table 20. Instructions sorted by mnemonic (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 page
	Primary (Inst$_{0:5}$)	Extended (Inst$_{21:31}$)			
X	111111	00000 01110 0	[2]		262
X	111111	00000 01110 1	[2]		262
X	111111	00000 01111 0	[2]		262
X	111111	00000 01111 1	[2]		262
A	111111	----- 10010 0	[2]		264
A	111111	----- 10010 1	[2]		264
A	111011	----- 10010 0	[2]		264
A	111011	----- 10010 1	[2]		264
A	111111	----- 11101 0	[2]		265
A	111111	----- 11101 1	[2]		265
A	111011	----- 11101 0	[2]		265
A	111011	----- 11101 1	[2]		265
X	111111	00010 01000 0	[2]		266
X	111111	00010 01000 1	[2]		266
A	111111	----- 11100 0	[2]		267
A	111111	----- 11100 1	[2]		267
A	111011	----- 11100 0	[2]		267
A	111011	----- 11100 1	[2]		267
A	111111	----- 11001 0	[2]		268
A	111111	----- 11001 1	[2]		268
A	111011	----- 11001 0	[2]		268
A	111011	----- 11001 1	[2]		268
X	111111	00100 01000 0	[2]		269

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’s Manual for the implementation

Table 20. Instructions sorted by mnemonic (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
X	111111	00100 01000 1	2		269
X	111111	00001 01000 0	2		269
X	111111	00001 01000 1	2		269
A	111111	----- 11111 0	2		270
A	111111	----- 11111 1	2		270
A	111011	----- 11111 0	2		270
A	111011	----- 11111 1	2		270
A	111111	----- 11110 0	2		271
A	111111	----- 11110 1	2		271
A	111011	----- 11110 0	2		271
A	111011	----- 11110 1	2		271
A	111011	----- 11000 0	2		272
A	111011	----- 11000 1	2		272
X	111111	00000 01100 0	2		273
X	111111	00000 01100 1	2		273
A	111111	----- 11010 0	2		276
A	111111	----- 11010 1	2		276
A	111111	----- 10111 0	2		277
A	111111	----- 10111 1	2		277
A	111111	----- 10110 0	2		278
A	111111	----- 10110 1	2		278
A	111011	----- 10110 0	2		278
A	111011	----- 10110 1	2		278

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's Manual for the implementation

Table 20. Instructions sorted by mnemonic (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
A	111111	----- 10100 0	[2]		279
A	111111	----- 10100 1	[2]		279
A	111011	----- 10100 0	[2]		279
A	111011	----- 10100 1	[2]		279
X	011111	11110 10110 /	**icbi**	Instruction Cache Block Invalidate	280
X	011111	00111 00110 /	**icblc**[1]	Instruction Cache Block Lock Clear	—
X	011111	00000 10110 /	**icbt**	Instruction Cache Block Touch	281
X	011111	01111 00110 /	**icbtls**[1]	Instruction Cache Block Touch and Lock Set	—
??	011111	----- 01111 /	**isel**[3]	Integer Select	—
XL	010011	00100 10110 /	**isync**	Instruction Synchronize	282
D	100010	----- ----- -	**lbz**	Load Byte & Zero	283
D	100011	----- ----- -	**lbzu**	Load Byte & Zero with Update	283
X	011111	00011 10111 /	**lbzux**	Load Byte & Zero with Update Indexed	283
X	011111	00010 10111 /	**lbzx**	Load Byte & Zero Indexed	283
D	110010	----- ----- -	[2]		286
D	110011	----- ----- -	[2]		286
X	011111	10011 10111 /	[2]		286
X	011111	10010 10111 /	[2]		286
D	110000	----- ----- -	[2]		287
D	110001	----- ----- -	[2]		287
X	011111	10001 10111 /	[2]		287
X	011111	10000 10111 /	[2]		287
D	101010	----- ----- -	**lha**	Load Halfword Algebraic	288

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

Table 20. Instructions sorted by mnemonic (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
D	101011	----- ----- -	**lhau**	Load Halfword Algebraic with Update	288
X	011111	01011 10111 /	**lhaux**	Load Halfword Algebraic with Update Indexed	288
X	011111	01010 10111 /	**lhax**	Load Halfword Algebraic Indexed	288
X	011111	11000 10110 /	**lhbrx**	Load Halfword Byte-Reverse Indexed	289
D	101000	----- ----- -	**lhz**	Load Halfword & Zero	290
D	101001	----- ----- -	**lhzu**	Load Halfword & Zero with Update	290
X	011111	01001 10111 /	**lhzux**	Load Halfword & Zero with Update Indexed	290
X	011111	01000 10111 /	**lhzx**	Load Halfword & Zero Indexed	290
D	101110	----- ----- -	**lmw**	Load Multiple Word	291
X	011111	10010 10101 /	4		292
X	011111	10000 10101 /	4		292
X	011111	00000 10100 /	**lwarx**[5]	Load Word & Reserve Indexed	294
X	011111	10000 10110 /	**lwbrx**	Load Word Byte-Reverse Indexed	296
D	100000	----- ----- -	**lwz**	Load Word & Zero	297
D	100001	----- ----- -	**lwzu**	Load Word & Zero with Update	297
X	011111	00001 10111 /	**lwzux**	Load Word & Zero with Update Indexed	297
X	011111	00000 10111 /	**lwzx**	Load Word & Zero Indexed	297
X	011111	11010 10110 /	**mbar**[5]	Memory Barrier	298
XL	010011	00000 00000 /	**mcrf**	Move Condition Register Field	299
X	111111	00010 00000 /	2		300
X	011111	10000 00000 /	**mcrxr**	Move to Condition Register from XER	300
X	011111	01000 10011 /	4		301
X	011111	00000 10011 /	**mfcr**	Move From Condition Register	301

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's Manual for the implementation

Table 20. Instructions sorted by mnemonic (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
XFX	011111	01010 00011 /	**mfdcr**	Move From Device Control Register	302
X	011111	01000 00011 /	[4]		302
X	111111	10010 00111 0	[2]		303
X	111111	10010 00111 1	[2]		303
X	011111	00010 10011 /	**mfmsr**	Move From Machine State Register	303
XFX	011111	01010 10011 /	**mfspr**	Move From Special Purpose Register	304
X	011111	10010 10110 /	**msync**[5]	Memory Synchronize	305
XFX	011111	00100 10000 /	**mtcrf**	Move To Condition Register Fields	306
XFX	011111	01110 00011 /	**mtdcr**	Move To Device Control Register	307
X	011111	01100 00011 /	[4]		307
X	111111	00010 00110 0	[2]		308
X	111111	00010 00110 1	[2]		308
X	111111	00001 00110 0	[2]		308
X	111111	00001 00110 1	[2]		308
XFL	111111	10110 00111 0	[2]		309
XFL	111111	10110 00111 1	[2]		309
X	111111	00100 00110 0	[2]		310
X	111111	00100 00110 1	[2]		310
X	011111	00100 10010 /	**mtmsr**	Move To Machine State Register	311
XFX	011111	01110 10011 /	**mtspr**	Move To Special Purpose Register	312
X	011111	/0010 01011 0	**mulhw**	Multiply High Word	314
X	011111	/0010 01011 1	**mulhw.**	Multiply High Word & record CR	314
X	011111	/0000 01011 0	**mulhwu**	Multiply High Word Unsigned	314

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's Manual for the implementation

Table 20. Instructions sorted by mnemonic (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
X	011111	/0000 01011 1	**mulhwu.**	Multiply High Word Unsigned & record CR	314
D	000111	----- ----- -	**mulli**	Multiply Low Immediate	315
X	011111	00111 01011 0	**mullw**	Multiply Low Word	316
X	011111	00111 01011 1	**mullw.**	Multiply Low Word & record CR	316
X	011111	10111 01011 0	**mullwo**	Multiply Low Word & record OV	316
X	011111	10111 01011 1	**mullwo.**	Multiply Low Word & record OV & CR	316
X	011111	01110 11100 0	**nand**	NAND	317
X	011111	01110 11100 1	**nand.**	NAND & record CR	317
X	011111	00011 01000 0	**neg**	Negate	318
X	011111	00011 01000 1	**neg.**	Negate & record CR	318
X	011111	10011 01000 0	**nego**	Negate & record OV	318
X	011111	10011 01000 1	**nego.**	Negate & record OV & record CR	318
X	011111	00011 11100 0	**nor**	NOR	319
X	011111	00011 11100 1	**nor.**	NOR & record CR	319
X	011111	01101 11100 0	**or**	OR	320
X	011111	01101 11100 1	**or.**	OR & record CR	320
X	011111	01100 11100 0	**orc**	OR with Complement	320
X	011111	01100 11100 1	**orc.**	OR with Complement & record CR	320
D	011000	----- ----- -	**ori**	OR Immediate	320
D	011001	----- ----- -	**oris**	OR Immediate Shifted	320
XL	010011	00001 10011 /	**rfci**	Return From Critical Interrupt	321
XL	010011	00001 00111 /	**rfdi**[6]	Return From Debug Interrupt	—
XL	010011	00001 10010 /	**rfi**	Return From Interrupt	322

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's Manual for the implementation

Table 20. Instructions sorted by mnemonic (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
XL	010011	00001 00110 /	**rfmci**[7]	Return From Machine Check Interrupt	—
M	010100	----- ----- 0	**rlwimi**	Rotate Left Word Immediate then Mask Insert	327
M	010100	----- ----- 1	**rlwimi.**	Rotate Left Word Immediate then Mask Insert & record CR	327
M	010101	----- ----- 0	**rlwinm**	Rotate Left Word Immediate then AND with Mask	328
M	010101	----- ----- 1	**rlwinm.**	Rotate Left Word Immediate then AND with Mask & record CR	328
M	010111	----- ----- 0	**rlwnm**	Rotate Left Word then AND with Mask	328
M	010111	----- ----- 1	**rlwnm.**	Rotate Left Word then AND with Mask & record CR	328
SC	010001	///// ////1 /	**sc**	System Call	330
X	011111	00000 11000 0	**slw**	Shift Left Word	332
X	011111	00000 11000 1	**slw.**	Shift Left Word & record CR	332
X	011111	11000 11000 0	**sraw**	Shift Right Algebraic Word	334
X	011111	11000 11000 1	**sraw.**	Shift Right Algebraic Word & record CR	334
X	011111	11001 11000 0	**srawi**	Shift Right Algebraic Word Immediate	334
X	011111	11001 11000 1	**srawi.**	Shift Right Algebraic Word Immediate & record CR	334
X	011111	10000 11000 0	**srw**	Shift Right Word	336
X	011111	10000 11000 1	**srw.**	Shift Right Word & record CR	336
D	100110	----- ----- -	**stb**	Store Byte	337
D	100111	----- ----- -	**stbu**	Store Byte with Update	337
X	011111	00111 10111 /	**stbux**	Store Byte with Update Indexed	337
X	011111	00110 10111 /	**stbx**	Store Byte Indexed	337
D	110110	----- ----- -	[2]		340
D	110111	----- ----- -	[2]		340
X	011111	10111 10111 /	[2]		340

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's Manual for the implementation

Table 20. Instructions sorted by mnemonic (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
X	011111	10110 10111 /	[2]		340
X	011111	11110 10111 /	[2]		341
D	110100	----- ----- -	[2]		342
D	110101	----- ----- -	[2]		342
X	011111	10101 10111 /	[2]		342
X	011111	10100 10111 /	[2]		342
D	101100	----- ----- -	**sth**	Store Halfword	343
X	011111	11100 10110 /	**sthbrx**	Store Halfword Byte-Reverse Indexed	344
D	101101	----- ----- -	**sthu**	Store Halfword with Update	343
X	011111	01101 10111 /	**sthux**	Store Halfword with Update Indexed	343
X	011111	01100 10111 /	**sthx**	Store Halfword Indexed	343
D	101111	----- ----- -	**stmw**	Store Multiple Word	345
X	011111	10110 10101 /	[4]		346
X	011111	10100 10101 /	[4]		346
D	100100	----- ----- -	**stw**	Store Word	347
X	011111	10100 10110 /	**stwbrx**	Store Word Byte-Reverse Indexed	348
X	011111	00100 10110 1	**stwcx.**[5]	Store Word Conditional Indexed & record CR	349
D	100101	----- ----- -	**stwu**	Store Word with Update	347
X	011111	00101 10111 /	**stwux**	Store Word with Update Indexed	347
X	011111	00100 10111 /	**stwx**	Store Word Indexed	347
X	011111	00001 01000 0	**subf**	Subtract From	351
X	011111	00001 01000 1	**subf.**	Subtract From & record CR	351
X	011111	00000 01000 0	**subfc**	Subtract From Carrying	352

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's Manual for the implementation

Table 20. Instructions sorted by mnemonic (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 page
	Primary (Inst$_{0:5}$)	Extended (Inst$_{21:31}$)			
X	011111	00000 01000 1	**subfc.**	Subtract From Carrying & record CR	352
X	011111	10000 01000 0	**subfco**	Subtract From Carrying & record OV	352
X	011111	10000 01000 1	**subfco.**	Subtract From Carrying & record OV & CR	352
X	011111	00100 01000 0	**subfe**	Subtract From Extended with CA	353
X	011111	00100 01000 1	**subfe.**	Subtract From Extended with CA & record CR	353
X	011111	10100 01000 0	**subfeo**	Subtract From Extended with CA & record OV	353
X	011111	10100 01000 1	**subfeo.**	Subtract From Extended with CA & record OV & CR	353
D	001000	----- ----- -	**subfic**	Subtract From Immediate Carrying	354
X	011111	00111 01000 0	**subfme**	Subtract From Minus One Extended with CA	355
X	011111	00111 01000 1	**subfme.**	Subtract From Minus One Extended with CA & record CR	355
X	011111	10111 01000 0	**subfmeo**	Subtract From Minus One Extended with CA & record OV	355
X	011111	10111 01000 1	**subfmeo.**	Subtract From Minus One Extended with CA & record OV & CR	355
X	011111	10001 01000 0	**subfo**	Subtract From & record OV	351
X	011111	10001 01000 1	**subfo.**	Subtract From & record OV & CR	351
X	011111	00110 01000 0	**subfze**	Subtract From Zero Extended with CA	356
X	011111	00110 01000 1	**subfze.**	Subtract From Zero Extended with CA & record CR	356
X	011111	10110 01000 0	**subfzeo**	Subtract From Zero Extended with CA & record OV	356
X	011111	10110 01000 1	**subfzeo.**	Subtract From Zero Extended with CA & record OV & CR	356
X	011111	11000 10010 /	**tlbivax**	TLB Invalidate Virtual Address Indexed	358
X	011111	11101 10010 /	**tlbre**	TLB Read Entry	359
X	011111	11100 10010 ?	**tlbsx**	TLB Search Indexed	360
X	011111	10001 10110 /	**tlbsync**	TLB Synchronize	361
X	011111	11110 10010 /	**tlbwe**	TLB Write Entry	362

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's Manual for the implementation

Table 20. Instructions sorted by mnemonic (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 page
	Primary (Inst$_{0:5}$)	Extended (Inst$_{21:31}$)			
X	011111	00000 00100 /	**tw**	Trap Word	363
D	000011	----- ----- -	**twi**	Trap Word Immediate	363
X	011111	00100 00011 /	**wrtee**	Write External Enable	364
X	011111	00101 00011 /	**wrteei**	Write External Enable Immediate	364
X	011111	01001 11100 0	**xor**	XOR	365
X	011111	01001 11100 1	**xor.**	XOR & record CR	365
D	011010	----- ----- -	**xori**	XOR Immediate	365
D	011011	----- ----- -	**xoris**	XOR Immediate Shifted	365
Legend: - Don't care, usually part of an operand field / Reserved bit, invalid instruction form if encoded as 1 ? Allocated for implementation-dependent use. See User's Manual for the implementation					

NOTES:
[1] Motorola Book E cache locking APU, refer to Section 11.12, Cache line locking/unlocking APU.
[2] Attempted execution causes an illegal instruction exception.
[3] Motorola Book E **isel** APU, refer to Section 3.9, ISEL APU.
[4] Attempted execution causes an an illegal instruction exception
[5] See Section 3.5, Memory synchronization and reservation instructions.
[6] See Section 3.10, Debug APU.
[7] See Section 3.11, Machine Check APU.

3.17.2 Instruction index sorted by opcode

Table 21. Instructions sorted by opcode

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 page
	Primary (Inst$_{0:5}$)	Extended (Inst$_{21:31}$)			
D	000011	----- ----- -	**twi**	Trap Word Immediate	363
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 21. Instructions sorted by opcode (continued)

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

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

Table 21. Instructions sorted by opcode (continued)

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

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 21. Instructions sorted by opcode (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
D	011101	----- ----- -	**andis.**	AND Immediate Shifted & record CR	230
??	011111	----- 01111 /	**isel**	Integer Select	—
X	011111	00000 00000 /	**cmp**	Compare	235
X	011111	00000 00100 /	**tw**	Trap Word	363
X	011111	00000 01000 0	**subfc**	Subtract From Carrying	352
X	011111	00000 01000 1	**subfc.**	Subtract From Carrying & record CR	352
X	011111	00000 01010 0	**addc**	Add Carrying	224
X	011111	00000 01010 1	**addc.**	Add Carrying & record CR	224
X	011111	/0000 01011 0	**mulhwu**	Multiply High Word Unsigned	314
X	011111	/0000 01011 1	**mulhwu.**	Multiply High Word Unsigned & record CR	314
X	011111	00000 10011 /	**mfcr**	Move From Condition Register	301
X	011111	00000 10100 /	**lwarx**	Load Word & Reserve Indexed	294
X	011111	00000 10110 /	**icbt**	Instruction Cache Block Touch	281
X	011111	00000 10111 /	**lwzx**	Load Word & Zero Indexed	297
X	011111	00000 11000 0	**slw**	Shift Left Word	332
X	011111	00000 11000 1	**slw.**	Shift Left Word & record CR	332
X	011111	00000 11010 0	**cntlzw**	Count Leading Zeros Word	237
X	011111	00000 11010 1	**cntlzw.**	Count Leading Zeros Word & record CR	237
X	011111	00000 11100 0	**and**	AND	230
X	011111	00000 11100 1	**and.**	AND & record CR	230
X	011111	00001 00000 /	**cmpl**	Compare Logical	236
X	011111	00001 01000 0	**subf**	Subtract From	351
X	011111	00001 01000 1	**subf.**	Subtract From & record CR	351

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 21. Instructions sorted by opcode (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
X	011111	00001 10110 /	**dcbst**	Data Cache Block Store	245
X	011111	00001 10111 /	**lwzux**	Load Word & Zero with Update Indexed	297
X	011111	00001 11100 0	**andc**	AND with Complement	230
X	011111	00001 11100 1	**andc.**	AND with Complement & record CR	230
X	011111	/0010 01011 0	**mulhw**	Multiply High Word	314
X	011111	/0010 01011 1	**mulhw.**	Multiply High Word & record CR	314
X	011111	00010 10011 /	**mfmsr**	Move From Machine State Register	303
X	011111	00010 10110 /	**dcbf**	Data Cache Block Flush	242
X	011111	00010 10111 /	**lbzx**	Load Byte & Zero Indexed	283
X	011111	00011 01000 0	**neg**	Negate	318
X	011111	00011 01000 1	**neg.**	Negate & record CR	318
X	011111	00011 10111 /	**lbzux**	Load Byte & Zero with Update Indexed	283
X	011111	00011 11100 0	**nor**	NOR	319
X	011111	00011 11100 1	**nor.**	NOR & record CR	319
X	011111	00100 00011 /	**wrtee**	Write External Enable	364
X	011111	00100 00110 /	**dcbtstls**[1]	Data Cache Block Touch for Store and Lock Set	—
X	011111	00100 01000 0	**subfe**	Subtract From Extended with CA	353
X	011111	00100 01000 1	**subfe.**	Subtract From Extended with CA & record CR	353
X	011111	00100 01010 0	**adde**	Add Extended with CA	225
X	011111	00100 01010 1	**adde.**	Add Extended with CA & record CR	225
XFX	011111	00100 10000 /	**mtcrf**	Move To Condition Register Fields	306
X	011111	00100 10010 /	**mtmsr**	Move To Machine State Register	311
X	011111	00100 10110 1	**stwcx.**	Store Word Conditional Indexed & record CR	349

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 21. Instructions sorted by opcode (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
X	011111	00100 10111 /	**stwx**	Store Word Indexed	347
X	011111	00101 00011 /	**wrteei**	Write External Enable Immediate	364
X	011111	00101 00110 /	**dcbtls**[1]	Data Cache Block Touch and Lock Set	—
X	011111	00101 10111 /	**stwux**	Store Word with Update Indexed	347
X	011111	00110 01000 0	**subfze**	Subtract From Zero Extended with CA	356
X	011111	00110 01000 1	**subfze.**	Subtract From Zero Extended with CA & record CR	356
X	011111	00110 01010 0	**addze**	Add to Zero Extended with CA	229
X	011111	00110 01010 1	**addze.**	Add to Zero Extended with CA & record CR	229
X	011111	00110 10111 /	**stbx**	Store Byte Indexed	337
X	011111	00111 00110 /	**icblc**[1]	Instruction Cache Block Lock Clear	—
X	011111	00111 01000 0	**subfme**	Subtract From Minus One Extended with CA	355
X	011111	00111 01000 1	**subfme.**	Subtract From Minus One Extended with CA & record CR	355
X	011111	00111 01010 0	**addme**	Add to Minus One Extended with CA	228
X	011111	00111 01010 1	**addme.**	Add to Minus One Extended with CA & record CR	228
X	011111	00111 01011 0	**mullw**	Multiply Low Word	316
X	011111	00111 01011 1	**mullw.**	Multiply Low Word & record CR	316
X	011111	00111 10110 /	**dcbtst**	Data Cache Block Touch for Store	247
X	011111	00111 10111 /	**stbux**	Store Byte with Update Indexed	337
X	011111	01000 00011 /			302
X	011111	01000 01010 0	**add**	Add	223
X	011111	01000 01010 1	**add.**	Add & record CR	223
X	011111	01000 10011 /			301
X	011111	01000 10110 /	**dcbt**	Data Cache Block Touch	246

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 21. Instructions sorted by opcode (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 page
	Primary (Inst$_{0:5}$)	Extended (Inst$_{21:31}$)			
X	011111	01000 10111 /	**lhzx**	Load Halfword & Zero Indexed	290
X	011111	01000 11100 0	**eqv**	Equivalent	253
X	011111	01000 11100 1	**eqv.**	Equivalent & record CR	253
X	011111	01001 10111 /	**lhzux**	Load Halfword & Zero with Update Indexed	290
X	011111	01001 11100 0	**xor**	XOR	365
X	011111	01001 11100 1	**xor.**	XOR & record CR	365
XFX	011111	01010 00011 /	**mfdcr**	Move From Device Control Register	302
XFX	011111	01010 10011 /	**mfspr**	Move From Special Purpose Register	304
X	011111	01010 10111 /	**lhax**	Load Halfword Algebraic Indexed	288
X	011111	01011 10111 /	**lhaux**	Load Halfword Algebraic with Update Indexed	288
X	011111	01100 00011 /			307
X	011111	01100 00110 /	**dcblc**[1]	Data Cache Block Lock Clear	—
X	011111	01100 10111 /	**sthx**	Store Halfword Indexed	343
X	011111	01100 11100 0	**orc**	OR with Complement	320
X	011111	01100 11100 1	**orc.**	OR with Complement & record CR	320
X	011111	01101 10111 /	**sthux**	Store Halfword with Update Indexed	343
X	011111	01101 11100 0	**or**	OR	320
X	011111	01101 11100 1	**or.**	OR & record CR	320
XFX	011111	01110 00011 /	**mtdcr**	Move To Device Control Register	307
X	011111	01110 01011 0	**divwu**	Divide Word Unsigned	252
X	011111	01110 01011 1	**divwu.**	Divide Word Unsigned & record CR	252
XFX	011111	01110 10011 /	**mtspr**	Move To Special Purpose Register	312
X	011111	01110 10110 /	**dcbi**	Data Cache Block Invalidate	243

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 21. Instructions sorted by opcode (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 page
	Primary (Inst$_{0:5}$)	Extended (Inst$_{21:31}$)			
X	011111	01110 11100 0	**nand**	NAND	317
X	011111	01110 11100 1	**nand.**	NAND & record CR	317
X	011111	01111 00110 /	**icbtls**[1]	Instruction Cache Block Touch and Lock Set	—
X	011111	01111 01011 0	**divw**	Divide Word	251
X	011111	01111 01011 1	**divw.**	Divide Word & record CR	251
X	011111	10000 00000 /	**mcrxr**	Move to Condition Register from XER	300
X	011111	10000 01000 0	**subfco**	Subtract From Carrying & record OV	352
X	011111	10000 01000 1	**subfco.**	Subtract From Carrying & record OV & CR	352
X	011111	10000 01010 0	**addco**	Add Carrying & record OV	224
X	011111	10000 01010 1	**addco.**	Add Carrying & record OV & CR	224
X	011111	10000 10101 /			292
X	011111	10000 10110 /	**lwbrx**	Load Word Byte-Reverse Indexed	296
X	011111	10000 10111 /			287
X	011111	10000 11000 0	**srw**	Shift Right Word	336
X	011111	10000 11000 1	**srw.**	Shift Right Word & record CR	336
X	011111	10001 01000 0	**subfo**	Subtract From & record OV	351
X	011111	10001 01000 1	**subfo.**	Subtract From & record OV & CR	351
X	011111	10001 10110 /	**tlbsync**	TLB Synchronize	361
X	011111	10001 10111 /			287
X	011111	10010 10101 /			292
X	011111	10010 10110 /	**msync**	Memory Synchronize	305
X	011111	10010 10111 /			286
X	011111	10011 01000 0	**nego**	Negate & record OV	318

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

Table 21. Instructions sorted by opcode (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 page
	Primary (Inst$_{0:5}$)	Extended (Inst$_{21:31}$)			
X	011111	10011 01000 1	**nego.**	Negate & record OV & record CR	318
X	011111	10011 10111 /			286
X	011111	10100 01000 0	**subfeo**	Subtract From Extended with CA & record OV	353
X	011111	10100 01000 1	**subfeo.**	Subtract From Extended with CA & record OV & CR	353
X	011111	10100 01010 0	**addeo**	Add Extended with CA & record OV	225
X	011111	10100 01010 1	**addeo.**	Add Extended with CA & record OV & CR	225
X	011111	10100 10101 /			346
X	011111	10100 10110 /	**stwbrx**	Store Word Byte-Reverse Indexed	348
X	011111	10100 10111 /			342
X	011111	10101 10111 /			342
X	011111	10110 01000 0	**subfzeo**	Subtract From Zero Extended with CA & record OV	356
X	011111	10110 01000 1	**subfzeo.**	Subtract From Zero Extended with CA & record OV & CR	356
X	011111	10110 01010 0	**addzeo**	Add to Zero Extended with CA & record OV	229
X	011111	10110 01010 1	**addzeo.**	Add to Zero Extended with CA & record OV & CR	229
X	011111	10110 10101 /			346
X	011111	10110 10111 /			340
X	011111	10111 01000 0	**subfmeo**	Subtract From Minus One Extended with CA & record OV	355
X	011111	10111 01000 1	**subfmeo.**	Subtract From Minus One Extended with CA & record OV & CR	355
X	011111	10111 01010 0	**addmeo**	Add to Minus One Extended with CA & record OV	228
X	011111	10111 01010 1	**addmeo.**	Add to Minus One Extended with CA & record OV & CR	228
X	011111	10111 01011 0	**mullwo**	Multiply Low Word & record OV	316
X	011111	10111 01011 1	**mullwo.**	Multiply Low Word & record OV & CR	316
X	011111	10111 10110 /	**dcba**	Data Cache Block Allocate	241

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 21. Instructions sorted by opcode (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 page
	Primary (Inst$_{0:5}$)	Extended (Inst$_{21:31}$)			
X	011111	10111 10111 /			340
X	011111	11000 01010 0	**addo**	Add & record OV	223
X	011111	11000 01010 1	**addo.**	Add & record OV & CR	223
X	011111	11000 10010 /	**tlbivax**	TLB Invalidate Virtual Address Indexed	358
X	011111	11000 10110 /	**lhbrx**	Load Halfword Byte-Reverse Indexed	289
X	011111	11000 11000 0	**sraw**	Shift Right Algebraic Word	334
X	011111	11000 11000 1	**sraw.**	Shift Right Algebraic Word & record CR	334
X	011111	11001 11000 0	**srawi**	Shift Right Algebraic Word Immediate	334
X	011111	11001 11000 1	**srawi.**	Shift Right Algebraic Word Immediate & record CR	334
X	011111	11010 10110 /	**mbar**	Memory Barrier	298
X	011111	11100 10010 ?	**tlbsx**	TLB Search Indexed	360
X	011111	11100 10110 /	**sthbrx**	Store Halfword Byte-Reverse Indexed	344
X	011111	11100 11010 0	**extsh**	Extend Sign Halfword	254
X	011111	11100 11010 1	**extsh.**	Extend Sign Halfword & record CR	254
X	011111	11101 10010 /	**tlbre**	TLB Read Entry	359
X	011111	11101 11010 0	**extsb**	Extend Sign Byte	254
X	011111	11101 11010 1	**extsb.**	Extend Sign Byte & record CR	254
X	011111	11110 01011 0	**divwuo**	Divide Word Unsigned & record OV	252
X	011111	11110 01011 1	**divwuo.**	Divide Word Unsigned & record OV & CR	252
X	011111	11110 10010 /	**tlbwe**	TLB Write Entry	362
X	011111	11110 10110 /	**icbi**	Instruction Cache Block Invalidate	280
X	011111	11110 10111 /			341
X	011111	11111 01011 0	**divwo**	Divide Word & record OV	251
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 21. Instructions sorted by opcode (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
X	011111	11111 01011 1	**divwo.**	Divide Word & record OV & CR	251
X	011111	11111 10110 /	**dcbz**	Data Cache Block set to Zero	248
D	100000	----- ----- -	**lwz**	Load Word & Zero	297
D	100001	----- ----- -	**lwzu**	Load Word & Zero with Update	297
D	100010	----- ----- -	**lbz**	Load Byte & Zero	283
D	100011	----- ----- -	**lbzu**	Load Byte & Zero with Update	283
D	100100	----- ----- -	**stw**	Store Word	347
D	100101	----- ----- -	**stwu**	Store Word with Update	347
D	100110	----- ----- -	**stb**	Store Byte	337
D	100111	----- ----- -	**stbu**	Store Byte with Update	337
D	101000	----- ----- -	**lhz**	Load Halfword & Zero	290
D	101001	----- ----- -	**lhzu**	Load Halfword & Zero with Update	290
D	101010	----- ----- -	**lha**	Load Halfword Algebraic	288
D	101011	----- ----- -	**lhau**	Load Halfword Algebraic with Update	288
D	101100	----- ----- -	**sth**	Store Halfword	343
D	101101	----- ----- -	**sthu**	Store Halfword with Update	343
D	101110	----- ----- -	**lmw**	Load Multiple Word	291
D	101111	----- ----- -	**stmw**	Store Multiple Word	345
D	110000	----- ----- -			287
D	110001	----- ----- -			287
D	110010	----- ----- -			286
D	110011	----- ----- -			286
D	110100	----- ----- -			342

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 21. Instructions sorted by opcode (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 page
	Primary (Inst$_{0:5}$)	Extended (Inst$_{21:31}$)			
D	110101	----- ----- -			342
D	110110	----- ----- -			340
D	110111	----- ----- -			340
A	111011	----- 10010 0			264
A	111011	----- 10010 1			264
A	111011	----- 10100 0			279
A	111011	----- 10100 1			279
A	111011	----- 10101 0			256
A	111011	----- 10101 1			256
A	111011	----- 10110 0			278
A	111011	----- 10110 1			278
A	111011	----- 11000 0			272
A	111011	----- 11000 1			272
A	111011	----- 11001 0			268
A	111011	----- 11001 1			268
A	111011	----- 11100 0			267
A	111011	----- 11100 1			267
A	111011	----- 11101 0			265
A	111011	----- 11101 1			265
A	111011	----- 11110 0			271
A	111011	----- 11110 1			271
A	111011	----- 11111 0			270
A	111011	----- 11111 1			270

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 21. Instructions sorted by opcode (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 page
	Primary ($\text{Inst}_{0:5}$)	Extended ($\text{Inst}_{21:31}$)			
A	111111	----- 10010 0			264
A	111111	----- 10010 1			264
A	111111	----- 10100 0			279
A	111111	----- 10100 1			279
A	111111	----- 10101 0			256
A	111111	----- 10101 1			256
A	111111	----- 10110 0			278
A	111111	----- 10110 1			278
A	111111	----- 10111 0			277
A	111111	----- 10111 1			277
A	111111	----- 11001 0			268
A	111111	----- 11001 1			268
A	111111	----- 11010 0			276
A	111111	----- 11010 1			276
A	111111	----- 11100 0			267
A	111111	----- 11100 1			267
A	111111	----- 11101 0			265
A	111111	----- 11101 1			265
A	111111	----- 11110 0			271
A	111111	----- 11110 1			271
A	111111	----- 11111 0			270
A	111111	----- 11111 1			270
X	111111	00000 00000 /			259

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 21. Instructions sorted by opcode (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
X	111111	00000 01100 0			273
X	111111	00000 01100 1			273
X	111111	00000 01110 0			262
X	111111	00000 01110 1			262
X	111111	00000 01111 0			262
X	111111	00000 01111 1			262
X	111111	00001 00000 /			259
X	111111	00001 00110 0			308
X	111111	00001 00110 1			308
X	111111	00001 01000 0			269
X	111111	00001 01000 1			269
X	111111	00010 00000 /			300
X	111111	00010 00110 0			308
X	111111	00010 00110 1			308
X	111111	00010 01000 0			266
X	111111	00010 01000 1			266
X	111111	00100 00110 0			310
X	111111	00100 00110 1			310
X	111111	00100 01000 0			269
X	111111	00100 01000 1			269
X	111111	01000 01000 0			255
X	111111	01000 01000 1			255
X	111111	10010 00111 0			303

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 21. Instructions sorted by opcode (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
X	111111	10010 00111 1			303
XFL	111111	10110 00111 0			309
XFL	111111	10110 00111 1			309
X	111111	11001 01110 /			260
X	111111	11001 01111 /			260
X	111111	11010 01110 /			257

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

NOTES:
[1] Motorola Book E cache locking APU, refer to Section 11.12, Cache line locking/unlocking APU.

Chapter 4
Instruction Pipeline and Execution Timing

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

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

4.1 Overview of operation

A block diagram of the e200z759n3 core is shown in Figure 17. The instruction fetch unit prefetches instructions from memory into the instruction buffers. The decode unit decodes each instruction and generates information needed by the branch unit and the execution units. Prefetched instructions are written into the instruction buffers.

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

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

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

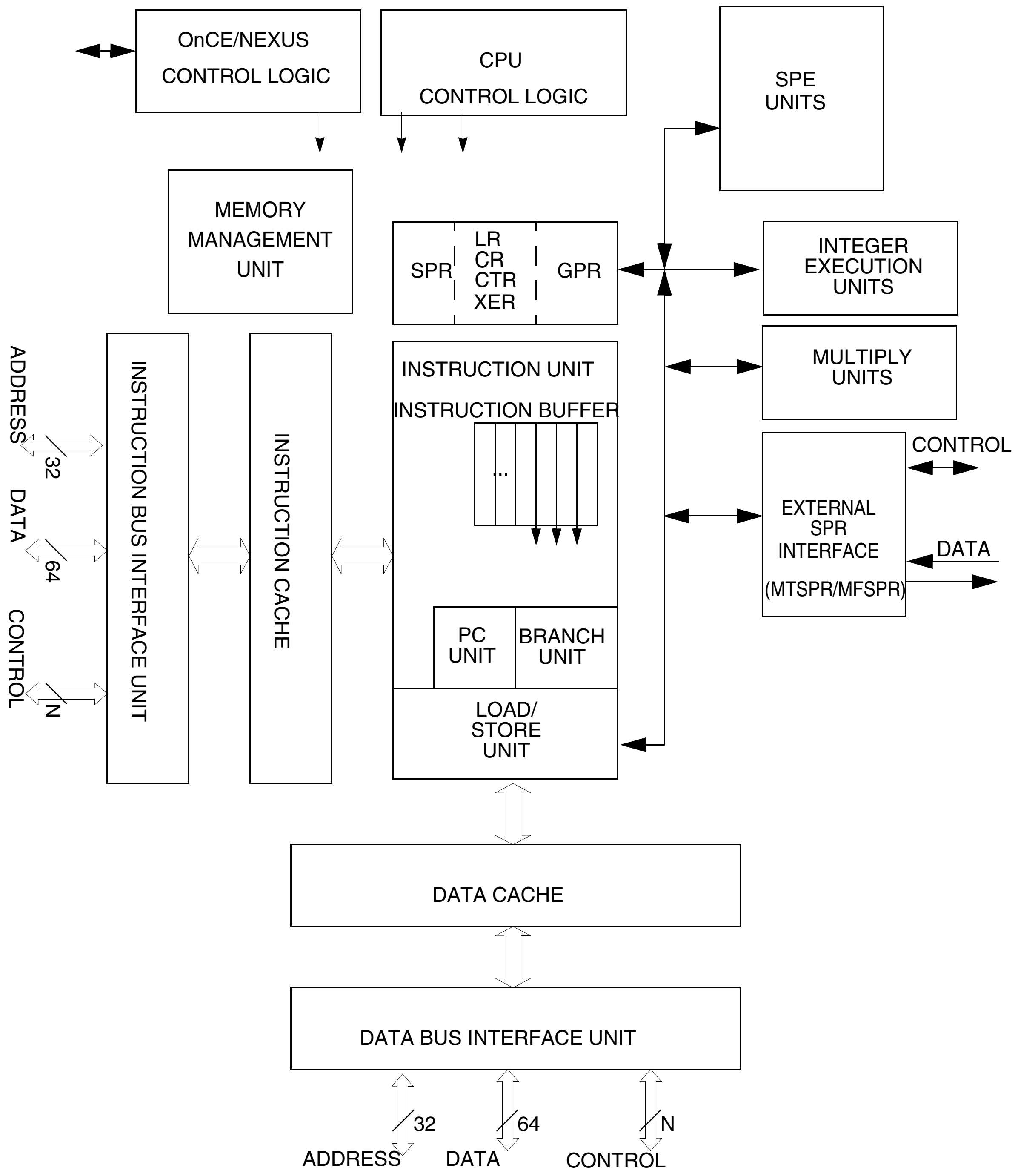


Figure 17. Zen block diagram

Table 22 shows the e200z759n3 concurrent instruction issue capabilities. Note that data dependencies between instructions will generally preclude dual-issue. in particular, read after write dependencies are handled by stalling the issue pipeline as required to ensure the proper execution ordering.

Table 22. Concurrent instruction issue capabilities

Class of instruction	Branch	Load/ store	Scalar integer	Scalar float	Vector integer	Vector float	Special
branch	—	4	4	4	4	4	—
load/store	4	—	4	4	4	4	—
scalar integer	4	4	4[1]	4	4[2]	4	—
scalar float	4	4	4	4	4	—	—
vector integer	4	4	4[2]	4	4[3]	4	—
vector float	4	4	4	—	4	—	—
special	—	—	—	—	—	—	—

NOTES:

1 excludes divide class instructions occurring in both issue slots

2 excludes vector MAC/multiply class instructions occurring with scalar multiply, or divide class instructions occurring in both issue slots

3 excludes vector MAC/multiply class instructions occurring in both issue slots, or divide class instructions occurring in both issue slots

4.1.1 Control unit

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

4.1.2 Instruction unit

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

4.1.3 Branch unit

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

4.1.4 Instruction decode unit

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

- Opcode decoding to determine the instruction class and resource requirements for each instruction being decoded.
- Source and destination register dependency checking.
- Execution unit assignment.

- Determine any decode serializations, and inhibit subsequent instruction decoding.

The decode unit operates in a single processor clock cycle.

4.1.5 Exception handling

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

4.2 Execution units

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

4.2.1 Integer execution units

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

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

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

4.2.2 Load / store unit

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

4.2.3 Embedded floating-point execution units

The embedded floating-point execution units are used to process EFPU floating-point arithmetic instructions. Adds, subtracts, compares, multiply, and multiply-accumulate pipelines have a latency of 4 cycles with a maximum throughput of 1 per cycle. EFPU floating-point divide and square root instructions have a latency of 9 cycles. While the divide is running, the rest of the pipeline is unavailable for additional instructions (blocking divide).

4.3 Instruction pipeline

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

Instructions proceed from this buffer to the instruction decode stage by entering the instruction decode register IR.

Table 23. Pipeline stages

Stage	Description
IFETCH0	Instruction Fetch From Memory, stage 0
IFETCH1	Instruction Fetch From Memory, stage 1
IFETCH2	Instruction Fetch From Memory, stage 2
DECODE0	Instruction Decode, stage 0
DECODE1 / RF READ	Instruction Decode, stage 1 / Register Read/ Operand Forwarding / Memory Effective Address Generation
EXECUTE0 / MEM0	Instruction Execution stage 0 / Memory Access stage 0
EXECUTE1 / MEM1	Instruction Execution stage 1 / Memory Access stage 1
EXECUTE2 / MEM2	Instruction Execution stage 2 / Memory Access stage 2
EXECUTE3	Instruction Execution stage 3
WB	Write Back to Registers

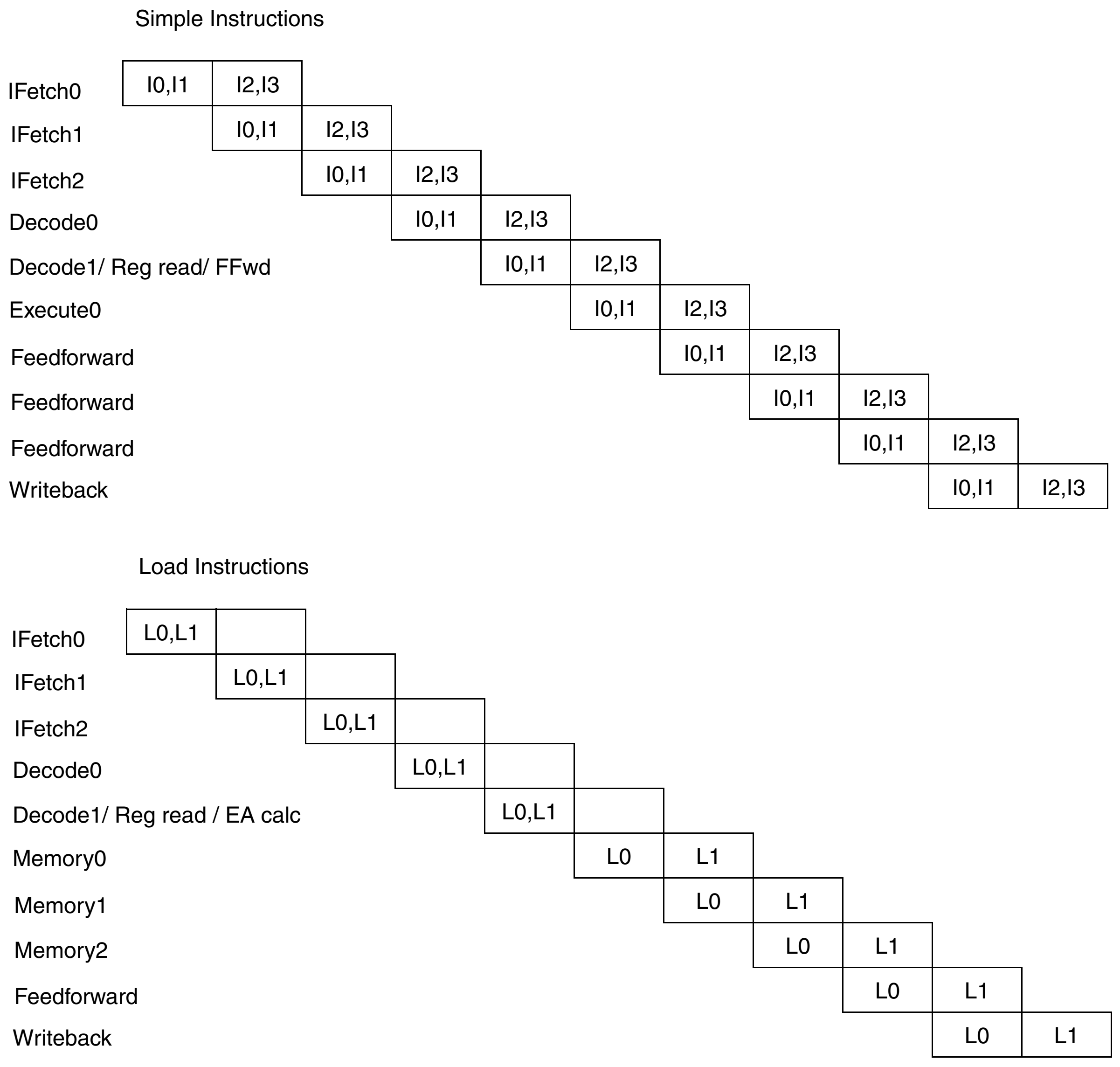


Figure 18. Pipeline diagram

4.3.1 Description of pipeline stages

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

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

Execution occurs in one or more of the four execute pipeline stages in each execution unit (perhaps over multiple cycles). Execution of most load/store instructions is pipelined. The load/store unit has four pipeline stages. The pipeline stages are: effective address calculation (EA Calc), memory access (MEM0, MEM1), and data format and forward (MEM2).

Simple integer instructions complete execution in the Execute 0 stage of the pipeline. Multiply instructions require all four execute stages but may be pipelined as well. Most condition-setting instructions complete in the Execute 0 stage of the pipeline, thus conditional branches dependent on a condition-setting instruction may be resolved by an instruction in this stage.

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

4.3.2 Instruction prefetch buffers and branch target buffer

Zen contains a 10-entry instruction prefetch buffer that supplies instructions into the Instruction Register (IR) for decoding. Each slot in the prefetch buffer is 32 bits wide, capable of holding a single 32-bit instruction, or a pair of 16-bit instructions.

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

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

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

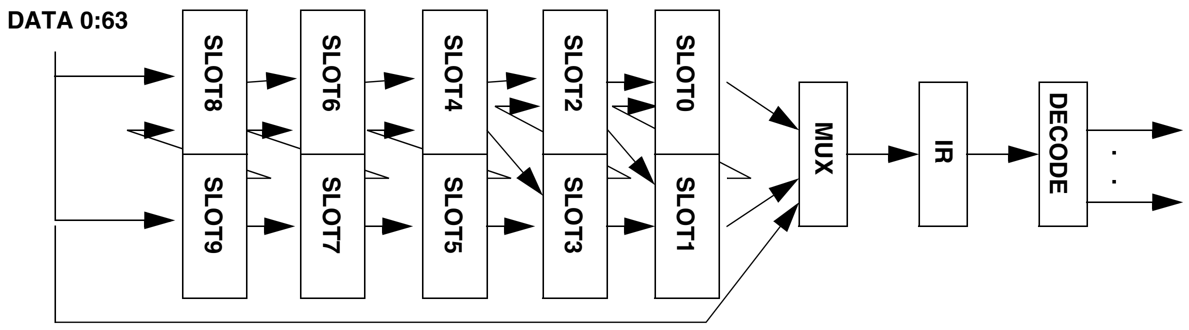


Figure 19. Zen instruction prefetch buffers

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

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

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

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

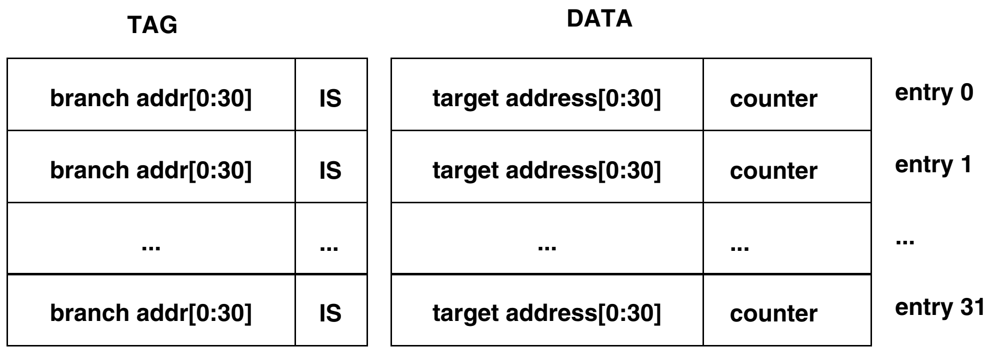
When a branch is predicted taken and the branch is later resolved (in the branch execute stage), the value of the appropriate BTB counter is updated. If a branch whose counter indicates weakly taken is resolved as taken, the counter increments so that the prediction becomes strongly taken. If the branch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

Zen does not implement the static branch prediction that is defined by the Power Architecture architecture. The BO prediction bit in branch encodings is ignored.

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

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

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


TAG
DATA
branch addr[0:30] | IS | target address[0:30] | counter | entry 0
branch addr[0:30] | IS | target address[0:30] | counter | entry 1
... | ... | ... | ... | ...
branch addr[0:30] | IS | target address[0:30] | counter | entry 31

IS = Instruction Space

Figure 20. Zen branch target buffer

4.3.3 Single-cycle instruction pipeline operation

Sequences of single-cycle execution instructions follow the flow in Figure 21. Instructions are issued and completed in program order. Most arithmetic and logical instructions fall into this category.

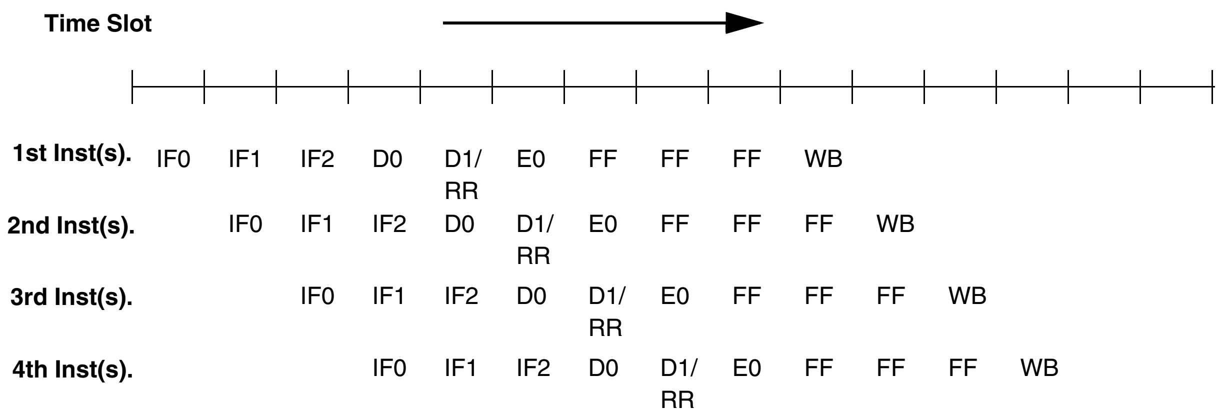

Time Slot
1st Inst(s). IF0 IF1 IF2 D0 D1/RR E0 FF FF FF WB
2nd Inst(s). IF0 IF1 IF2 D0 D1/RR E0 FF FF FF WB
3rd Inst(s). IF0 IF1 IF2 D0 D1/RR E0 FF FF FF WB
4th Inst(s). IF0 IF1 IF2 D0 D1/RR E0 FF FF FF WB

Figure 21. Basic pipeline flow, single cycle instructions

4.3.4 Basic load and store instruction pipeline operation

For load and store instructions, the effective address is calculated in the EA Calc stage, and memory is accessed in the MEM0–MEM1 stages. Data selection and alignment is performed in MEM2, and the result is available at the end of MEM2 for the following instruction.

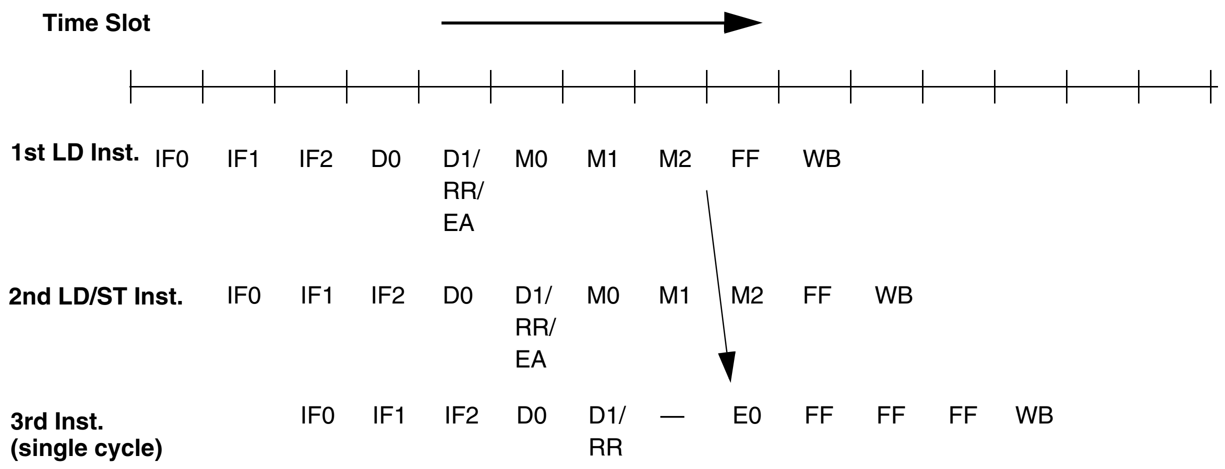


Figure 22. Basic pipeline flow, load/store instructions

4.3.5 Change-of-flow instruction pipeline operation

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

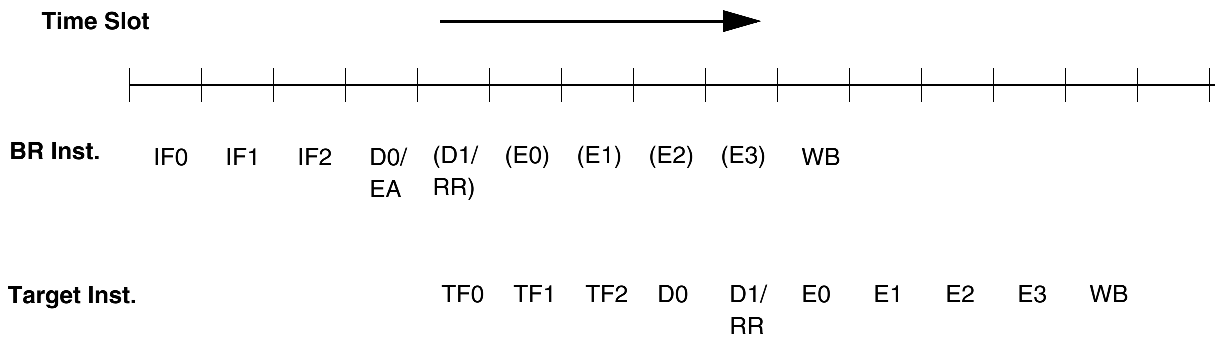


Figure 23. Basic pipeline flow, branch instructions, no prediction

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

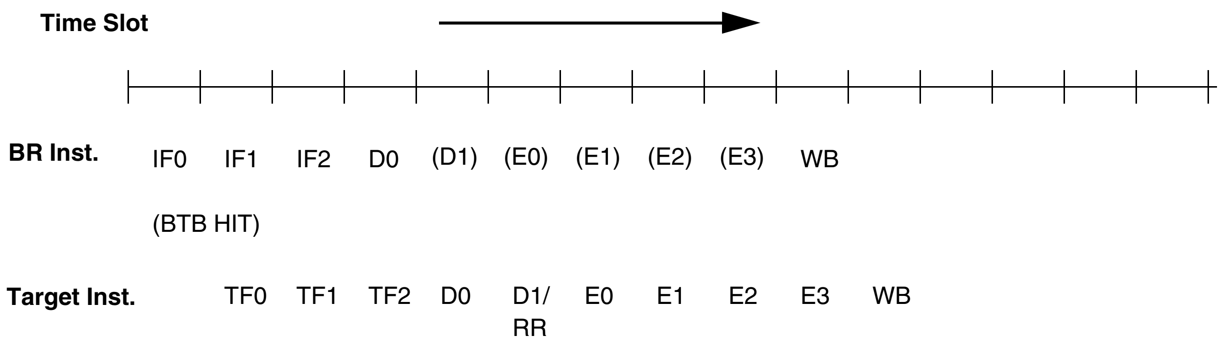


Figure 24. Basic pipeline flow, branch instructions, BTB hit, correct prediction, branch taken

For certain cases where the branch is incorrectly predicted, 6 cycles are required to correct the misprediction outcome. Figure 25 shows one example.

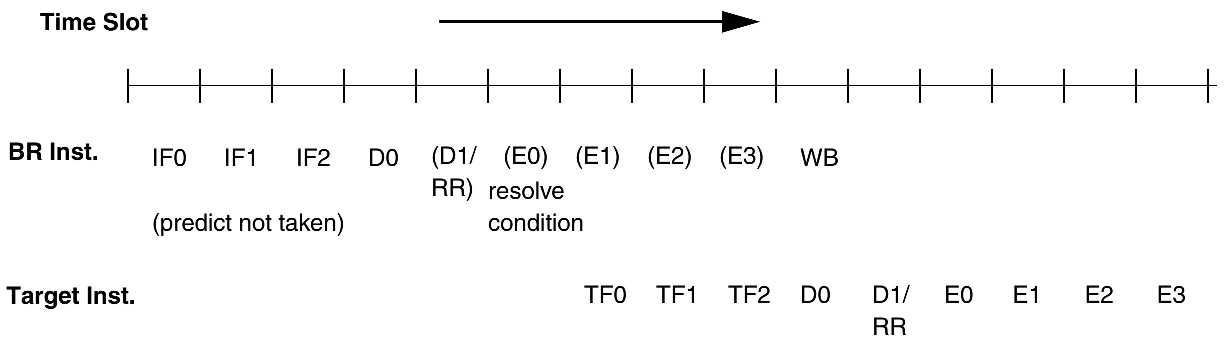


Figure 25. Basic pipeline flow, branch instructions, predict not taken, incorrect prediction

For **bcctr** and **e_bctr** cases where the branch is correctly predicted as taken, 5 cycles are required to execute the branch as shown in Figure 26.

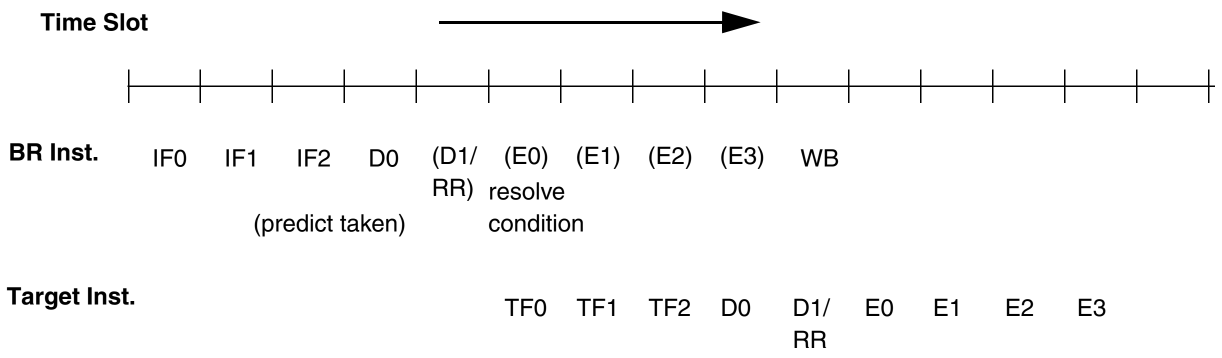


Figure 26. Basic pipeline flow, bcctr Instruction, predict taken, correct prediction

For **bcctr** and **e_bctr** cases where the branch is incorrectly predicted as taken, but the fall-through instruction is already in the instruction buffer, 3 cycles are required to execute the branch as shown in Figure 25.

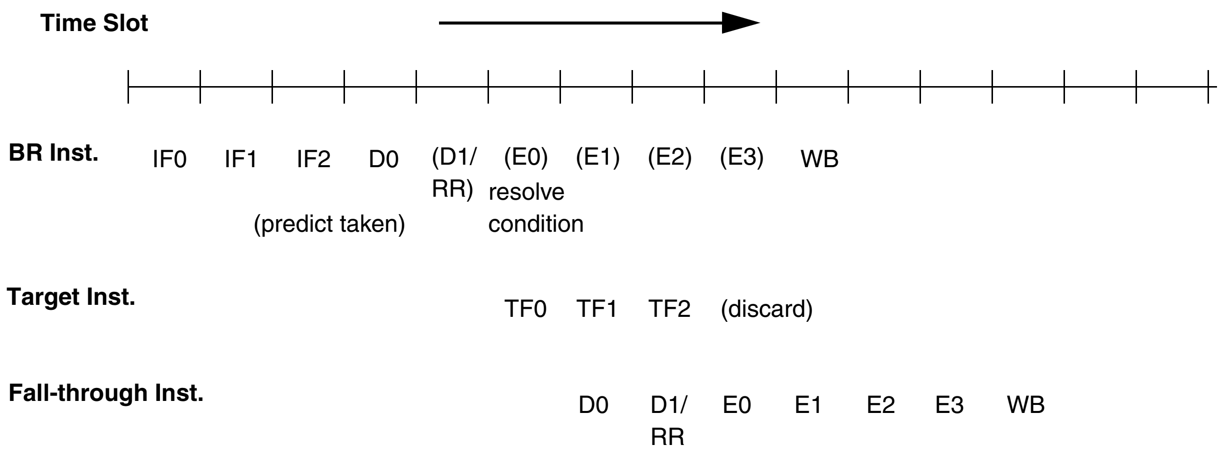


Figure 27. Basic pipeline flow, bcctr Instruction, predict taken, incorrect prediction, instruction buffer not empty

For **bcctr** and **e_bctr** cases where the branch is incorrectly predicted as taken, and the fall-through instruction is not already in the instruction buffer (a rare case), 6 cycles are required to execute the branch as shown in Figure 25.

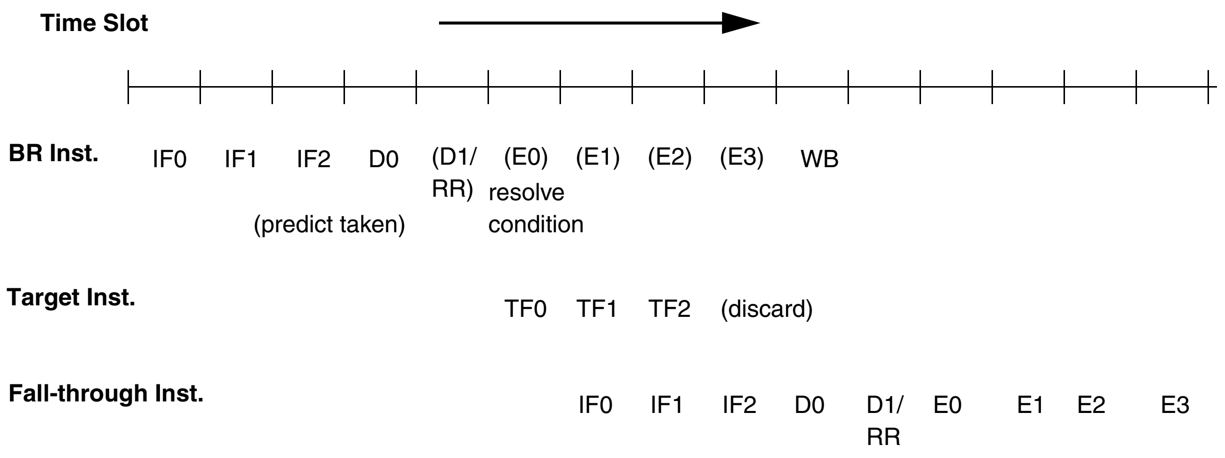


Figure 28. Basic pipeline flow, bcctr Instruction, predict taken, incorrect prediction, instruction buffer empty

4.3.6 Basic multi-cycle instruction pipeline operation

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

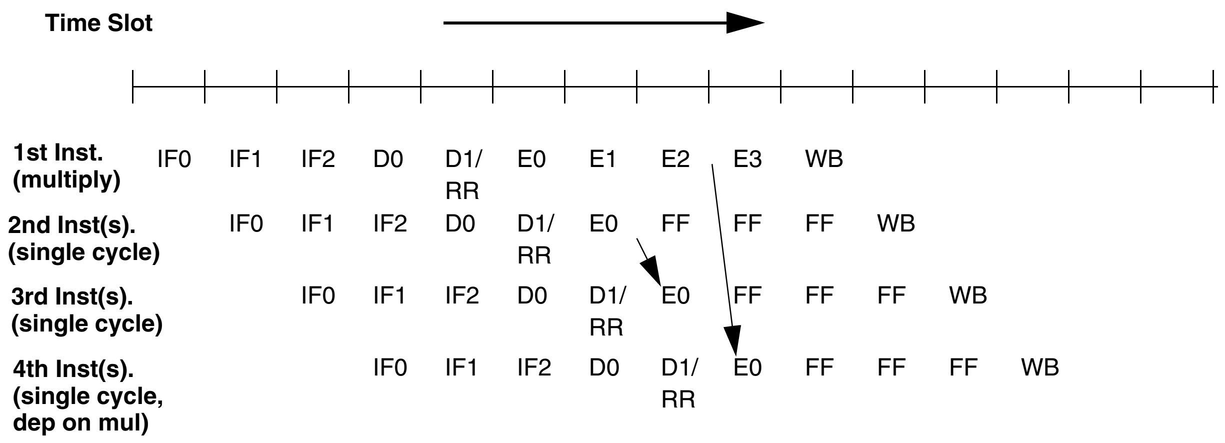

Time Slot
1st Inst. (multiply)
IF0 IF1 IF2 D0 D1/RR E0 E1 E2 E3 WB
2nd Inst(s). (single cycle)
IF0 IF1 IF2 D0 D1/RR E0 FF FF FF WB
3rd Inst(s). (single cycle)
IF0 IF1 IF2 D0 D1/RR E0 FF FF FF WB
4th Inst(s). (single cycle, dep on mul)
IF0 IF1 IF2 D0 D1/RR E0 FF FF FF WB

Figure 29. Basic pipeline flow, integer multiply class instructions

The divide and load and store multiple instructions require multiple cycles in the execute stage.

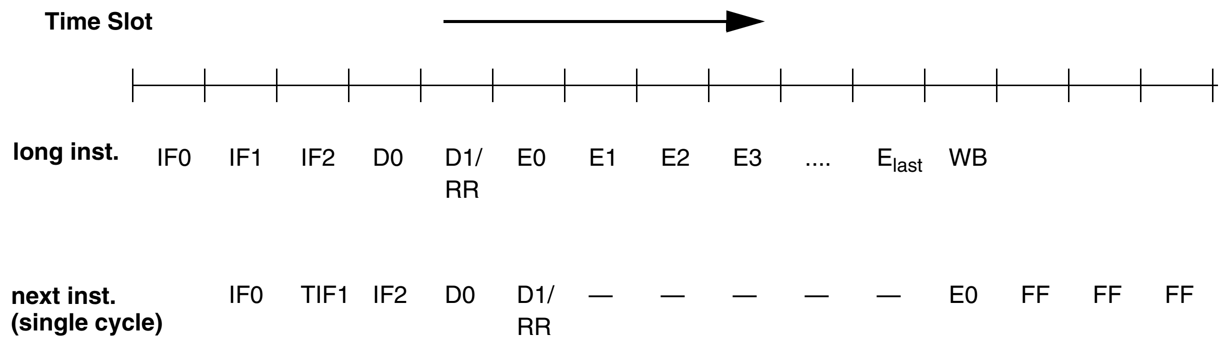

Time Slot
long inst.
IF0 IF1 IF2 D0 D1/RR E0 E1 E2 E3 .... Elast WB
next inst. (single cycle)
IF0 TIF1 IF2 D0 D1/RR — — — — — E0 FF FF FF

Figure 30. Basic pipeline flow, long instruction

4.3.7 Additional examples of instruction pipeline operation for load and store

Figure 31 shows an example of pipelining two non-data-dependent load or store instructions with a following load target data-dependent single cycle instruction. While the first load or store begins accessing memory in the M0 stage, the next load can be calculating a new effective address in the D1/EA stage. The **add** in this example will stall for two cycles since a data dependency exists on the target register of the second load.

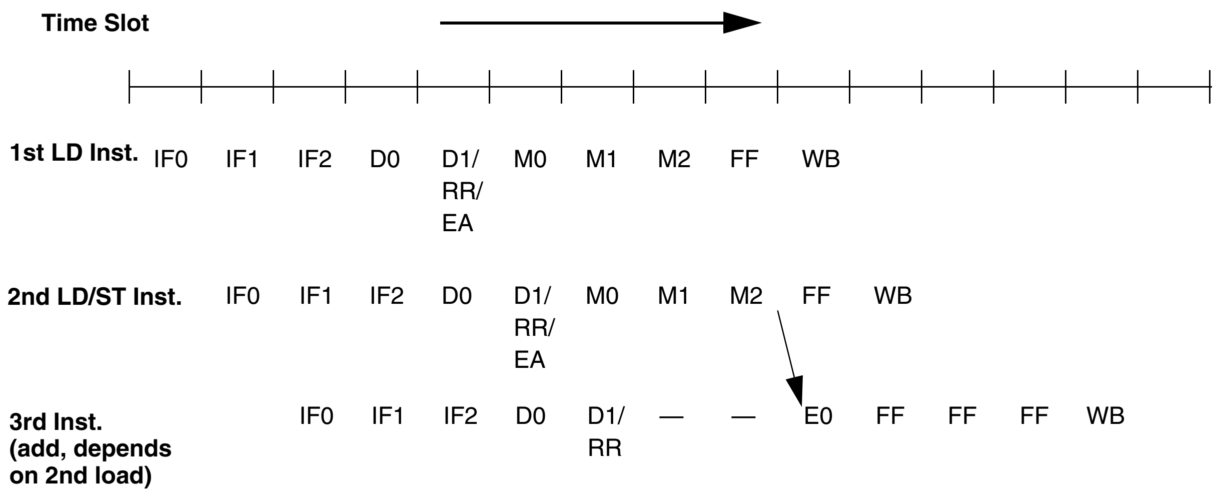


Figure 31. Pipelined load instructions with load target data dependency

Figure 32 shows an example of pipelining a data-dependent add instruction following a load with update instruction. While the first load begins accessing memory in the M0 stage, the next load with update can be calculating a new effective address in the EA Calc stage. Following the EA Calc, the updated base register value can be fed-forward to subsequent instructions. The **add** in this example will not stall, even though a data dependency exists on the updated base register of the load with update.

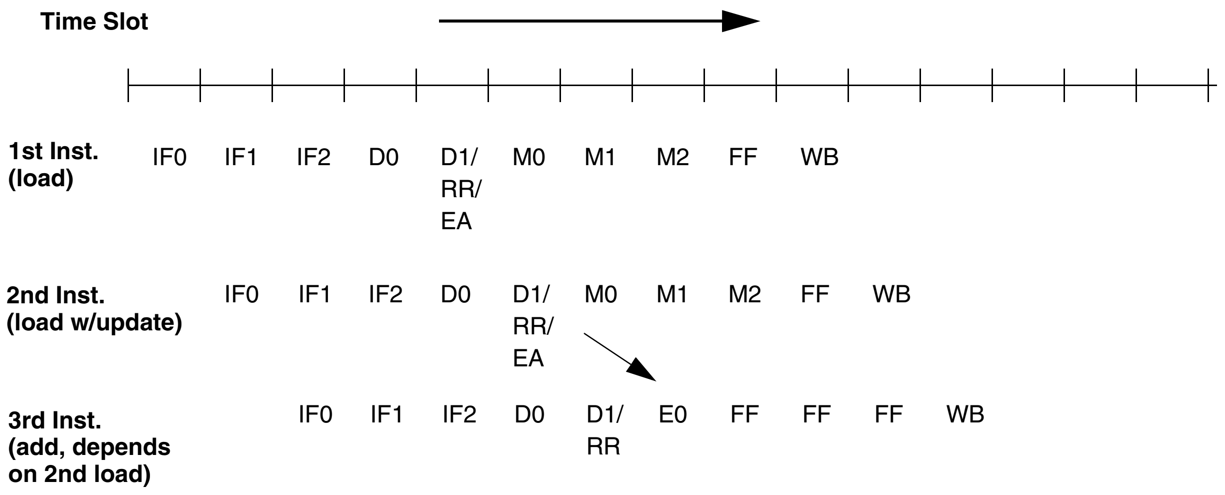


Figure 32. Pipelined instructions with base register update data dependency

Figure 33 shows an example of pipelining a data-dependent store instruction following a load instruction. While the first load begins accessing memory in the M0 stage, the store can be calculating a new effective address in the D1/EA stage. The **store** in this example will not stall due to the data dependency existing on the load data of the load instruction.

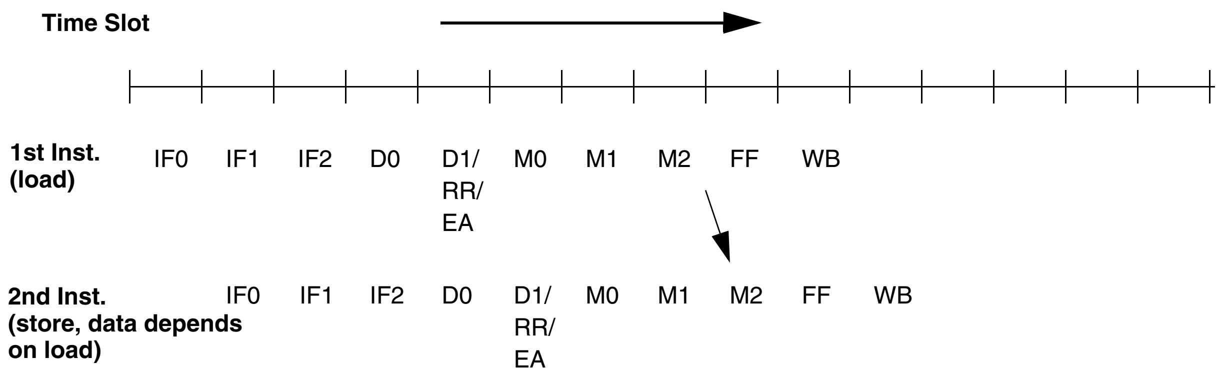

Time Slot
1st Inst. (load)
IF0 IF1 IF2 D0 D1/ RR/ EA M0 M1 M2 FF WB
2nd Inst. (store, data depends on load)
IF0 IF1 IF2 D0 D1/ RR/ EA M0 M1 M2 FF WB

Figure 33. Pipelined store instruction with store data dependency

4.3.8 Move to/from SPR instruction pipeline operation

Many **mtspr** and **mfspr** instructions are treated like single cycle instructions in the pipeline, and do not cause stalls. Exceptions are for the MSR, the Debug SPRs, the SPE Unit, and Cache/MMU SPRs, which do cause stalls. Figure 34 through Figure 36 show examples of **mtspr** and **mfspr** instruction timing.

Figure 34 applies to the Debug SPRs and the SPE APU's SPEFSCR. These instructions do not begin execution until all previous instructions have finished their execute stage(s). In addition, execution of subsequent instructions is stalled until the **mfspr** and **mtspr** instructions complete.

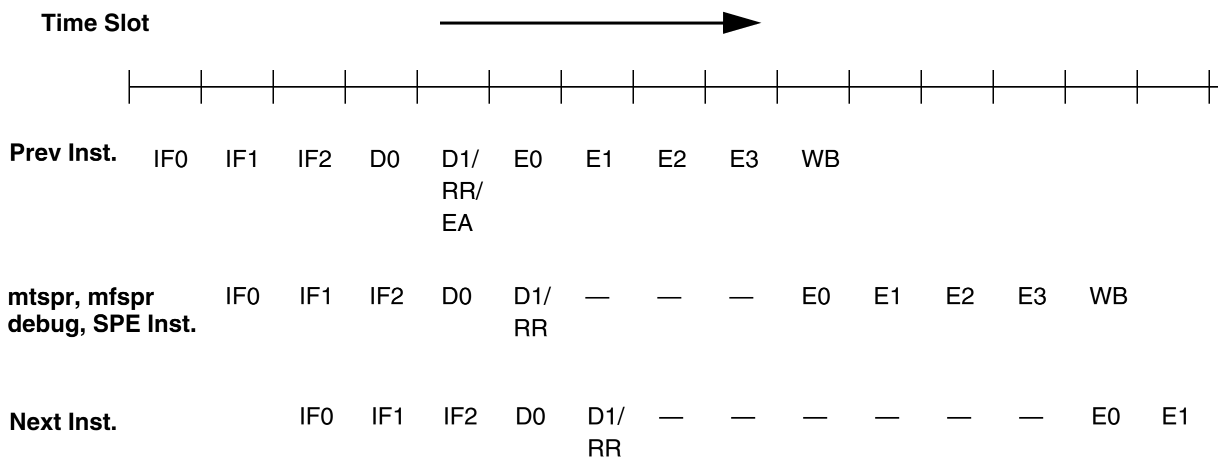

Time Slot
Prev Inst.
IF0 IF1 IF2 D0 D1/ RR/ EA E0 E1 E2 E3 WB
mtspr, mfspr debug, SPE Inst.
IF0 IF1 IF2 D0 D1/ RR — — — E0 E1 E2 E3 WB
Next Inst.
IF0 IF1 IF2 D0 D1/ RR — — — — — — E0 E1

Figure 34. mtspr, mfspr instruction execution, debug and SPE SPRs

Figure 35 applies to the **mtmsr** instruction and the **wrtee** and **wrteei** instructions. Execution of subsequent instructions is stalled until the cycle after these instructions writeback.

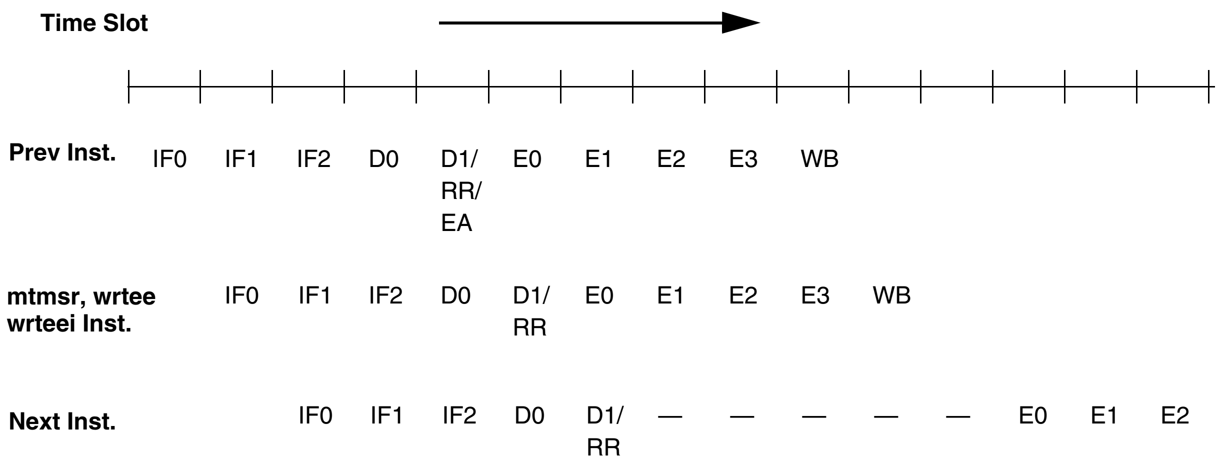


Figure 35. mtmsr, wrtee[i] instruction execution

Access to cache and MMU SPRs are stalled until all outstanding bus accesses have completed on both interfaces and the caches and MMU are idle (**p_[d,i]_cmbusy** negated) to allow an access window where no translations or cache cycles are required. Figure 36 shows an example where an outstanding bus access causes **mtspr**/**mfspr** execution to be delayed until the bus becomes idle. Other situations such as a cache linefill may cause the cache to be busy even when the processor interface is idle (**p_[d,i]_tbusy[0]_b** is negated). In these cases execution stalls until the cache and MMU are idle as signaled by negation of **p_[d,i]_cmbusy**. Processor access requests will be held off during execution of a Cache/MMU SPR instruction. A subsequent access request may be generated the cycle following the last execute stage (i.e. during the WB cycle). This same protocol applies to cache and MMU management instructions (e.g. **dcbz**, **dcbf**, etc., **tlbre**, **tlbwe**, etc.).

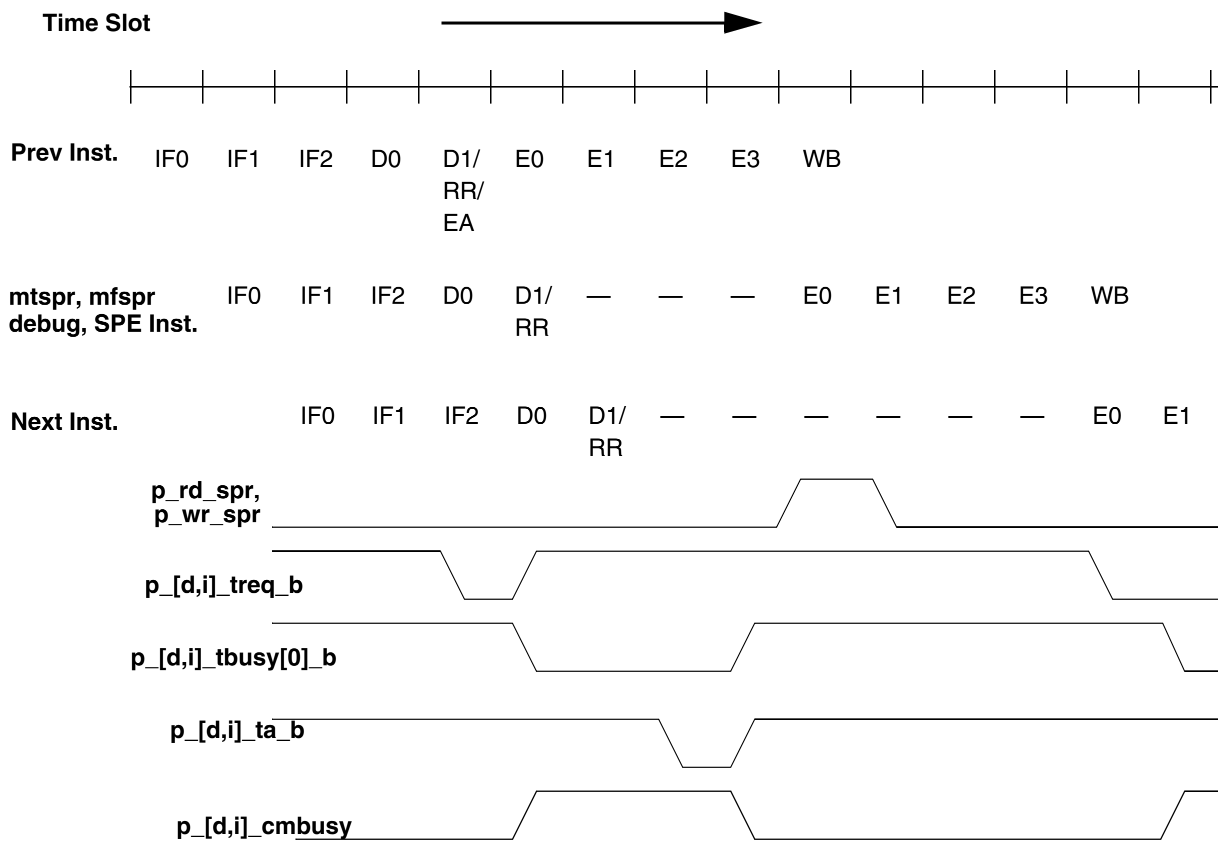


Figure 36. Cache / MMU mtspr, mfspr and management instruction execution

4.4 Control hazards

Several internal control hazards exist in Zen that can cause certain instruction sequences to incur one or more stall cycles. These include:

- **mfspr** instruction preceded by a **mtspr** instruction — issue stalls until the **mtspr** completes

4.5 Instruction serialization

There are three types of serialization required by the core:

- Completion serialization
- Dispatch (Decode/Issue) serialization
- Refetch serialization

4.5.1 Completion serialization

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

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

4.5.2 Dispatch serialization

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

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

4.5.3 Refetch serialization

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

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

Figure 39 shows

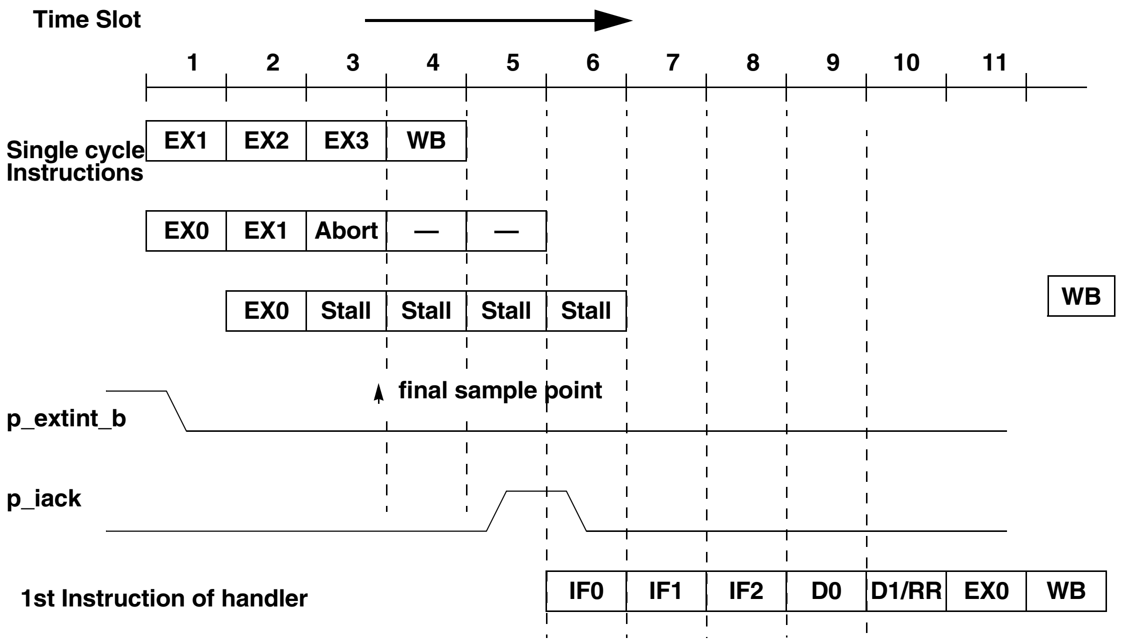


Figure 37. Interrupt recognition and handler instruction execution

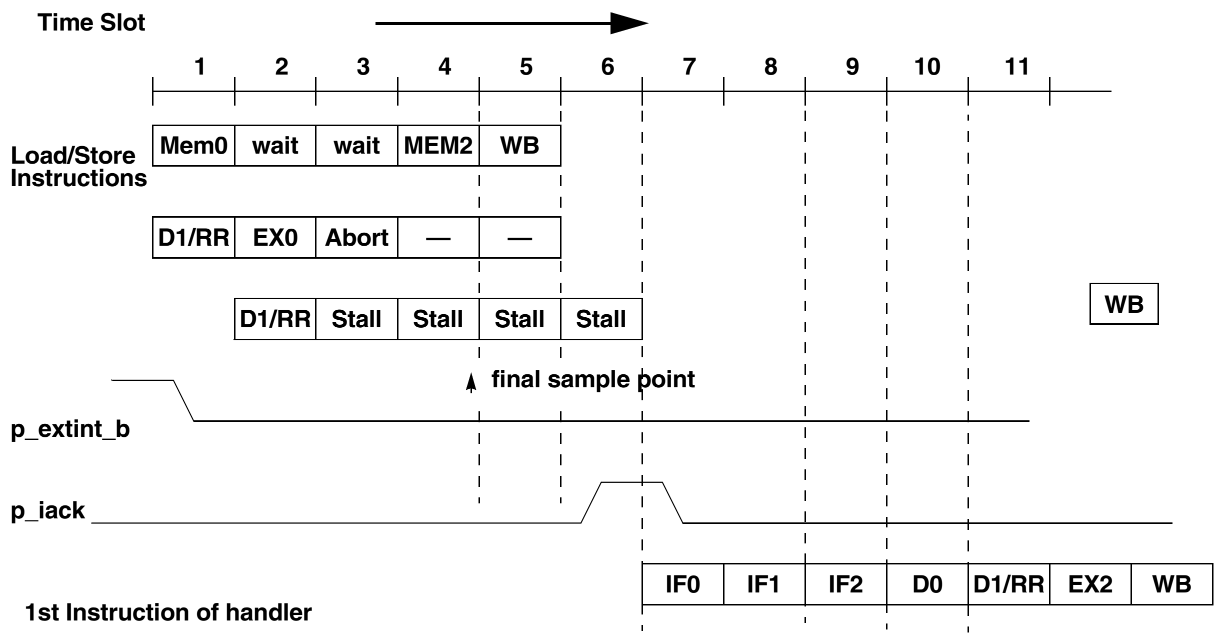


Figure 38. Interrupt recognition and handler instruction execution —load/store in progress

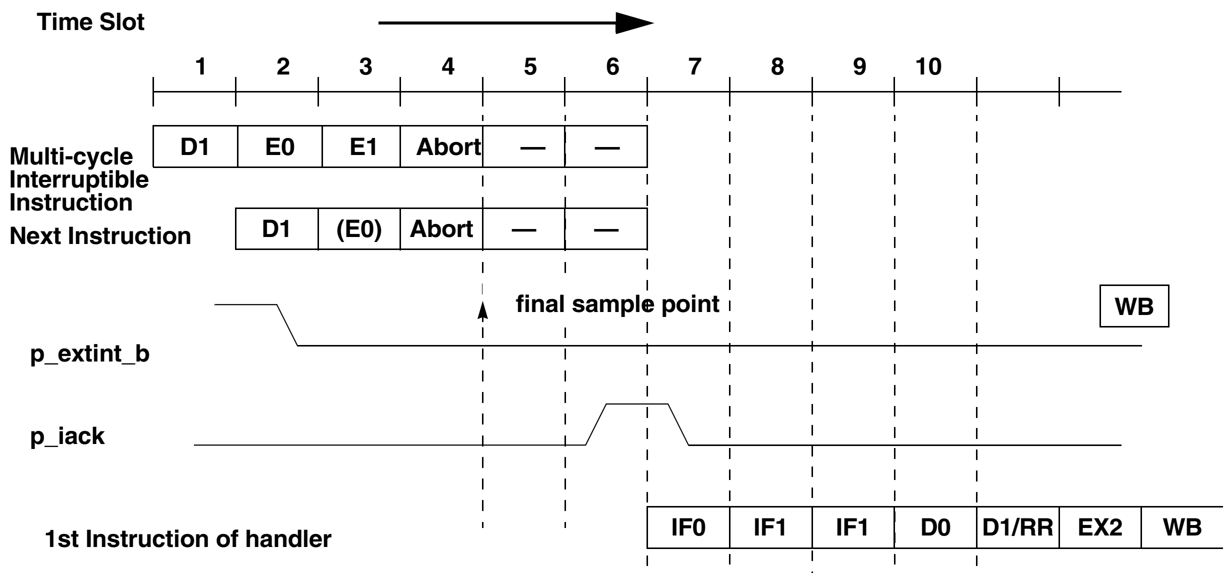


Figure 39. Interrupt recognition and handler instruction execution — multi-cycle instruction abort

4.6 Concurrent instruction execution

The core effectively has several execution units:

- Branch unit
- Dual scalar integer units
- Dual vector integer units
- Dual scalar Embedded Floating-point units/ Single vector Embedded Floating-point unit

- Load/store unit

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

The following instruction timing section accurately indicates the number of cycles an instruction executes in the appropriate unit, however, determining the elapsed time or cycles to execute a sequence of instructions is beyond the scope of this document.

4.7 Instruction Timings

Instruction timing in number of processor clock cycles for various instruction classes is shown in Table 24. Pipelined instructions are shown with cycles of total latency and throughput cycles. Divide instructions are not pipelined and block other instructions from executing during divide execution. Timing for SPE instructions is detailed in Section 6.6, SPE instruction timing.

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

Table 24. Instruction class cycle counts

Class of Instructions	Latency	Throughput	Special notes
integer: **add**, **sub**, **shift**, **rotate**, **logical**, **cntlzw**	1	1	—
integer: compare	1	1	—
Branch	6/4/1	6/4/1	Correct branch lookahead allows single cycle execution Worst-case mispredicted branch is 6 cycles
multiply	3/4	1	result data is available after 3 cycles, record form conditions are available after 4th cycle
divide	4-15	4-15	data-dependent timing
CR logical	1	1	—
loads (non-multiple)	3	1	—
load multiple	3 + n/2 (max)	1 + n/2 (max)	Actual timing depends on n and address alignment.
stores (non-multiple)	3	1	—
store multiple	3 + n/2 (max)	1 + n/2 (max)	Actual timing depends on n and address alignment.
mtmsr, **wrtee**, **wrteei**	6&	6	
mcrf	1	1	
mfspr, **mtspr**	4&	4&	Applies to Debug SPRs, optional unit SPRS
mfspr, **mfmsr**	1	1	Applies to internal, non Debug SPRs

Table 24. Instruction class cycle counts (continued)

Class of Instructions	Latency	Throughput	Special notes
mfcr, mtcr	1	1	—
rfi, **rfci**, **rfdi, rfmci**	6	—	—
sc	4	—	—
tw, **twi**	4	—	Trap taken timing

Detailed timing for each instruction mnemonic along with serialization requirements is shown in Table 25.

Table 25. Instruction Timing by Mnemonic

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

Table 25. Instruction Timing by Mnemonic (continued)

Mnemonic	Latency	Serialization
crorc	1	none
crxor	1	none
divw[o][.]	4-15[1]	none
divwu[o][.]	4-15[1]	none
eqv[.]	1	none
extsb[.]	1	none
extsh[.]	1	none
isel	1	none
isync	6[2]	refetch
lbarx	3	none
lbz	3[3]	none
lbzu	3[3]	none
lbzux	3[3]	none
lbzx	3[3]	none
lha	3[3]	none
lharx	3	none
lhau	3[3]	none
lhaux	3[3]	none
lhax	3[3]	none
lhbrx	3[3]	none
lhz	3[3]	none
lhzu	3[3]	none
lhzux	3[3]	none
lhzx	3[3]	none
lmw	3 +(n/2)	none
lwarx	3	none
lwbrx	3[3]	none
lwz	3[3]	none
lwzu	3[3]	none
lwzux	3[3]	none
lwzx	3[3]	none
mbar	1[2]	pseudo- dispatch
mcrf	1	none
mcrxr	1	completion
mfcr	1	none

Table 25. Instruction Timing by Mnemonic (continued)

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

Table 25. Instruction Timing by Mnemonic (continued)

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

NOTES:
[1] with early-out capability, timing is data dependent
[2] plus additional synchronization time
[3] Aligned

4.8 Operand placement on performance

The placement (location and alignment) of operands in memory affects relative performance of memory accesses, and in some cases, affects it significantly. Table 26 indicates the effects for the Zen core.

In Table 26, optimal means that one effective address (EA) calculation occurs during the memory operation. Good means that multiple EA calculations occur during the memory operation, which may cause additional bus activities with multiple bus transfers. Poor means that an alignment interrupt is generated by the storage operation.

Table 26. Performance effects of storage operand placement

Operand		Boundary crossing*		
Size	**Byte alignment**	**None**	**Cache line**	**Protection boundary**
4 Byte	4 < 4	optimal[1] good[2]	— good	— good
2 Byte	2 < 2	optimal good	— good	— good
1 Byte	1	optimal	—	—
lmw, stmw	4 < 4	good poor[3]	good poor	good poor
string	N/A	—	—	—

NOTES:
[1] optimal: One EA calculation occurs.
[2] good: Multiple EA calculations occur, which may cause additional bus activities with multiple bus transfers.
[3] poor: Alignment Interrupt occurs.

Chapter 5
Embedded Floating-Point APU (EFPU2)

This chapter describes the instruction set architecture of the Embedded Floating-point APU version 2 (EFPU2) implemented on e200z759n3. This unit implements scalar and vector single-precision floating-point instructions to accelerate signal processing and other algorithms. In comparison to version 1.1 of the EFPU architecture, version 2 of the architecture implements additional operations such as minimum, maximum, and square root, as well as an extensive set of vector operations with permuted operands and mixed add/sub, sum, and differences. For the remainder of this chapter, the term EFPU implies version 2 of the architecture unless otherwise noted.

5.1 Nomenclature and conventions

Several conventions regarding nomenclature are used in this chapter:

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

5.2 EFPU programming model

The e200z759n3 core provides a register file with thirty-two 64-bit registers. The Power Architecture 32-bit Book E instructions operate on the lower (least significant) 32 bits of the 64-bit register. EFPU instructions are defined that view the 64-bit register as being composed of a vector of two 32-bit elements, or a single scalar 32-bit element. Vector floating-point instructions operate on a vector of two 32-bit single-precision floating-point numbers resident in the 64-bit GPRs. Scalar single-precision floating-point instructions operate on the lower half of GPRs. The floating-point instructions do not have a separate register file; there is a single shared register file for all instructions.

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

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

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

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
SOVH	OVH	FGH	FXH	FINVH	FDBZH	FUNFH	FOVFH	0		FINXS	FINVS	FDBZS	FUNFS	FOVFS	MODE	SOV	OV	FG	FX	FINV	FDBZ	FUNF	FOVF	0	FINXE	FINVE	FDBZE	FUNFE	FOVFE	FRMC	

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

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

The SPEFSCR bits are defined in Table 5-1.

Table 5-1. SPEFSCR field descriptions

Bits	Name	Description
0 (32)	SOVH	Summary Integer Overflow High Defined by SPE.
1 (33)	OVH	Integer Overflow High Defined by SPE.
2 (34)	FGH	Embedded Floating-point Guard bit High FGH is supplied for use by the Floating-point Round exception handler. FGH is zeroed if a Floating-point Data Exception occurs for the high element(s). FGH corresponds to the high element result. FGH is cleared by a scalar floating point instruction.
3 (35)	FXH	Embedded Floating-point Sticky bit High FXH is supplied for use by the Floating-point Round exception handler. FXH is zeroed if a Floating-point Data Exception occurs for the high element(s). FXH corresponds to the high element result. FXH is cleared by a scalar floating point instruction.
4 (36)	FINVH	Embedded Floating-point Invalid Operation / Input error High In mode 0, the FINVH bit is set to 1 if the A or B high element operand of a floating-point instruction is Infinity, NaN, or Denorm, or if the operation is a divide and the high element dividend and divisor are both 0. In mode 1, the FINVH bit is set on an IEEE754 invalid operation (IEEE754-1985 sec7.1) in the high element. FINVHH is cleared by a scalar floating point instruction.
5 (37)	FDBZH	Embedded Floating-point Divide by Zero High The FDBZH bit is set to 1 when a floating-point divide instruction executed with a high element divisor of 0, and the high element dividend is a finite non-zero number. FDBZH is cleared by a scalar floating point instruction.
6 (38)	FUNFH	Embedded Floating-point Underflow High The FUNFH bit is set to 1 when the execution of a floating-point instruction results in an underflow in the high element. FUNFH is cleared by a scalar floating point instruction.
7 (39)	FOVFH	Embedded Floating-point Overflow High The FOVFH bit is set to 1 when the execution of a floating-point instruction results in an overflow in the high element. FOVFH is cleared by a scalar floating point instruction.
8:9 (40:41)	—	Reserved

Table 5-1. SPEFSCR field descriptions (continued)

Bits	Name	Description
10 (42)	FINXS	Embedded Floating-point Inexact Sticky Flag The FINXS bit is set to 1 whenever the execution of a floating-point instruction delivers an inexact result for either the low or high element and no Floating-point Data exception is taken for either element, or if the result of a Floating-point instruction results in overflow (FOVF=1 or FOVFH=1), but Floating-point Overflow exceptions are disabled (FOVFE=0), or if the result of a Floating-point instruction results in underflow (FUNF=1 or FUNFH=1), but Floating-point Underflow exceptions are disabled (FUNFE=0), and no Floating-point Data exception occurs. The FINXS bit remains set until it is cleared by a **mtspr** instruction specifying the SPEFSCR register.
11 (43)	FINVS	Embedded Floating-point Invalid Operation Sticky Flag The FINVS bit is set to a 1 when a floating-point instruction sets the FINVH or FINV bit to 1. The FINVS bit remains set until it is cleared by a **mtspr** instruction specifying the SPEFSCR register.
12 (44)	FDBZS	Embedded Floating-point Divide by Zero Sticky Flag The FDBZS bit is set to 1 when a floating-point divide instruction sets the FDBZH or FDBZ bit to 1. The FDBZS bit remains set until it is cleared by a **mtspr** instruction specifying the SPEFSCR register.
13 (45)	FUNFS	Embedded Floating-point Underflow Sticky Flag The FUNFS bit is set to 1 when a floating-point instruction sets the FUNFH or FUNF bit to 1. The FUNFS bit remains set until it is cleared by a **mtspr** instruction specifying the SPEFSCR register.
14 (46)	FOVFS	Embedded Floating-point Overflow Sticky Flag The FOVFS bit is set to 1 when a floating-point instruction sets the FOVFH or FOVF bit to 1. The FOVFS bit remains set until it is cleared by a **mtspr** instruction specifying the SPEFSCR register.
15 (47)	MODE	Embedded Floating-point Operating Mode 0 Default hardware results operating mode 1 IEEE754 hardware results operating mode (not supported by Zen) This bit controls the operating mode of the EFPU. Zen supports only mode 0. Software should read the value of this bit after writing it to determine if the implementation supports the selected mode. Implementations will return the value written if the selected mode is a supported mode, otherwise the value read will indicate the hardware supported mode.
16 (48)	SOV	Summary integer overflow Defined by SPE.
17 (49)	OV	Integer overflow Defined by SPE.
18 (50)	FG	Embedded Floating-point Guard bit FG is supplied for use by the Floating-point Round exception handler. FG is zeroed if a Floating-point Data Exception occurs for the low element(s). FG corresponds to the low element result.
19 (51)	FX	Embedded Floating-point Sticky bit FX is supplied for use by the Floating-point Round exception handler.FX is zeroed if a Floating-point Data Exception occurs for the low element(s). FX corresponds to the low element result.

Table 5-1. SPEFSCR field descriptions (continued)

Bits	Name	Description
20 (52)	FINV	Embedded Floating-point Invalid Operation / Input error In mode 0, the FINV bit is set to 1 if the A or B low element operand of a floating-point instruction is Infinity, NaN, or Denorm, or if the operation is a divide and the low element dividend and divisor are both 0. In mode 1, the FINV bit is set on an IEEE754 invalid operation (IEEE754-1985 sec7.1) in the low element.
21 (53)	FDBZ	Embedded Floating-point Divide by Zero The FDBZ bit is set to 1 when a floating-point divide instruction executed with a low element divisor of 0, and the low element dividend is a finite non-zero number.
22 (54)	FUNF	Embedded Floating-point Underflow The FUNF bit is set to 1 when the execution of a floating-point instruction results in an underflow in the low element.
23 (55)	FOVF	Embedded Floating-point Overflow The FOVF bit is set to 1 when the execution of a floating-point instruction results in an overflow in the low element.
24 (56)	—	Reserved
25 (57)	FINXE	Embedded Floating-point Inexact Exception Enable 0 Exception disabled 1 Exception enabled If the exception is enabled, a Floating-point Round exception is taken if for both elements, the result of a Floating-point instruction does not result in overflow or underflow, and the result for either element is inexact (FG | FX = 1, or FGH | FXH =1), or if the result of a Floating-point instruction does result in overflow (FOVF=1 or FOVFH=1) for either element, but Floating-point Overflow exceptions are disabled (FOVFE=0), or if the result of a Floating-point instruction results in underflow (FUNF=1 or FUNFH=1), but Floating-point Underflow exceptions are disabled (FUNFE=0), and no Floating-point Data exception occurs.
26 (58)	FINVE	Embedded Floating-point Invalid Operation / Input Error Exception Enable 0 Exception disabled 1 Exception enabled If the exception is enabled, a Floating-point Data exception is taken if the FINV or FINVH bit is set by a floating-point instruction.
27 (59)	FDBZE	Embedded Floating-point Divide by Zero Exception Enable 0 Exception disabled 1 Exception enabled If the exception is enabled, a Floating-point Data exception is taken if the FDBZ or FDBZH bit is set by a floating-point instruction.
28 (60)	FUNFE	Embedded Floating-point Underflow Exception Enable 0 Exception disabled 1 Exception enabled If the exception is enabled, a Floating-point Data exception is taken if the FUNF or FUNFH bit is set by a floating-point instruction.

Table 5-1. SPEFSCR field descriptions (continued)

Bits	Name	Description
29 (61)	FOVFE	Embedded Floating-point Overflow Exception Enable 0 Exception disabled 1 Exception enabled If the exception is enabled, a Floating-point Data exception is taken if the FOVF or FOVFH bit is set by a floating-point instruction.
30:31 (62:63)	FRMC	Embedded Floating-point Rounding Mode Control 00 Round to Nearest 01 Round toward Zero 10 Round toward +Infinity 11 Round toward -Infinity

5.2.2 GPRs and *PowerISA 2.06* instructions

The e200z759n3 core implements the 32-bit forms of the Book E instructions. All 32-bit *PowerISA 2.06* instructions operate upon the lower half of the 64-bit GPR. These instructions do not affect the upper half of a GPR.

5.2.3 SPE/EFPU available bit in MSR

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

5.2.4 Embedded floating-point exception bit in ESR

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

5.2.5 EFPU exceptions

The architecture defines the following Embedded Floating-point APU exceptions:

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

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

5.2.5.1 EFPU unavailable exception

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

- SRR0 is set to the effective address of the instruction causing the exception.
- SRR1 is set to the contents of the MSR at the time of the exception.
- $MSR_{CE,ME,DE}$ are unchanged. All other bits are cleared.
- The ESR_{SPE} bit is set. All other ESR bits are cleared.

Instruction execution resumes at address $IVPR_{0:15}||ivor32_{16:27}||0b0000$.

5.2.5.2 Embedded floating-point data exception

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

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

Instruction execution resumes at address $IVPR_{0:15}||IVOR33_{16:27}||0b0000$.

5.2.5.3 Embedded floating-point round exception

The embedded floating-point round exception occurs if the $SPEFSCR_{FINXE}$ bit is set and either the unrounded result of an operation is not exact, or an overflow occurs and overflow exceptions are disabled (FOVF or FOVFH set with FOVFE cleared), or if an underflow occurs and underflow exceptions are disabled (FUNF set with FUNFE cleared), and no floating-point data exception is taken. The embedded floating-point round exception will not occur if an enabled embedded floating-point data exception occurs.

When the embedded floating-point round exception occurs, the unrounded (truncated) result of an inexact high or low element is placed in the target register. If only a single element is inexact, the other exact element will be updated with the correctly rounded result. The FG and FX bits corresponding to the other exact element will both be ‘0’.

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

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

- SRR0 is set to the effective address of the instruction following the instruction causing the exception.
- SRR1 is set to the contents of the MSR at the time of the exception.
- MSR bits CE, ME and DE are unchanged. All other bits are cleared.
- The ESR_{SPE} bit is set. All other ESR bits are cleared.
- $SPEFSCR_{FGH, FG, FXH, FX}$ are set appropriately. $SPEFSCR_{FINXS}$ will be set.

Instruction execution resumes at address $IVPR_{0:15}||IVOR34_{16:27}||0b0000$.

5.2.6 Exception Priorities

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

1. EFPU Unavailable exception
2. EFPU Floating-point Data exception
3. EFPU Floating-point Round exception

An embedded Floating-point Data exception will be taken if either element generates a embedded Floating-point Data exception. An embedded Floating-point Round exception will be taken if either element generates an embedded Floating-point Round exception and neither element generates a EFPU Floating-point Data exception.

5.3 Embedded floating-point APU operations

e200z759n3 implements floating-point instructions that operate upon the contents of a 64-bit register that is a vector of two single-precision floating-point elements. The floating-point unit shares the same register file as the integer unit. There is no separate floating-point register file. Floating-point instructions are also provided to perform scalar single precision floating-point operations on the low elements of registers, without affecting the high-order portion. The Power Architecture UISA and Book E floating-point instructions are not implemented in e200z759n3.

The *Freescale EIS* architecture definition for embedded floating-point defines two operating modes; a real-time, ‘default results’ oriented mode (mode 0) and a ‘true IEEE754 results’ operating mode (mode 1). Implementations of the embedded floating-point APU may choose to implement one or both of these modes. The e200z759n3 hardware implements mode 0. IEEE754 compliant operation is still available in mode 0 with assistance of a software envelope.

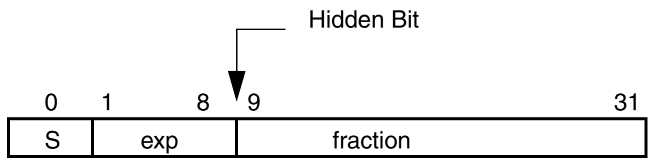
5.3.1 Floating-point data formats

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

5.3.1.1 Single-precision floating-point format

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

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



S - sign bit 0 - positive; 1 - negative

exp - biased exponent field (excess 127 notation)

fraction- fractional portion of number

Figure 5-2. Single-precision data format

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

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

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

Two specific values of the biased exponent are reserved; 0, and 255, for encoding special values of ± 0, $\pm\infty$, NaN, and Denorm .

Zeros of both positive and negative sign are represented by a biased exponent value *e* of zero and a fraction *f* that is zero.

Infinities of both positive and negative sign are represented by a biased exponent value of 255 and a fraction that is zero.

Denormalized numbers of both positive and negative sign are represented by a biased exponent value *e* of 0 and a fraction *f* that is non-zero. For these numbers, the hidden bit is defined by the IEEE-754 standard to be '0'. This number type is not directly supported in hardware. Instead, either a software exception handler is invoked, or a default value is defined, depending on the operating mode.

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

Defining *pmax* to be the most positive normalized value (farthest from zero), *pmin* the smallest positive normalized value (closest to zero), *nmax* the most negative normalized value (farthest from zero) and *nmin* the smallest normalized negative value (closest to zero), an overflow is said to have occurred if the numerically correct result of an instruction is such that r>*pmax* or r<*nmax*. An underflow is said to have occurred if the numerically correct result of an instruction is such that 0<r<*pmin* or *nmin*<r<0. In this case, r may be denormalized, or may be smaller than the smallest denormalized number. If *e*=255 and *f*!= 0, then the value is a NaN. If *e*=0 and *f*=0, then the value is a signed 0.

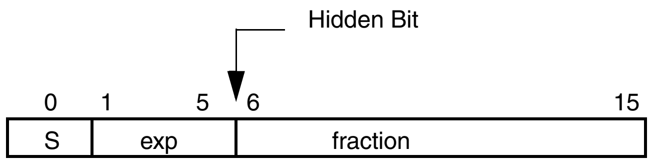
The EFPU hardware will not produce +Inf, -Inf, NaN, or a Denormalized number. If the result of an instruction overflows and Floating-point Overflow exceptions are disabled (SPEFSCR$_{\text{FOVFE}}$ bit is cleared), then *pmax* or *nmax* is generated as the result of that instruction depending upon the sign of the result. If the result of an instruction underflows and Floating-point Underflow exceptions are disabled (SPEFSCR$_{\text{FUNFE}}$ bit is cleared), then +0 or -0 is generated as the result of that instruction based upon the sign of the result.

5.3.1.2 Half-precision floating-point format

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

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

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



S - sign bit 0 - positive; 1 - negative

exp - biased exponent field (excess 15 notation)

fraction- fractional portion of number

Figure 5-3. Half-precision data format

For Normalized numbers, the biased exponent value '*e*' lies in the range of 1 to 30 corresponding to an actual exponent value E in the range –14 to +15, the hidden bit is a '1' (for normalized numbers), and the value of the number is interpreted as

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

where E is the unbiased exponent and 1.fraction is the significand consisting of a leading '1' (the hidden bit) and a fractional part (fraction field). With this format, the maximum positive normalized number (*pmax*$_{hp}$) is represented by the encoding `0x7BFF`, which is 65504, and the minimum positive normalized value (*pmin*$_{hp}$) is represented by the encoding `0x0400`, which is approximately 6.1E-5 (2^{-14}).

Two specific values of the biased exponent are reserved; 0, and 31, for encoding special values of ± 0, $\pm\infty$, NaN, and Denorm .

Zeros of both positive and negative sign are represented by a biased exponent value *e* of zero and a fraction *f* that is zero.

Infinities of both positive and negative sign are represented by a biased exponent value of 31 and a fraction that is zero.

Denormalized numbers of both positive and negative sign are represented by a biased exponent value *e* of 0 and a fraction *f* that is non-zero. For these numbers, the hidden bit is defined to be ‘0’.

Not a Numbers (*NaNs*) are represented by a biased exponent value *e* of 31 and a fraction *f* that is non-zero.

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

The EFPU hardware will not produce +Inf, –Inf, NaN, or a Denormalized number. If the result of a conversion to half-precision format overflows and Floating-point Overflow exceptions are disabled ($SPEFSCR_{FOVFE}$ bit is cleared), then $pmax_{hp}$ or $nmax_{hp}$ is generated as the result of that instruction depending upon the sign of the result. If the result of conversion to half-precision format underflows and Floating-point Underflow exceptions are disabled ($SPEFSCR_{FUNFE}$ bit is cleared), then +0 or -0 is generated as the result of that instruction based upon the sign of the result. Conversions from half-precision format to single-precision format are always exact, unless the source operand is a NaN, Inf, or Denorm. In such cases, if Floating-point Invalid Input exceptions are disabled ($SPEFSCR_{FINVE}$ bit is cleared), the conversion results in a properly signed max norm or zero default result.

5.3.2 IEEE 754 compliance

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

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

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

5.3.3 Floating-point exceptions

See Section 5.2.5, EFPU exceptions.

5.3.4 Embedded scalar single-precision floating-point instructions

In the following instruction descriptions, “sa” is the sign of operand A, “ea” is the biased exponent value of operand A, “sb” is the sign of operand B, “eb” is the biased exponent value of operand B, “ei” is an intermediate exponent value, “r” is a result value.

efsabs efsabs

Floating-Point Single-Precision Absolute Value

efsabs **r**D,**r**A

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

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

Description:

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

Exceptions:

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

efsadd

Floating-Point Single-Precision Add

efsadd rD,**r**A,**r**B

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

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

Description:

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

Exceptions:

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

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

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

efscfh   efscfh

Convert Floating-Point Single-Precision from Half-Precision

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

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

```
FP16format f;
FP32format result;

f ← $rB_{48:63}$

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

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

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

Exceptions:

If the source element of rB is Infinity, Denorm, or NaN, $SPEFSCR_{FINV}$ is set. If $SPEFSCR_{FINVE}$ is set, an interrupt is taken, the destination register is not updated, and the FGH, FXH, FG, and FX bits are cleared.

efscfsf efscfsf

Convert Floating-Point Single-Precision from Signed Fraction

efscfsf **r**D,**r**B

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

Description:

```
bl = RB$_{32:63}$
RD$_{32:63}$ = CnvtSF32ToFP32(bl)
```

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

Exceptions:

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

efscfsi

Convert Floating-Point Single-Precision from Signed Integer

efscfsi **r**D,**r**B

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

Description:

```
bl = RB$_{32:63}$
RD$_{32:63}$ = CnvtSI32ToFP32(bl)
```

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

Exceptions:

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

efscfuf efscfuf

Convert Floating-Point Single-Precision from Unsigned Fraction

efscfuf **r**D,**r**B

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

Description:

```
bl = RB32:63
RD32:63 = CnvtUF32ToFP32(bl)
```

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

Exceptions:

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

efscfui efscfui

Convert Floating-Point Single-Precision from Unsigned Integer

efscfui **r**D,**r**B

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

Description:

```
bl = $RB_{32:63}$
$RD_{32:63}$ = CnvtUI32ToFP32(bl)
```

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

Exceptions:

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

efscmpgt efscmpgt

Floating-Point Single-Precision Compare Greater Than

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

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

Description:

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

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

Exceptions:

If the contents of RA or RB are Infinity, Denorm, or NaN, the SPEFSCR$_{FINV}$ bit is set, and the FGH FXH, FG and FX bits are cleared. If Floating-point Invalid Input exceptions are enabled then an exception is taken, and the Condition Register is not updated. Otherwise, the comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of ‘*e*’ and ‘*f*’ directly.

efscmpeq efscmpeq

Floating-Point Single-Precision Compare Equal

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

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

Description:

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

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

Exceptions:

If the contents of RA or RB are Infinity, Denorm, or NaN, the $SPEFSCR_{FINV}$ bit is set, and the FGH FXH, FG and FX bits are cleared. If Floating-point Invalid Input exceptions are enabled then an exception is taken, and the Condition Register is not updated. Otherwise, the comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of ‘*e*’ and ‘*f*’ directly.

efscmplt efscmplt

Floating-Point Single-Precision Compare Less Than

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

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

Description:

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

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

Exceptions:

If the contents of RA or RB are Infinity, Denorm, or NaN, the $SPEFSCR_{FINV}$ bit is set, and the FGH FXH, FG and FX bits are cleared. If Floating-point Invalid Input exceptions are enabled then an exception is taken, and the Condition Register is not updated. Otherwise, the comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of ‘*e*’ and ‘*f*’ directly.

Convert Floating-Point Single-Precision to Half-Precision

efscth **r**D,**r**B

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

```
FP32format f;
FP16format result;

f ← rB$_{32:63}$

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

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

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

Exceptions:

If the source element of rB is Infinity, Denorm, or NaN, SPEFSCR$_{FINV}$ is set. If SPEFSCR$_{FINVE}$ is set, an interrupt is taken, the destination register is not updated, and the FGH, FXH, FG, and FX bits are cleared. Otherwise, if an overflow occurs, SPEFSCR$_{FOVF}$ is set, or if an underflow occurs, SPEFSCR$_{FUNF}$ is set. If either underflow or overflow exceptions are enabled and the corresponding bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

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

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

efsctsf efsctsf

Convert Floating-Point Single-Precision to Signed Fraction

efsctsf **r**D,**r**B

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

Description:

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

The single-precision floating-point low element in RB is converted to a signed fraction using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit fraction. NaNs are converted as though they were zero.

Exceptions:

If the contents of RB are Infinity, Denorm, or NaN, or if an overflow occurs, then the $SPEFSCR_{FINV}$ bit is set, and the FGH, FXH, FG, and FX bits are cleared. If $SPEFSCR_{FINVE}$ is set, an exception is taken, and the destination register is not updated.

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

efsctsi efsctsi

Convert Floating-Point Single-Precision to Signed Integer

efsctsi **r**D,**r**B

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

Description:

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

The single-precision floating-point low element in RB is converted to a signed integer using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Exceptions:

If the contents of RB are Infinity, Denorm, or NaN, or if an overflow occurs, then the SPEFSCR$_{FINV}$ bit is set, and the FGH, FXH, FG, and FX bits are cleared. If SPEFSCR$_{FINVE}$ is set, an exception is taken, the destination register is not updated, and no other status bits are set.

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

efsctsiz

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

efsctsiz rD,rB

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

Description:

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

The single-precision floating-point low element in RB is converted to a signed integer using the rounding mode Round toward Zero and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Exceptions:

If the contents of RB are Infinity, Denorm, or NaN, or if an overflow occurs, then the $SPEFSCR_{FINV}$ bit is set, and the FGH, FXH, FG, and FX bits are cleared. If $SPEFSCR_{FINVE}$ is set, an exception is taken, the destination register is not updated, and no other status bits are set.

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

efsctuf efsctuf

Convert Floating-Point Single-Precision to Unsigned Fraction

efsctuf **r**D,**r**B

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

Description:

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

The single-precision floating-point low element in RB is converted to an unsigned fraction using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit unsigned fraction. NaNs are converted as though they were zero.

Exceptions:

If the contents of RB are Infinity, Denorm, or NaN, or if an overflow occurs, then the SPEFSCR$_{FINV}$ bit is set, and the FGH, FXH, FG, and FX bits are cleared. If SPEFSCR$_{FINVE}$ is set, an exception is taken, and the destination register is not updated.

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

efsctui efsctui

Convert Floating-Point Single-Precision to Unsigned Integer

efsctui rD,rB

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

Description:

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

The single-precision floating-point low element in RB is converted to an unsigned integer using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Exceptions:

If the contents of RB are Infinity, Denorm, or NaN, or if an overflow occurs, then the $SPEFSCR_{FINV}$ bit is set, and the FGH, FXH, FG, and FX bits are cleared. If $SPEFSCR_{FINVE}$ is set, an exception is taken, and the destination register is not updated.

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

efsctuiz

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

efsctui **r**D,**r**B

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

Description:

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

The single-precision floating-point low element in RB is converted to an unsigned integer using the rounding mode Round toward Zero and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Exceptions:

If the contents of RB are Infinity, Denorm, or NaN, or if an overflow occurs, then the $SPEFSCR_{FINV}$ bit is set, and the FGH, FXH, FG, and FX bits are cleared. If $SPEFSCR_{FINVE}$ is set, an exception is taken, and the destination register is not updated.

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

efsdiv efsdiv

Floating-Point Single-Precision Divide

efsdiv rD,rA,rB

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

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

Description:

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

Exceptions:

If the contents of RA or RB are Infinity, Denorm, or NaN, or if both RA and RB are +/-0, the $SPEFSCR_{FINV}$ bit is set. If $SPEFSCR_{FINVE}$ is set, an exception is taken, and the destination register is not updated. Otherwise, if the content of RB is +/-0 and the content of RA is a finite normalized non-zero number, the $SPEFSCR_{FDBZ}$ bit is set. If Floating-point Divide by Zero exceptions are enabled, an exception is then taken. Otherwise, if an overflow occurs, then the $SPEFSCR_{FOVF}$ bit is set, or if an underflow occurs, then the $SPEFSCR_{FUNF}$ bit is set. If either underflow or overflow exceptions are enabled and the corresponding bit is set, an exception is taken. If any of these exceptions are taken, the destination register is not updated.

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

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

efsmadd

Floating-Point Single-Precision Multiply-Add

efsmadd **r**D,**r**A,**r**B

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

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

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

Exceptions:

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

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

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

efsmax

Floating-Point Single-Precision Maximum

efsmax **rD,rA,rB**

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

```
al ← rA$_{32:63}$
bl ← rB$_{32:63}$
if (al < bl) then temp ← bl
else temp ← al
if (isnan(al) & ~(isnan(bl))) then temp ← bl
if (isnan(bl) & ~(isnan(al))) then temp ← al
rD$_{32:63}$ ← temp
```

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

Exceptions:

If the contents of rA or rB are Infinity, Denorm, or NaN, SPEFSCR$_{FINV}$ is set, and the FGH, FXH, FG and FX bits are cleared. If SPEFSCR$_{FINVE}$ is set, an interrupt is taken, and the destination register is not updated. Otherwise, the comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of ‘*e*’ and ‘*f*’ directly. If one of the elements is a NaN and the other is not, the non-NaN element is selected rather than the comparison result. If the selected element is denorm, the result is a same signed zero. If the selected element is +NaN or +infinity, the corresponding result is *pmax*. Otherwise, if the selected element is -NaN or -infinity, the corresponding result is *nmax*.

efsmin

Floating-Point Single-Precision Minimum

efsmin **rD,rA,rB**

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

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

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

Exceptions:

If the contents of rA or rB are Infinity, Denorm, or NaN, $SPEFSCR_{FINV}$ is set, and the FGH, FXH, FG and FX bits are cleared. If $SPEFSCR_{FINVE}$ is set, an interrupt is taken, and the destination register is not updated. Otherwise, the comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of '*e*' and '*f*' directly. If one of the elements is a NaN and the other is not, the non-NaN element is selected rather than the comparison result. If the selected element is denorm, the result is a same signed zero. If the selected element is +NaN or +infinity, the corresponding result is *pmax*. Otherwise, if the selected element is -NaN or -infinity, the corresponding result is *nmax*.

efsmsub

Floating-Point Single-Precision Multiply-Subtract

efsmsub rD**,r**A**,r**B

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

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

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

Exceptions:

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

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

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

efsmul efsmul

Floating-Point Single-Precision Multiply

efsmul **r**D,**r**A,**r**B

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

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

Description:

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

Exceptions:

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

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

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

efsnabs efsnabs

Floating-Point Single-Precision Negative Absolute Value

efsnabs **r**D,**r**A

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

```
RD$_{32:63}$ = 0b1 || RA$_{33:63}$
```

Description:

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

Exceptions:

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

efsneg efsneg

Floating-Point Single-Precision Negate

efsneg **r**D,**r**A

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

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

Description:

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

Exceptions:

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

efsnmadd

Floating-Point Single-Precision Negative Multiply-Add

efsnmadd rD,**r**A,**r**B

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

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

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

Exceptions:

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

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

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

efsnmsub

Floating-Point Single-Precision Negative Multiply-Subtract

efsnmsub **r**D,**r**A,**r**B

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

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

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

Exceptions:

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

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

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

efssqrt efssqrt

Floating-Point Single-Precision Square Root

efssqrt **rD,rA**

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

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

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

Exceptions:

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

If the result element of this instruction is inexact, or underflows but underflow exceptions are disabled, and no other interrupt is taken, $SPEFSCR_{FINXS}$ is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler, and the FGH and FXH bits are cleared.

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

efssub efssub

Floating-Point Single-Precision Subtract

efssub rD**,r**A**,r**B

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

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

Description:

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

Exceptions:

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

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

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

efststeq efststeq

Floating-Point Single-Precision Test Equal

efststeq **crf**D,**r**A,**r**B

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

Description:

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

The low element of RA is compared against the low element of RB. If RA is equal to RB, then the bit in the crfD is set, otherwise it is cleared. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of ‘*e*’ and ‘*f*’ directly.

No exceptions are generated during the execution of **efststeq** instruction. If strict IEEE 754 compliance is required, then the program should use the **efscmpeq** instruction.

Implementation note: In an implementation, the execution of **efststeq** is likely to be faster than the execution of **efscmpeq** instruction.

efststgt efststgt

Floating-Point Single-Precision Test Greater Than

efststgt **crf**D,**r**A,**r**B

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

Description:

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

The low element of RA is compared against the low element of RB. If RA is greater than RB, then the bit in the crfD is set, otherwise it is cleared. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of ‘*e*’ and ‘*f*’ directly.

No exceptions are generated during the execution of **efststgt** instruction. If strict IEEE 754 compliance is required, then the program should use the **efscmpgt** instruction.

Implementation note: In an implementation, the execution of **efststgt** is likely to be faster than the execution of **efscmpgt** instruction.

efststlt efststlt

Floating-Point Single-Precision Test Less Than

efststlt **crf**D,**r**A,**r**B

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

Description:

```
al = RA$_{32:63}$
bl = RB$_{32:63}$
if (al < bl) then cl = 1
else cl = 0
CR$_{4*crfD:4*crfD+3}$ = undefined || cl || undefined || undefined
```

The low element of RA is compared against the low element of RB. If RA is less than RB, then the bit in the crfD is set, otherwise it is cleared. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of ‘*e*’ and ‘*f*’ directly.

No exceptions are generated during the execution of **efststlt** instruction. If strict IEEE 754 compliance is required, then the program should use the **efscmplt** instruction.

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

5.3.5 EFPU Vector Single-precision Embedded Floating-Point Instructions

In the following instruction descriptions, “sa” is the sign of operand A, “ea” is the biased exponent value of operand A, “sb” is the sign of operand B, “eb” is the biased exponent value of operand B, “ei” is an intermediate exponent value, “r” is a result value.

evfsabs evfsabs

Vector Floating-Point Single-Precision Absolute Value

evfsabs **r**D,**r**A

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

```
$RD_{0:31}$ = 0b0 || $RA_{1:31}$
$RD_{32:63}$ = 0b0 || $RA_{33:63}$
```

Description:

The sign bit of each element in RA is set to 0 and the results are placed into RD.

Exceptions:

If the contents of either element of RA are Infinity, Denorm, or NaN, the SPEFSCR$_{\text{FINV, FINVH}}$ bits are set appropriately, and the SPEFSCR$_{\text{FGH, FXH, FG, FX}}$ bits are cleared appropriately. If Floating-point Invalid Input exceptions are enabled then an exception is taken, and the destination register is not updated.

evfsadd evfsadd

Vector Floating-Point Single-Precision Add

evfsadd rD**,r**A**,r**B

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

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

Description:

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

Exceptions:

If the contents of either element of RA or RB are Infinity, Denorm, or NaN, the $SPEFSCR_{FINV, FINVH}$ bits are set appropriately, and the $SPEFSCR_{FGH, FXH, FG, FX}$ bits are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an exception is taken and the destination register is not updated. Otherwise, if an overflow occurs, then the $SPEFSCR_{FOVF, FOVFH}$ bits are set appropriately, or if an underflow occurs, then the $SPEFSCR_{FUNF, FUNFH}$ bits are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an exception is taken. If any of these exceptions are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other exception is taken, or underflows but underflow exceptions are disabled, and no other exception is taken, the $SPEFSCR_{FINXS}$ bit will be set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the exception handler.

FG and FX (FGH and FXH) will be cleared if an overflow or underflow exception is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfsaddsub evfsaddsub

Vector Floating-Point Single-Precision Add / Subtract

evfsaddsub rD,rA,rB

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

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

The high order single-precision floating-point element of rA is added to the corresponding element of rB, the low order single-precision floating-point element of rB is subtracted from the corresponding element of rA, and the results are stored in rD. If an element of rA is NaN or infinity, the corresponding result is either *pmax* (`sa==0`)or *nmax* (`sa==1`). Otherwise, if an element of rB is NaN or infinity, the corresponding result is either *pmax* (`sb==0`) or *nmax* (`sb==1`). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of rD. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in the corresponding element of rD.

Exceptions:

If the contents of either element of rA or rB are Infinity, Denorm, or NaN, $SPEFSCR_{FINV,FINVH}$ are set appropriately, and $SPEFSCR_{FGH,FXH,FG,FX}$ are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an interrupt is taken and the destination register is not updated. Otherwise, if an overflow occurs, $SPEFSCR_{FOVF,FOVFH}$ are set appropriately, or if an underflow occurs, $SPEFSCR_{FUNF,FUNFH}$ are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, $SPEFSCR_{FINXS,FINXSH}$ is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfsaddsubx evfsaddsubx

Vector Floating-Point Single-Precision Add / Subtract Exchanged

evfsaddsubx **rD,rA,rB**

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

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

The high-order single-precision floating-point element of rB is added to the low-order element of rA, the low-order single-precision floating-point element of rB is subtracted from the high-order element of rA, and the results are stored in rD. If an element of rA is NaN or infinity, the corresponding result is either *pmax* (`sa==0`)or *nmax* (`sa==1`). Otherwise, if an element of rB is NaN or infinity, the corresponding result is either *pmax* (`sb==0`) or *nmax* (`sb==1`). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of rD. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in the corresponding element of rD.

Exceptions:

If the contents of either element of rA or rB are Infinity, Denorm, or NaN, $\text{SPEFSCR}_{\text{FINV,FINVH}}$ are set appropriately, and $\text{SPEFSCR}_{\text{FGH,FXH,FG,FX}}$ are cleared appropriately. If $\text{SPEFSCR}_{\text{FINVE}}$ is set, an interrupt is taken and the destination register is not updated. Otherwise, if an overflow occurs, $\text{SPEFSCR}_{\text{FOVF,FOVFH}}$ are set appropriately, or if an underflow occurs, $\text{SPEFSCR}_{\text{FUNF,FUNFH}}$ are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, $\text{SPEFSCR}_{\text{FINXS,FINXSH}}$ is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

Vector Floating-Point Single-Precision Add Exchanged

evfsaddx **rD,rA,rB**

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

```
rD0:31 ← rA32:63 +sp rB0:31
rD32:63 ← rA0:31 +sp rB32:63
```

The high-order single-precision floating-point element of rB is added to the low-order element of rA, the low-order single-precision floating-point element of rB is added to the high-order element of rA, and the results are stored in rD. If an element of rA is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if an element of rB is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of rD. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in the corresponding element of rD.

Exceptions:

If the contents of either element of rA or rB are Infinity, Denorm, or NaN, $\text{SPEFSCR}_{\text{FINV,FINVH}}$ are set appropriately, and $\text{SPEFSCR}_{\text{FGH,FXH,FG,FX}}$ are cleared appropriately. If $\text{SPEFSCR}_{\text{FINVE}}$ is set, an interrupt is taken and the destination register is not updated. Otherwise, if an overflow occurs, $\text{SPEFSCR}_{\text{FOVF,FOVFH}}$ are set appropriately, or if an underflow occurs, $\text{SPEFSCR}_{\text{FUNF,FUNFH}}$ are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, $\text{SPEFSCR}_{\text{FINXS,FINXSH}}$ is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfscfh evfscfh

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

evfscfh **r**D,**r**B

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

```
FP16format f;
FP32format result;

fh ← $rB_{24:31}$
fl ← $rB_{48:63}$

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

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

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

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

Exceptions:

If either element of RB is Infinity, Denorm, or NaN, then the $SPEFSCR_{FINV, FINVH}$ bits are set appropriately, and the $SPEFSCR_{FGH, FXH, FG, FX}$ bits are cleared. If $SPEFSCR_{FINVE}$ is set, an exception is taken, the destination register is not updated, and no other status bits are set.

evfscfsf

Vector Convert Floating-Point Single-Precision from Signed Fraction

evfscfsf **r**D,**r**B

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

Description:

```
RD0:31 = CnvtSF32ToFP32(RB0:31)
RD32:63 = CnvtSF32ToFP32(RB32:63)
```

Each signed fractional element of **r**B is converted to a single-precision floating-point value using the current rounding mode and the results are placed into the corresponding elements of **r**D.

Exceptions:

This instruction can signal an inexact status and set $\mathrm{SPEFSCR}_{\mathrm{FINXS}}$ if the conversions are not exact. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result(s). The FGH, FXH, FG and FX bits are properly updated to allow rounding to be performed in the exception handler.

evfscfsi evfscfsi

Vector Convert Floating-Point Single-Precision from Signed Integer

evfscfsi **r**D,**r**B

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

Description:

```
RD$_{0:31}$ = CnvtSI32ToFP32(RB$_{0:31}$)
RD$_{32:63}$ = CnvtSI32ToFP32(RB$_{32:63}$)
```

Each signed integer element of **r**B is converted to the nearest single-precision floating-point value using the current rounding mode and the results are placed into the corresponding element of **r**D.

Exceptions:

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

evfscfuf

Vector Convert Floating-Point Single-Precision from Unsigned Fraction

evfscfuf **r**D,**r**B

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

```
RD$_{0:31}$ = CnvtUF32ToFP32(RB$_{0:31}$)
RD$_{32:63}$ = CnvtUF32ToFP32(RB$_{32:63}$)
```

Each unsigned fractional element of **r**B is converted to a single-precision floating-point value using the current rounding mode and the results are placed into the corresponding elements of **r**D.

Exceptions:

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

evfscfui evfscfui

Vector Convert Floating-Point Single-Precision from Unsigned Integer

evfscfui **r**D,**r**B

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

Description:

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

Each unsigned integer element of **r**B is converted to the nearest single-precision floating-point value using the current rounding mode and the results are placed into the corresponding elements of **r**D.

Exceptions:

This instruction can signal an inexact status and set $SPEFSCR_{FINXS}$ if the conversions are not exact. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result(s). The FGH, FXH, FG and FX bits are properly updated to allow rounding to be performed in the exception handler.

evfscmpeq evfscmpeq

Vector Floating-Point Single-Precision Compare Equal

evfscmpeq **crf**D,**r**A,**r**B

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

Description:

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

Each element of **r**A is compared against the corresponding element of **r**B. If **r**A equals RB, the **crf**D bit is set, otherwise it is cleared. Comparison ignores the sign of 0 (+0 = -0).

Exceptions:

If the contents of either element of RA or RB are Infinity, Denorm, or NaN, the $SPEFSCR_{FINV, FINVH}$ bits are set appropriately, and the $SPEFSCR_{FGH, FXH, FG, FX}$ bits are cleared appropriately. If Floating-point Invalid Input exceptions are enabled then an exception is taken, and the Condition Register is not updated. Otherwise, the comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of ‘*e*’ and ‘*f*’ directly.

evfscmpgt evfscmpgt

Vector Floating-Point Single-Precision Compare Greater Than

evfscmpgt **crf**D,**r**A,**r**B

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

Description:

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

Each element of **r**A is compared against the corresponding element of **r**B. If **r**A is greater than **r**B, the bit in the **crf**D is set, otherwise it is cleared. Comparison ignores the sign of 0 (+0 = -0).

Exceptions:

If the contents of either element of RA or RB are Infinity, Denorm, or NaN, the $SPEFSCR_{FINV, FINVH}$ bits are set appropriately, and the $SPEFSCR_{FGH, FXH, FG, FX}$ bits are cleared appropriately. If Floating-point Invalid Input exceptions are enabled then an exception is taken, and the Condition Register is not updated. Otherwise, the comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of ‘*e*’ and ‘*f*’ directly.

evfscmplt

Vector Floating-Point Single-Precision Compare Less Than

evfscmplt **crf**D,**r**A,**r**B

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

Description:

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

Each element of **r**A is compared against the corresponding element of **r**B. If **r**A is less than **r**B, the bit in the **crf**D is set, otherwise it is cleared. Comparison ignores the sign of 0 (+0 = -0).

Exceptions:

If the contents of either element of RA or RB are Infinity, Denorm, or NaN, the $SPEFSCR_{FINV, FINVH}$ bits are set appropriately, and the $SPEFSCR_{FGH, FXH, FG, FX}$ bits are cleared appropriately. If Floating-point Invalid Input exceptions are enabled then an exception is taken, and the Condition Register is not updated. Otherwise, the comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of '*e*' and '*f*' directly.

evfscth

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

evfscth **rD,rB**

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

```
FP32format fh, fl;
FP16format resulth, resultl;

fh ← $rB_{0:31}$; fl ← $rB_{32:63}$

if ($fh_{exp}$ = 0) & ($fh_{frac}$ = 0)) then
    resulth ← $fh_{sign}$ || $^{15}0$    // signed zero value
else if Isa32NaNorInfinity(fh) then
    $SPEFSCR_{FINVH}$ ← 1
    result ← $fh_{sign}$ || 0b11110 || $^{10}1$    // max value
else if Isa32Denorm(fh) then
    $SPEFSCR_{FINVH}$ ← 1
    resulth ← $f_{sign}$ || $^{15}0$
else
    unbias ← $fh_{exp}$ - 127
    if unbias > 15 then
        resulth ← $fh_{sign}$ || 0b11110 || $^{10}1$     // max value
        $SPEFSCR_{FOVFH}$ ← 1
    else if unbias < -14 && (result would not round up to bmin) then
        resulth ← $fh_{sign}$ || $^{15}0$    // like-signed zero value
        $SPEFSCR_{FUNFH}$ ← 1
    else
        $resulth_{sign}$ ← $fh_{sign}$; $resulth_{exp}$ ← unbias + 15; $resulth_{frac}$ ← $fh_{frac[0:9]}$
        guard ← $fh_{frac[10]}$; sticky ← ($fh_{frac[11:22]}$ ≠ 0)
        result ← Round16(result, LOWER, guard, sticky)
        $SPEFSCR_{FGH}$ ← guard; $SPEFSCR_{FXH}$ ← sticky
        if guard | sticky then $SPEFSCR_{FINXS}$ ← 1

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

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

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

Exceptions:

If the contents of either element of rB is Infinity, Denorm, or NaN, $\text{SPEFSCR}_{\text{FINV,FINVH}}$ are set appropriately, and $\text{SPEFSCR}_{\text{FGH,FXH,FG,FX}}$ are cleared appropriately. If $\text{SPEFSCR}_{\text{FINVE}}$ is set, an interrupt is taken and the destination register is not updated. Otherwise, if an overflow occurs, $\text{SPEFSCR}_{\text{FOVF,FOVFH}}$ are set appropriately, or if an underflow occurs, $\text{SPEFSCR}_{\text{FUNF,FUNFH}}$ are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, $\text{SPEFSCR}_{\text{FINXS,FINXSH}}$ is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result(s). The FGH, FXH, FG, and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfsctsf evfsctsf

Vector Convert Floating-Point Single-Precision to Signed Fraction

evfsctsf **r**D,**r**B

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

Description:

```
ah = $RB_{0:31}$
if (ah == Denorm) then
    $RD_{0:31}$ = 0
else if ((al == +0) || (al == -0)) // zero cases
    $RD_{0:31}$ = 0
else if (eah < 127) then
    $RD_{0:31}$ = CnvtFP32ToSF32Sat(ah)
else if ((eah == 127) && (sah == 1) && (fah==0)) then
    $RD_{0:31}$ = 0x80000000 // max negative, no overflow
else if (ah == NAN) then $RD_{0:31}$ = 0
else // Overflow
    if (sah == 0) then // Positive
        $RD_{0:31}$ = 0x7FFFFFFF
    else
        $RD_{0:31}$ = 0x80000000


al = $RB_{32:63}$
if (al == Denorm) then
    $RD_{32:63}$ = 0
else if ((al == +0) || (al == -0)) // zero cases
    $RD_{32:63}$ = 0
else if (eal < 127) then
    $RD_{32:63}$ = CnvtFP32ToSF32Sat(al)
else if ((eal == 127) && (sal == 1) && (fal==0)) then
    $RD_{32:63}$ = 0x80000000 // max negative, no overflow
else if (al == NAN) then $RD_{32:63}$ = 0
else // Overflow
    if (sal == 0) then // Positive
        $RD_{32:63}$ = 0x7FFFFFFF
    else
        $RD_{32:63}$ = 0x80000000
```

Each single-precision floating-point element in RB is converted to a signed fraction using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit signed fraction. NaNs are converted as though they were zero.

Exceptions:

If either element of RB is Infinity, Denorm, or NaN, or if an overflow occurs, then the $SPEFSCR_{FINV, FINVH}$ bits are set appropriately, and the $SPEFSCR_{FGH, FXH, FG, FX}$ bits are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an exception is taken, the destination register is not updated, and no other status bits are set.

If either result element of this instruction is inexact and no other exception is taken, the $SPEFSCR_{FINXS}$ bit will be set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated

result. The FGH, FXH, FG and FX bits are properly updated to allow rounding to be performed in the exception handler.

evfsctsi   evfsctsi

Vector Convert Floating-Point Single-Precision to Signed Integer

evfsctsi **r**D,**r**B

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

Description:

```
ah = $RB_{0:31}$
if (ah == Denorm) then
    $RD_{0:31}$ = 0
else if (eah < 158) then
    $RD_{0:31}$ = CnvtFP32ToSI32Sat(ah)
else if ((eah == 158) && (sah == 1) && (fah==0)) then
    $RD_{0:31}$ = 0x80000000 // max negative, no overflow
else if (ah == NAN) then $RD_{0:31}$ = 0
else // Overflow
    if (sah == 0) then // Positive
        $RD_{0:31}$ = 0x7FFFFFFF
    else
        $RD_{0:31}$ = 0x80000000

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

Each single-precision floating-point element in RB is converted to a signed integer using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Exceptions:

If the contents of either element of RB are Infinity, Denorm, or NaN, or if an overflow occurs on conversion, then the $SPEFSCR_{FINV, FINVH}$ bits are set appropriately, and the $SPEFSCR_{FGH, FXH, FG, FX}$ bits are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an exception is taken, the destination register is not updated, and no other status bits are set.

If either result element of this instruction is inexact and no other exception is taken, the $SPEFSCR_{FINXS}$ bit will be set. If the Floating-point Inexact exception is enabled, an exception is taken using the

Floating-point Round exception vector. In this case, the destination register is updated with the truncated result. The FGH, FXH, FG and FX bits are properly updated to allow rounding to be performed in the exception handler.

evfsctsiz evfsctsiz

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

evfsctsiz rD,rB

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

Description:

```
ah = $RB_{0:31}$
if (ah == Denorm) then
    $RD_{0:31}$ = 0
else if (eah < 158) then
    $RD_{0:31}$ = CnvtFP32ToSI32Sat(ah)
else if ((eah == 158) && (sah == 1) && (fah==0)) then
    $RD_{0:31}$ = 0x80000000 // max negative, no overflow
else if (ah == NAN) then $RD_{0:31}$ = 0
else // Overflow
    if (sah == 0) then // Positive
        $RD_{0:31}$ = 0x7FFFFFFF
    else
        $RD_{0:31}$ = 0x80000000

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

Each single-precision floating-point element in RB is converted to a signed integer using the rounding mode Round toward Zero and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Exceptions:

If either element of RB is Infinity, Denorm, or NaN, or if an overflow occurs, then the SPEFSCR$_{\text{FINV, FINVH}}$ bits are set appropriately, and the SPEFSCR$_{\text{FGH, FXH, FG, FX}}$ bits are cleared appropriately. If SPEFSCR$_{\text{FINVE}}$ is set, an exception is taken, the destination register is not updated, and no other status bits are set.

If either result element of this instruction is inexact and no other exception is taken, the SPEFSCR$_{\text{FINXS}}$ bit will be set. If the Floating-point Inexact exception is enabled, an exception is taken using the

Floating-point Round exception vector. In this case, the destination register is updated with the truncated result. The FGH, FXH, FG and FX bits are properly updated to allow rounding to be performed in the exception handler.

evfsctuf

Vector Convert Floating-Point Single-Precision to Unsigned Fraction

evfsctuf **r**D,**r**B

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

Description:

```
ah = RB$_{0:31}$
if (ah == Denorm) then // force denorm to zero
    RD$_{0:31}$ = 0
else if ((ah == +0) || (ah == -0)) // zero cases
    RD$_{0:31}$ = 0
else if (sah == 1) // Negative
    RD$_{0:31}$ = 0
else if (eah < 127)
    RD$_{0:31}$ = CnvtFP32ToUF32Sat(ah)
else if (ah == NAN) then RD$_{0:31}$ = 0
else // Overflow
    RD$_{0:31}$ = 0xFFFFFFFF

al = RB$_{32:63}$
if (al == Denorm) then
    RD$_{32:63}$ = 0
else if ((al == +0) || (al == -0)) // zero cases
    RD$_{32:63}$ = 0
else if (sal == 1) // Negative
    RD$_{32:63}$ = 0
else if (eal < 127)
    RD$_{32:63}$ = CnvtFP32ToUF32Sat(al)
else if (al == NAN) then RD$_{32:63}$ = 0
else // Overflow
    RD$_{32:63}$ = 0xFFFFFFFF
```

Each single-precision floating-point element in RB is converted to an unsigned fraction using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit fraction. NaNs are converted as though they were zero.

Exceptions:

If either element of RB is Infinity, Denorm, or NaN, or if an overflow occurs, then the SPEFSCR$_{\text{FINV, FINVH}}$ bits are set appropriately, and the SPEFSCR$_{\text{FGH, FXH, FG, FX}}$ bits are cleared appropriately. If SPEFSCR$_{\text{FINVE}}$ is set, an exception is taken, the destination register is not updated, and no other status bits are set.

If either result element of this instruction is inexact and no other exception is taken, the SPEFSCR$_{\text{FINXS}}$ bit will be set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated

result. The FGH, FXH, FG and FX bits are properly updated to allow rounding to be performed in the exception handler.

evfsctui

Vector Convert Floating-Point Single-Precision to Unsigned Integer

evfsctui **r**D,**r**B

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

Description:

```
ah = $RB_{0:31}$
if (ah == Denorm) then // force denorm to zero
    $RD_{0:31}$ = 0
else if ((ah == +0) || (ah == -0)) // zero cases
    $RD_{0:31}$ = 0
else if (sah == 1) // Negative
    $RD_{0:31}$ = 0
else if (eah <= 158)
    $RD_{0:31}$ = CnvtFP32ToUI32Sat(ah)
else if (ah == NAN) then $RD_{0:31}$ = 0
else // Overflow
    $RD_{0:31}$ = 0xFFFFFFFF

al = $RB_{32:63}$
if (al == Denorm) then
    $RD_{32:63}$ = 0
else if ((al == +0) || (al == -0)) // zero cases
    $RD_{32:63}$ = 0
else if (sal == 1) // Negative
    $RD_{32:63}$ = 0
else if (eal <= 158)
    $RD_{32:63}$ = CnvtFP32ToUI32Sat(al)
else if (al == NAN) then $RD_{32:63}$ = 0
else // Overflow
    $RD_{32:63}$ = 0xFFFFFFFF
```

Each single-precision floating-point element in RB is converted to an unsigned integer using the current rounding mode and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Exceptions:

If either element of RB is Infinity, Denorm, or NaN, or if an overflow occurs, then the SPEFSCR$_{FINV, FINVH}$ bits are set appropriately, and the SPEFSCR$_{FGH, FXH, FG, FX}$ bits are cleared appropriately. If SPEFSCR$_{FINVE}$ is set, an exception is taken, the destination register is not updated, and no other status bits are set.

If either result element of this instruction is inexact and no other exception is taken, the SPEFSCR$_{FINXS}$ bit will be set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated

result. The FGH, FXH, FG and FX bits are properly updated to allow rounding to be performed in the exception handler.

evfsctuiz evfsctuiz

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

evfsctui **r**D,**r**B

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

Description:

```
ah = RB$_{0:31}$
if (ah == Denorm) then // force denorm to zero
    RD$_{0:31}$ = 0
else if ((ah == +0) || (ah == -0)) // zero cases
    RD$_{0:31}$ = 0
else if (sah == 1) // Negative
    RD$_{0:31}$ = 0
else if (eah <= 158)
    RD$_{0:31}$ = CnvtFP32ToUI32Sat(ah)
else if (ah == NAN) then RD$_{0:31}$ = 0
else // Overflow
    RD$_{0:31}$ = 0xFFFFFFFF

al = RB$_{32:63}$
if (al == Denorm) then
    RD$_{32:63}$ = 0
else if ((al == +0) || (al == -0)) // zero cases
    RD$_{32:63}$ = 0
else if (sal == 1) // Negative
    RD$_{32:63}$ = 0
else if (eal <= 158)
    RD$_{32:63}$ = CnvtFP32ToUI32Sat(al)
else if (al == NAN) then RD$_{32:63}$ = 0
else // Overflow
    RD$_{32:63}$ = 0xFFFFFFFF
```

Each single-precision floating-point element in RB is converted to an unsigned integer using the rounding mode Round toward Zero and the result is saturated if it cannot be represented in a 32-bit integer. NaNs are converted as though they were zero.

Exceptions:

If either element of RB is Infinity, Denorm, or NaN, or if an overflow occurs, then the SPEFSCR$_{FINV, FINVH}$ bits are set appropriately, and the SPEFSCR$_{FGH, FXH, FG, FX}$ bits are cleared appropriately. If SPEFSCR$_{FINVE}$ is set, an exception is taken, the destination register is not updated, and no other status bits are set.

If either result element of this instruction is inexact and no other exception is taken, the SPEFSCR$_{FINXS}$ bit will be set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated

result. The FGH, FXH, FG and FX bits are properly updated to allow rounding to be performed in the exception handler.

evfsdiff evfsdiff

Vector Floating-Point Single-Precision Differences

evfsdiff **rD,rA,rB**

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

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

The low-order single-precision floating-point element of rA is subtracted from the high-order element of rA, the low-order single-precision floating-point element of rB is subtracted from the high-order element of rB, and the results are stored in rD. If the high-order element of rA or rB is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if the low order element of rA or rB is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of rD. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in the corresponding element of rD.

Exceptions:

If the contents of either element of rA or rB are Infinity, Denorm, or NaN, $SPEFSCR_{FINV,FINVH}$ are set appropriately, and $SPEFSCR_{FGH,FXH,FG,FX}$ are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an interrupt is taken and the destination register is not updated. Otherwise, if an overflow occurs, $SPEFSCR_{FOVF,FOVFH}$ are set appropriately, or if an underflow occurs, $SPEFSCR_{FUNF,FUNFH}$ are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, $SPEFSCR_{FINXS,FINXSH}$ is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfsdiffsum

Vector Floating-Point Single-Precision Difference / Sum

evfsdiffsum **rD,rA,rB**

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

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

The low-order single-precision floating-point element of rA is subtracted from the high-order element of rA, the low-order single-precision floating-point element of rB is added to the high-order element of rB, and the results are stored in rD. If the high-order element of rA or rB is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if the low order element of rA or rB is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of rD. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in the corresponding element of rD.

Exceptions:

If the contents of either element of rA or rB are Infinity, Denorm, or NaN, $SPEFSCR_{FINV,FINVH}$ are set appropriately, and $SPEFSCR_{FGH,FXH,FG,FX}$ are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an interrupt is taken and the destination register is not updated. Otherwise, if an overflow occurs, $SPEFSCR_{FOVF,FOVFH}$ are set appropriately, or if an underflow occurs, $SPEFSCR_{FUNF,FUNFH}$ are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, $SPEFSCR_{FINXS,FINXSH}$ is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfsdiv evfsdiv

Vector Floating-Point Single-Precision Divide

evfsdiv rD,**r**A,**r**B

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

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

Each single-precision floating-point element of **r**A is divided by the corresponding element of **r**B and the result is stored in **r**D. If RB is a NaN or infinity, the result is a properly signed zero. Otherwise, if RB is a denormalized number or a zero, or if RA is either NaN or infinity, the result is either *pmax* (`sa==sb`), or *nmax* (`sa!=sb`). Otherwise, if an overflow occurs, then *pmax* or *nmax* (as appropriate) is stored in RD. If an underflow occurs, then +0 or -0 (as appropriate) is stored in RD.

Exceptions:

If the contents of RA or RB are Infinity, Denorm, or NaN, or if both RA and RB are +/-0, the SPEFSCR$_{FINV, FINVH}$ bits are set appropriately, and the SPEFSCR$_{FGH, FXH, FG, FX}$ bits are cleared appropriately. If SPEFSCR$_{FINVE}$ is set, an exception is taken and the destination register is not updated. Otherwise, if the content of RB is +/-0 and the content of RA is a finite normalized non-zero number, the SPEFSCR$_{FDBZ, FDBZH}$ bits are set appropriately. If Floating-point Divide by Zero exceptions are enabled, an exception is then taken. Otherwise, if an overflow occurs, then the SPEFSCR$_{FOVF, FOVFH}$ bits are set appropriately, or if an underflow occurs, then the SPEFSCR$_{FUNF, FUNFH}$ bits are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding bit is set, an exception is taken. If any of these exceptions are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other exception is taken, or underflows but underflow exceptions are disabled, and no other exception is taken, the SPEFSCR$_{FINXS}$ bit will be set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the exception handler.

FG and FX (FGH and FXH) will be cleared if an overflow or underflow exception is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfsmadd   evfsmadd

Vector Floating-Point Single-Precision Multiply-Add

evfsmadd **r**D,**r**A,**r**B

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

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

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

Exceptions:

If the contents of either element of RA, RB, or RD are Infinity, Denorm, or NaN, the $SPEFSCR_{FINV, FINVH}$ bits are set appropriately, and the $SPEFSCR_{FGH, FXH, FG, FX}$ bits are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an exception is taken and the destination register is not updated. Otherwise, if an overflow occurs, then the $SPEFSCR_{FOVF, FOVFH}$ bits are set appropriately, or if an underflow occurs, then the $SPEFSCR_{FUNF, FUNFH}$ bits are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an exception is taken. If any of these exceptions are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other exception is taken, or underflows but underflow exceptions are disabled, and no other exception is taken, the $SPEFSCR_{FINXS}$ bit will be set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the exception handler.

FG and FX (FGH and FXH) will be cleared if an overflow or underflow exception is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfsmax

Vector Floating-Point Single-Precision Maximum

evfsmax **rD,rA,rB**

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

```
ah ← rA$_{0:31}$
bh ← rB$_{0:31}$
if (ah < bh) then temph ← bh
else temph ← ah
if (isnan(ah) & ~(isnan(bh))) then temph ← bh
if (isnan(bh) & ~(isnan(ah))) then temph ← ah
rD$_{0:31}$ ← temph

al ← rA$_{32:63}$
bl ← rB$_{32:63}$
if (al < bl) then templ ← bl
else templ ← al
if (isnan(al) & ~(isnan(bl))) then templ ← bl
if (isnan(bl) & ~(isnan(al))) then templ ← al
rD$_{32:63}$ ← templ
```

Each single-precision floating-point element of rA is compared against the corresponding elements of rB. The larger element is selected and placed into the corresponding element of rD. The maximum of +0 and -0 is +0.

Exceptions:

If the contents of either element of rA or rB are Infinity, Denorm, or NaN, SPEFSCR$_{FINV,FINVH}$ are set appropriately, and SPEFSCR$_{FGH,FXH,FG,FX}$ are cleared appropriately. If SPEFSCR$_{FINVE}$ is set, an interrupt is taken, and the destination register is not updated. Otherwise, the comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of '*e*' and '*f*' directly. If one of the elements is a NaN and the other is not, the non-NaN element is selected rather than the comparison result. If the selected element is denorm, the result is a same signed zero. If the selected element is +NaN or +infinity, the corresponding result is *pmax*. Otherwise, if the selected element is -NaN or -infinity, the corresponding result is *nmax*.

evfsmin evfsmin

Vector Floating-Point Single-Precision Minimum

evfsmin rD,rA,rB

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

```
ah ← $rA_{0:31}$
bh ← $rB_{0:31}$
if (ah < bh) then temph ← ah
else temph ← bh
if (isnan(ah) & ~(isnan(bh))) then temph ← bh
if (isnan(bh) & ~(isnan(ah))) then temph ← ah
$rD_{0:31}$ ← temph

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

Each single-precision floating-point element of rA is compared against the corresponding elements of rB. The smaller element is selected and placed into the corresponding element of rD. The minimum of +0 and -0 is -0.

Exceptions:

If the contents of either element of rA or rB are Infinity, Denorm, or NaN, SPEFSCR$_{FINV,FINVH}$ are set appropriately, and SPEFSCR$_{FGH,FXH,FG,FX}$ are cleared appropriately. If SPEFSCR$_{FINVE}$ is set, an interrupt is taken, and the destination register is not updated. Otherwise, the comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of ‘*e*’ and ‘*f*’ directly. If one of the elements is a NaN and the other is not, the non-NaN element is selected rather than the comparison result. If the selected element is denorm, the result is a same signed zero. If the selected element is +NaN or +infinity, the corresponding result is *pmax*. Otherwise, if the selected element is -NaN or -infinity, the corresponding result is *nmax*.

evfsmsub

Vector Floating-Point Single-Precision Multiply-Subtract

evfsmsub rD,**r**A,**r**B

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

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

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

Exceptions:

If the contents of either element of RA, RB, or RD are Infinity, Denorm, or NaN, the $SPEFSCR_{FINV, FINVH}$ bits are set appropriately, and the $SPEFSCR_{FGH, FXH, FG, FX}$ bits are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an exception is taken and the destination register is not updated. Otherwise, if an overflow occurs, then the $SPEFSCR_{FOVF, FOVFH}$ bits are set appropriately, or if an underflow occurs, then the $SPEFSCR_{FUNF, FUNFH}$ bits are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an exception is taken. If any of these exceptions are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other exception is taken, or underflows but underflow exceptions are disabled, and no other exception is taken, the $SPEFSCR_{FINXS}$ bit will be set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the exception handler.

FG and FX (FGH and FXH) will be cleared if an overflow or underflow exception is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfsmul

Vector Floating-Point Single-Precision Multiply

evfsmul rD,**r**A,**r**B

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

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

Each single-precision floating-point element of **r**A is multiplied with the corresponding element of **r**B and the result is stored in **r**D. If RA or RB are either zero or denormalized, the result is a properly signed zero. Otherwise, if RA or RB are either NaN or infinity, the result is either *pmax* (`sa==sb`), or *nmax* (`sa!=sb`). Otherwise, if an overflow occurs, then *pmax* or *nmax* (as appropriate) is stored in RD. If an underflow occurs, then +0 or -0 (as appropriate) is stored in RD.

Exceptions:

If the contents of either element of RA or RB are Infinity, Denorm, or NaN, the $SPEFSCR_{FINV, FINVH}$ bits are set appropriately, and the $SPEFSCR_{FGH, FXH, FG, FX}$ bits are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an exception is taken and the destination register is not updated. Otherwise, if an overflow occurs, then the $SPEFSCR_{FOVF, FOVFH}$ bits are set appropriately, or if an underflow occurs, then the $SPEFSCR_{FUNF, FUNFH}$ bits are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an exception is taken. If any of these exceptions are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other exception is taken, or underflows but underflow exceptions are disabled, and no other exception is taken, the $SPEFSCR_{FINXS}$ bit will be set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the exception handler.

FG and FX (FGH and FXH) will be cleared if an overflow or underflow exception is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfsmule evfsmule

Vector Floating-Point Single-Precision Multiply By Even Element

evfsmule rD,rA,rB

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

```
rD0:31  ← rA0:31 ×sp rB0:31
rD32:63 ← rA0:31 ×sp rB32:63
```

The single-precision floating-point elements of rB are multiplied by the even (high-order) element of rA, and the results are stored in rD. If an element of rB or the even element of rA is either zero denormalized, the corresponding result is a properly signed zero. Otherwise, if an element of rB or the even element of rA is either NaN or infinity, the corresponding result is either *pmax* ($a_{sign}==b_{sign}$), or *nmax* ($a_{sign}!=b_{sign}$). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of rD. If an underflow occurs, +0 or -0 (as appropriate) is stored in the corresponding element of rD.

Exceptions:

If the contents of either element of rB or the even element of rA is Infinity, Denorm, or NaN, $SPEFSCR_{FINV,FINVH}$ are set appropriately, and $SPEFSCR_{FGH,FXH,FG,FX}$ are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an interrupt is taken and the destination register is not updated. Otherwise, if an overflow occurs, $SPEFSCR_{FOVF,FOVFH}$ are set appropriately, or if an underflow occurs, $SPEFSCR_{FUNF,FUNFH}$ are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, $SPEFSCR_{FINXS}$ is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow exception is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfsmulo evfsmulo

Vector Floating-Point Single-Precision Multiply By Odd Element

evfsmulo rD,rA,rB

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

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

The single-precision floating-point elements of rB are multiplied by the odd (low-order) element of rA, and the results are stored in rD. If an element of rB or the odd element of rA is either zero or denormalized, the corresponding result is a properly signed zero. Otherwise, if an element of rB or the odd element of rA is either NaN or infinity, the corresponding result is either *pmax* (a_{sign}==b_{sign}), or *nmax* (a_{sign}!=b_{sign}). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of rD. If an underflow occurs, +0 or -0 (as appropriate) is stored in the corresponding element of rD.

Exceptions:

If the contents of either element of rB or the odd element of rA is Infinity, Denorm, or NaN, $SPEFSCR_{FINV,FINVH}$ are set appropriately, and $SPEFSCR_{FGH,FXH,FG,FX}$ are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an interrupt is taken and the destination register is not updated. Otherwise, if an overflow occurs, $SPEFSCR_{FOVF,FOVFH}$ are set appropriately, or if an underflow occurs, $SPEFSCR_{FUNF,FUNFH}$ are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, $SPEFSCR_{FINXS}$ is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow exception is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfsmulx evfsmulx

Vector Floating-Point Single-Precision Multiply Exchanged

evfsmulx **rD,rA,rB**

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

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

The high-order single-precision floating-point element of rB is multiplied by the low-order element of rA, the low-order single-precision floating-point element of rB is multiplied by the high-order element of rA, and the results are stored in rD. If an element of rA or rB is either zero or denormalized, the corresponding result is a properly signed zero. Otherwise, if an element of rA or rB are either NaN or infinity, the corresponding result is either *pmax* (a_{sign}==b_{sign}), or *nmax* (a_{sign}!=b_{sign}). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of rD. If an underflow occurs, +0 or -0 (as appropriate) is stored in the corresponding element of rD.

Exceptions:

If the contents of either element of rA or rB are Infinity, Denorm, or NaN, $\text{SPEFSCR}_{\text{FINV,FINVH}}$ are set appropriately, and $\text{SPEFSCR}_{\text{FGH,FXH,FG,FX}}$ are cleared appropriately. If $\text{SPEFSCR}_{\text{FINVE}}$ is set, an interrupt is taken and the destination register is not updated. Otherwise, if an overflow occurs, $\text{SPEFSCR}_{\text{FOVF,FOVFH}}$ are set appropriately, or if an underflow occurs, $\text{SPEFSCR}_{\text{FUNF,FUNFH}}$ are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, $\text{SPEFSCR}_{\text{FINXS}}$ is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow exception is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfsnabs evfsnabs

Vector Floating-Point Single-Precision Negative Absolute Value

evfsnabs **r**D,**r**A

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

```
$RD_{0:31}$ = 0b1 || $RA_{1:31}$
$RD_{32:63}$ = 0b1 || $RA_{33:63}$
```

Description:

The sign bit of each element in RA is set to 1 and the results are placed into RD.

Exceptions:

If the contents of either element of RA are Infinity, Denorm, or NaN, the $\text{SPEFSCR}_{\text{FINV, FINVH}}$ bits are set appropriately, and the $\text{SPEFSCR}_{\text{FGH, FXH, FG, FX}}$ bits are cleared appropriately. If Floating-point Invalid Input exceptions are enabled then an exception is taken, and the destination register is not updated.

Vector Floating-Point Single-Precision Negate

evfsneg rD,rA

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

```
$RD_{0:31} = \neg RA_{0} \,||\, RA_{1:31}$
$RD_{32:63} = \neg RA_{32} \,||\, RA_{33:63}$
```

Description:

The sign bit of each element in RA is complemented and the results are placed into RD.

Exceptions:

If the contents of either element of RA are Infinity, Denorm, or NaN, the $SPEFSCR_{FINV, FINVH}$ bits are set appropriately, and the $SPEFSCR_{FGH, FXH, FG, FX}$ bits are cleared appropriately. If Floating-point Invalid Input exceptions are enabled then an exception is taken, and the destination register is not updated.

evfsnmadd

Vector Floating-Point Single-Precision Negative Multiply-Add

evfsnmadd rD,**r**A,**r**B

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

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

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

Exceptions:

If the contents of either element of RA, RB, or RD are Infinity, Denorm, or NaN, the $SPEFSCR_{FINV, FINVH}$ bits are set appropriately, and the $SPEFSCR_{FGH, FXH, FG, FX}$ bits are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an exception is taken and the destination register is not updated. Otherwise, if an overflow occurs, then the $SPEFSCR_{FOVF, FOVFH}$ bits are set appropriately, or if an underflow occurs, then the $SPEFSCR_{FUNF, FUNFH}$ bits are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an exception is taken. If any of these exceptions are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other exception is taken, or underflows but underflow exceptions are disabled, and no other exception is taken, the $SPEFSCR_{FINXS}$ bit will be set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the exception handler.

FG and FX (FGH and FXH) will be cleared if an overflow or underflow exception is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfsnmsub evfsnmsub

Vector Floating-Point Single-Precision Negative Multiply-Subtract

evfsnmsub rD,**r**A,**r**B

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

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

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

Exceptions:

If the contents of either element of RA, RB, or RD are Infinity, Denorm, or NaN, the $SPEFSCR_{FINV, FINVH}$ bits are set appropriately, and the $SPEFSCR_{FGH, FXH, FG, FX}$ bits are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an exception is taken and the destination register is not updated. Otherwise, if an overflow occurs, then the $SPEFSCR_{FOVF, FOVFH}$ bits are set appropriately, or if an underflow occurs, then the $SPEFSCR_{FUNF, FUNFH}$ bits are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an exception is taken. If any of these exceptions are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other exception is taken, or underflows but underflow exceptions are disabled, and no other exception is taken, the $SPEFSCR_{FINXS}$ bit will be set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the exception handler.

FG and FX (FGH and FXH) will be cleared if an overflow or underflow exception is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

Vector Floating-Point Single-Precision Square Root

evfssqrt rD,rA

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

```
rD0:31  ← SQRT(rA0:31)
rD32:63 ← SQRT(rA32:63)
```

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

Exceptions:

If the contents of either element of rA are non-zero and have a negative sign, or are Infinity, Denorm, or NaN, $SPEFSCR_{FINV,FINVH}$ are set appropriately, and $SPEFSCR_{FGH,FXH,FG,FX}$ are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an interrupt is taken and the destination register is not updated. Otherwise, if an underflow occurs, $SPEFSCR_{FUNF,FUNFH}$ are set appropriately. If underflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

If either result element of this instruction is inexact, or underflows but underflow exceptions are disabled, and no other interrupt is taken, $SPEFSCR_{FINXS,FINXSH}$ is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfssub

Vector Floating-Point Single-Precision Subtract

evfssub rD**,r**A**,r**B

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

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

Description:

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

Exceptions:

If the contents of either element of RA or RB are Infinity, Denorm, or NaN, the $SPEFSCR_{FINV,\ FINVH}$ bits are set appropriately, and the $SPEFSCR_{FGH,\ FXH,\ FG,\ FX}$ bits are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an exception is taken and the destination register is not updated. Otherwise, if an overflow occurs, then the $SPEFSCR_{FOVF,\ FOVFH}$ bits are set appropriately, or if an underflow occurs, then the $SPEFSCR_{FUNF,\ FUNFH}$ bits are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an exception is taken. If any of these exceptions are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other exception is taken, or underflows but underflow exceptions are disabled, and no other exception is taken, the $SPEFSCR_{FINXS}$ bit will be set. If the Floating-point Inexact exception is enabled, an exception is taken using the Floating-point Round exception vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the exception handler.

FG and FX (FGH and FXH) will be cleared if an overflow or underflow exception is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfssubadd

Vector Floating-Point Single-Precision Subtract / Add

evfssubadd **rD,rA,rB**

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

```
rD0:31 ← rA0:31 -sp rB0:31
rD32:63 ← rA32:63 +sp rB32:63
```

The high-order single-precision floating-point element of rB is subtracted from the corresponding element of rA, the low-order single-precision floating-point element of rB is subtracted from the corresponding element of rA, and the results are stored in rD. If an element of rA is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if an element of rB is NaN or infinity, the corresponding result is either *nmax* or *pmax* (as appropriate). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of rD. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in the corresponding element of rD.

Exceptions:

If the contents of either element of rA or rB are Infinity, Denorm, or NaN, $\text{SPEFSCR}_{\text{FINV,FINVH}}$ are set appropriately, and $\text{SPEFSCR}_{\text{FGH,FXH,FG,FX}}$ are cleared appropriately. If $\text{SPEFSCR}_{\text{FINVE}}$ is set, an interrupt is taken and the destination register is not updated. Otherwise, if an overflow occurs, $\text{SPEFSCR}_{\text{FOVF,FOVFH}}$ are set appropriately, or if an underflow occurs, $\text{SPEFSCR}_{\text{FUNF,FUNFH}}$ are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, $\text{SPEFSCR}_{\text{FINXS}}$ is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfssubaddx evfssubaddx

Vector Floating-Point Single-Precision Subtract / Add Exchanged

evfssubaddx rD,rA,rB

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

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

The high-order single-precision floating-point element of rB is subtracted from the low-order element of rA, the low-order single-precision floating-point element of rB is added to the high-order from the corresponding element of rA, and the results are stored in rD. If an element of rA is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if an element of rB is NaN or infinity, the corresponding result is either *nmax* or *pmax* (as appropriate). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of rD. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in the corresponding element of rD.

Exceptions:

If the contents of either element of rA or rB are Infinity, Denorm, or NaN, $SPEFSCR_{FINV,FINVH}$ are set appropriately, and $SPEFSCR_{FGH,FXH,FG,FX}$ are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an interrupt is taken and the destination register is not updated. Otherwise, if an overflow occurs, $SPEFSCR_{FOVF,FOVFH}$ are set appropriately, or if an underflow occurs, $SPEFSCR_{FUNF,FUNFH}$ are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, $SPEFSCR_{FINXS}$ is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfssubx evfssubx

Vector Floating-Point Single-Precision Subtract Exchanged

evfssubx **rD,rA,rB**

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

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

The high-order single-precision floating-point element of rB is subtracted from the low-order element of rA, the low-order single-precision floating-point element of rB is subtracted from the high-order from the corresponding element of rA, and the results are stored in rD. If an element of rA is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if an element of rB is NaN or infinity, the corresponding result is either *nmax* or *pmax* (as appropriate). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of rD. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in the corresponding element of rD.

Exceptions:

If the contents of either element of rA or rB are Infinity, Denorm, or NaN, $SPEFSCR_{FINV,FINVH}$ are set appropriately, and $SPEFSCR_{FGH,FXH,FG,FX}$ are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an interrupt is taken and the destination register is not updated. Otherwise, if an overflow occurs, $SPEFSCR_{FOVF,FOVFH}$ are set appropriately, or if an underflow occurs, $SPEFSCR_{FUNF,FUNFH}$ are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, $SPEFSCR_{FINXS}$ is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfssum evfssum

Vector Floating-Point Single-Precision Sums

evfssum rD,rA,rB

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

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

The high-order single-precision floating-point element of rA is added to the low-order element of rA, the high-order single-precision floating-point element of rB is added to the low-order element of rB, and the results are stored in rD. If the high-order element of rA or rB is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if the low order element of rA or rB is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of rD. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or –0 (for rounding mode RM) is stored in the corresponding element of rD.

Exceptions:

If the contents of either element of rA or rB are Infinity, Denorm, or NaN, $SPEFSCR_{FINV,FINVH}$ are set appropriately, and $SPEFSCR_{FGH,FXH,FG,FX}$ are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an interrupt is taken and the destination register is not updated. Otherwise, if an overflow occurs, $SPEFSCR_{FOVF,FOVFH}$ are set appropriately, or if an underflow occurs, $SPEFSCR_{FUNF,FUNFH}$ are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, $SPEFSCR_{FINXS,FINXSH}$ is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfssumdiff evfssumdiff

Vector Floating-Point Single-Precision Sum / Difference

evfssumdiff **rD,rA,rB**

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

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

The high-order single-precision floating-point element of rA is added to the low-order element of rA, the low-order single-precision floating-point element of rB is subtracted from the high-order element of rB, and the results are stored in rD. If the high-order element of rA or rB is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if the low order element of rA or rB is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of rD. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in the corresponding element of rD.

Exceptions:

If the contents of either element of rA or rB are Infinity, Denorm, or NaN, $SPEFSCR_{FINV,FINVH}$ are set appropriately, and $SPEFSCR_{FGH,FXH,FG,FX}$ are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an interrupt is taken and the destination register is not updated. Otherwise, if an overflow occurs, $SPEFSCR_{FOVF,FOVFH}$ are set appropriately, or if an underflow occurs, $SPEFSCR_{FUNF,FUNFH}$ are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, the destination register is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, $SPEFSCR_{FINXS,FINXSH}$ is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, the destination register is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

evfststeq evfststeq

Vector Floating-Point Single-Precision Test Equal

evfststeq **crf**D,**r**A,**r**B

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

Description:

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

Each element of **r**A is compared against the corresponding element of **r**B. If **r**A equals RB, the bit in **crf**D is set, otherwise it is cleared. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of ‘*e*’ and ‘*f*’ directly.

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

Implementation note: In an implementation, the execution of **evfststeq** is likely to be faster than the execution of **evfscmpeq**.

evfststgt evfststgt

Vector Floating-Point Single-Precision Test Greater Than

evfststgt **crf**D,**r**A,**r**B

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

Description:

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

Each element of **r**A is compared against the corresponding element of **r**B. If **r**A is greater than **r**B, the bit in **crf**D is set, otherwise it is cleared. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of '*e*' and '*f*' directly.

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

Implementation note: In an implementation, the execution of **evfststgt** is likely to be faster than the execution of **evfscmpgt**.

evfststlt evfststlt

Vector Floating-Point Single-Precision Test Less Than

evfststlt **crf**D,**r**A,**r**B

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

Description:

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

Each element of **r**A is compared with the corresponding element of **r**B. If **r**A is less than **r**B, the bit in the **crf**D is set, otherwise it is cleared. Comparison ignores the sign of 0 (+0 = -0). The comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of ‘*e*’ and ‘*f*’ directly.

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

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

5.4 Embedded floating-point results summary

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

Table 5-2. Floating-point results summary — add, sub, mul, div

Operation	Operand A	Operand B	Result	FINV	FOVF	FUNF	FDBZ	FINX
Add								
Add	∞	∞	amax	1	0	0	0	0
Add	∞	NaN	amax	1	0	0	0	0
Add	∞	denorm	amax	1	0	0	0	0
Add	∞	zero	amax	1	0	0	0	0
Add	∞	Norm	amax	1	0	0	0	0
Add	NaN	∞	amax	1	0	0	0	0
Add	NaN	NaN	amax	1	0	0	0	0

Table 5-2. Floating-point results summary — add, sub, mul, div (continued)

Operation	Operand A	Operand B	Result	FINV	FOVF	FUNF	FDBZ	FINX
Add	NaN	denorm	amax	1	0	0	0	0
Add	NaN	zero	amax	1	0	0	0	0
Add	NaN	norm	amax	1	0	0	0	0
Add	denorm	∞	bmax	1	0	0	0	0
Add	denorm	NaN	bmax	1	0	0	0	0
Add	denorm	denorm	zero[1]	1	0	0	0	0
Add	denorm	zero	zero[1]	1	0	0	0	0
Add	denorm	norm	operand_b	1	0	0	0	0
Add	zero	∞	bmax	1	0	0	0	0
Add	zero	NaN	bmax	1	0	0	0	0
Add	zero	denorm	zero[1]	1	0	0	0	0
Add	zero	zero	zero[1]	0	0	0	0	0
Add	zero	norm	operand_b	0	0	0	0	0
Add	norm	∞	bmax	1	0	0	0	0
Add	norm	NaN	bmax	1	0	0	0	0
Add	norm	denorm	operand_a	1	0	0	0	0
Add	norm	zero	operand_a	0	0	0	0	0
Add	norm	norm	_Calc_	0	*	*	0	*
Subtract								
Sub	∞	∞	amax	1	0	0	0	0
Sub	∞	NaN	amax	1	0	0	0	0
Sub	∞	denorm	amax	1	0	0	0	0
Sub	∞	zero	amax	1	0	0	0	0
Sub	∞	Norm	amax	1	0	0	0	0
Sub	NaN	∞	amax	1	0	0	0	0
Sub	NaN	NaN	amax	1	0	0	0	0
Sub	NaN	denorm	amax	1	0	0	0	0
Sub	NaN	zero	amax	1	0	0	0	0
Sub	NaN	norm	amax	1	0	0	0	0
Sub	denorm	∞	-bmax	1	0	0	0	0
Sub	denorm	NaN	-bmax	1	0	0	0	0
Sub	denorm	denorm	zero[2]	1	0	0	0	0

Table 5-2. Floating-point results summary — add, sub, mul, div (continued)

Operation	Operand A	Operand B	Result	FINV	FOVF	FUNF	FDBZ	FINX
Sub	denorm	zero	zero[2]	1	0	0	0	0
Sub	denorm	norm	-operand_b	1	0	0	0	0
Sub	zero	∞	-bmax	1	0	0	0	0
Sub	zero	NaN	-bmax	1	0	0	0	0
Sub	zero	denorm	zero[2]	1	0	0	0	0
Sub	zero	zero	zero[2]	0	0	0	0	0
Sub	zero	norm	-operand_b	0	0	0	0	0
Sub	norm	∞	-bmax	1	0	0	0	0
Sub	norm	NaN	-bmax	1	0	0	0	0
Sub	norm	denorm	operand_a	1	0	0	0	0
Sub	norm	zero	operand_a	0	0	0	0	0
Sub	norm	norm	_Calc_	0	*	*	0	*
Multiply[3]								
Mul	∞	∞	max	1	0	0	0	0
Mul	∞	NaN	max	1	0	0	0	0
Mul	∞	denorm	zero	1	0	0	0	0
Mul	∞	zero	zero	1	0	0	0	0
Mul	∞	Norm	max	1	0	0	0	0
Mul	NaN	∞	max	1	0	0	0	0
Mul	NaN	NaN	max	1	0	0	0	0
Mul	NaN	denorm	zero	1	0	0	0	0
Mul	NaN	zero	zero	1	0	0	0	0
Mul	NaN	norm	max	1	0	0	0	0
Mul	denorm	∞	zero	1	0	0	0	0
Mul	denorm	NaN	zero	1	0	0	0	0
Mul	denorm	denorm	zero	1	0	0	0	0
Mul	denorm	zero	zero	1	0	0	0	0
Mul	denorm	norm	zero	1	0	0	0	0
Mul	zero	∞	zero	1	0	0	0	0
Mul	zero	NaN	zero	1	0	0	0	0
Mul	zero	denorm	zero	1	0	0	0	0
Mul	zero	zero	zero	0	0	0	0	0

Table 5-2. Floating-point results summary — add, sub, mul, div (continued)

Operation	Operand A	Operand B	Result	FINV	FOVF	FUNF	FDBZ	FINX
Mul	zero	norm	zero	0	0	0	0	0
Mul	norm	∞	max	1	0	0	0	0
Mul	norm	NaN	max	1	0	0	0	0
Mul	norm	denorm	zero	1	0	0	0	0
Mul	norm	zero	zero	0	0	0	0	0
Mul	norm	norm	_Calc_	0	*	*	0	*
Divide[3]								
Div	∞	∞	zero	1	0	0	0	0
Div	∞	NaN	zero	1	0	0	0	0
Div	∞	denorm	max	1	0	0	0	0
Div	∞	zero	max	1	0	0	0	0
Div	∞	Norm	max	1	0	0	0	0
Div	NaN	∞	zero	1	0	0	0	0
Div	NaN	NaN	zero	1	0	0	0	0
Div	NaN	denorm	max	1	0	0	0	0
Div	NaN	zero	max	1	0	0	0	0
Div	NaN	norm	max	1	0	0	0	0
Div	denorm	∞	zero	1	0	0	0	0
Div	denorm	NaN	zero	1	0	0	0	0
Div	denorm	denorm	max	1	0	0	0	0
Div	denorm	zero	max	1	0	0	0	0
Div	denorm	norm	zero	1	0	0	0	0
Div	zero	∞	zero	1	0	0	0	0
Div	zero	NaN	zero	1	0	0	0	0
Div	zero	denorm	max	1	0	0	0	0
Div	zero	zero	max	1	0	0	0	0
Div	zero	norm	zero	0	0	0	0	0
Div	norm	∞	zero	1	0	0	0	0
Div	norm	NaN	zero	1	0	0	0	0
Div	norm	denorm	max	1	0	0	0	0
Div	norm	zero	max	0	0	0	1	0
Div	norm	norm	_Calc_	0	*	*	0	*

Table 5-2. Floating-point results summary — add, sub, mul, div (continued)

Operation	Operand A	Operand B	Result	FINV	FOVF	FUNF	FDBZ	FINX

Notes: the following definitions apply

[1] - sign of result is positive when sign_a and sign_b are different for all rounding modes except round to minus infinity, where it is negative.

[2] - sign of result is positive when sign_a and sign_b are the same for all rounding modes except round to minus infinity, where it is negative.

[3] - sign of result is always (sign_a XOR sign_b)

* - updated according to results of calculation

Calc - result is updated with the results of calculation

max - max normalized number with sign of (sign_a XOR sign_b)

amax - max normalized number with sign of sign_a

bmax - max normalized number with sign of sign_b

nmax - max negative normalized number

pmax - max positive normalized number

Table 5-3. Floating-point results summary — madd, msub, nmadd, nmsub

Operation	Operand A	Operand B	Operand D	Result	FINV	FOVF	FUNF	FDBZ	FINX
				madd					
madd	∞ , NaN	∞ , NaN, Norm	∞ , NaN, denorm, zero, Norm	abmax	1	0	0	0	0
madd	∞ , NaN	denorm, zero	∞ , NaN	dmax	1	0	0	0	0
madd	∞ , NaN	denorm, zero	denorm, zero	zero[1]	1	0	0	0	0
madd	∞ , NaN	denorm, zero	Norm	operand_d	1	0	0	0	0
madd	denorm	∞ , NaN, denorm, zero, Norm	∞ , NaN	dmax	1	0	0	0	0
madd	denorm	∞ , NaN, denorm, zero, Norm	denorm, zero	zero[1]	1	0	0	0	0
madd	denorm	∞ , NaN, denorm, zero, Norm	Norm	operand_d	1	0	0	0	0
madd	zero	∞ , NaN, denorm,	∞ , NaN	dmax	1	0	0	0	0
madd	zero	∞ , NaN, denorm	denorm, zero	zero[1]	1	0	0	0	0
madd	zero	∞ , NaN, denorm	Norm	operand_d	1	0	0	0	0
madd	zero	zero, Norm	∞ , NaN	dmax	1	0	0	0	0
madd	zero	zero, Norm	denorm	zero[1]	1	0	0	0	0
madd	zero	zero, Norm	zero	zero[1]	0	0	0	0	0
madd	zero	zero, Norm	Norm	operand_d	0	0	0	0	0
madd	norm	∞ , NaN	∞ , NaN, denorm, zero, Norm	abmax	1	0	0	0	0
madd	norm	denorm	∞ , NaN	dmax	1	0	0	0	0

Table 5-3. Floating-point results summary — madd, msub, nmadd, nmsub (continued)

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

Table 5-3. Floating-point results summary — madd, msub, nmadd, nmsub (continued)

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

Table 5-3. Floating-point results summary — madd, msub, nmadd, nmsub (continued)

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

Table 5-3. Floating-point results summary — madd, msub, nmadd, nmsub (continued)

Operation	Operand A	Operand B	Operand D	Result	FINV	FOVF	FUNF	FDBZ	FINX
nmsub	norm	norm	∞ , NaN	dmax	1	0	0	0	0
nmsub	norm	norm	denorm	-ab_Calc	1	*	*	0	*
nmsub	norm	norm	zero	-ab_Calc	0	*	*	0	*
nmsub	norm	norm	norm	-(_Calc_)	0	*	*	0	*

Notes: the following definitions apply

[1] - sign of result is positive when (sign_a XOR sign_b) and sign_d are different for all rounding modes except round to minus infinity, where it is negative.

[2] - sign of result is positive when (sign_a XOR sign_b) and sign_d are the same for all rounding modes except round to minus infinity, where it is negative.

[3]- sign of result is negative when (sign_a XOR sign_b) and sign_d are different for all rounding modes except round to minus infinity, where it is positive.

[4] - sign of result is negative when (sign_a XOR sign_b) and sign_d are the same for all rounding modes except round to minus infinity, where it is positive.

* - updated according to results of calculation

ab_Calc - result is updated with the results of intermediate product calculation, rounded

Calc - result is updated with the results of calculation, rounded

abmax - max normalized number with sign of (sign_a XOR sign_b)

dmax - max normalized number with sign of sign_d

nmax - max negative normalized number

pmax - max positive normalized number

Table 5-4. Floating-point results summary—sqrt

Operand A	Result	FINV	FOVF	FUNF	FDBZ	FINX
+∞	pmax	1	0	0	0	0
-∞	-0	1	0	0	0	0
+NaN	pmax	1	0	0	0	0
-NaN	-0	1	0	0	0	0
+denorm	+zero	1	0	0	0	0
-denorm	-zero	1	0	0	0	0
+zero	+zero	0	0	0	0	0
-zero	-zero	0	0	0	0	0
+norm	_Calc_	0	*	*	0	*
-norm	-0	1	0	0	0	0

Table 5-5. Floating-point results summary—min, max

Operand A	Operand B	Result	FINV	FOVF	FUNF	FDBZ	FINX
Max							
+∞	+∞	pmax	1	0	0	0	0
+∞	–∞	pmax	1	0	0	0	0
+∞	+NaN	pmax	1	0	0	0	0
+∞	-NaN	pmax	1	0	0	0	0
+∞	denorm	pmax	1	0	0	0	0
+∞	zero	pmax	1	0	0	0	0
+∞	Norm	pmax	1	0	0	0	0
–∞	+∞	pmax	1	0	0	0	0
–∞	–∞	nmax	1	0	0	0	0
–∞	+NaN	nmax	1	0	0	0	0
–∞	-NaN	nmax	1	0	0	0	0
–∞	denorm	bzero	1	0	0	0	0
–∞	zero	bzero	1	0	0	0	0
–∞	Norm	operand_b	1	0	0	0	0
+NaN	+∞	pmax	1	0	0	0	0
+NaN	–∞	nmax	1	0	0	0	0
+NaN	+NaN	pmax	1	0	0	0	0
+NaN	-NaN	pmax	1	0	0	0	0
+NaN	denorm	bzero	1	0	0	0	0
+NaN	zero	bzero	1	0	0	0	0
+NaN	Norm	operand_b	1	0	0	0	0
-NaN	+∞	pmax	1	0	0	0	0
-NaN	–∞	nmax	1	0	0	0	0
-NaN	+NaN	pmax	1	0	0	0	0
-NaN	-NaN	nmax	1	0	0	0	0
-NaN	denorm	bzero	1	0	0	0	0
-NaN	zero	bzero	1	0	0	0	0
-NaN	Norm	operand_b	1	0	0	0	0
+denorm	+∞	pmax	1	0	0	0	0
+denorm	–∞	azero	1	0	0	0	0
+denorm	+NaN	azero	1	0	0	0	0

Table 5-5. Floating-point results summary—min, max (continued)

Operand A	Operand B	Result	FINV	FOVF	FUNF	FDBZ	FINX
+denorm	-NaN	azero	1	0	0	0	0
+denorm	denorm	azero	1	0	0	0	0
+denorm	zero	azero	1	0	0	0	0
+denorm	+Norm	operand_b	1	0	0	0	0
+denorm	-Norm	azero	1	0	0	0	0
-denorm	+∞	pmax	1	0	0	0	0
-denorm	–∞	azero	1	0	0	0	0
-denorm	+NaN	azero	1	0	0	0	0
-denorm	-NaN	azero	1	0	0	0	0
-denorm	denorm	bzero	1	0	0	0	0
-denorm	zero	bzero	1	0	0	0	0
-denorm	+Norm	operand_b	1	0	0	0	0
-denorm	-Norm	azero	1	0	0	0	0
+zero	+∞	pmax	1	0	0	0	0
+zero	–∞	azero	1	0	0	0	0
+zero	+NaN	azero	1	0	0	0	0
+zero	-NaN	azero	1	0	0	0	0
+zero	denorm	azero	1	0	0	0	0
+zero	zero	azero	0	0	0	0	0
+zero	+Norm	operand_b	0	0	0	0	0
+zero	-Norm	azero	0	0	0	0	0
-zero	+∞	pmax	1	0	0	0	0
-zero	–∞	azero	1	0	0	0	0
-zero	+NaN	azero	1	0	0	0	0
-zero	-NaN	azero	1	0	0	0	0
-zero	denorm	bzero	1	0	0	0	0
-zero	zero	bzero	0	0	0	0	0
-zero	+Norm	operand_b	0	0	0	0	0
-zero	-Norm	azero	0	0	0	0	0
+Norm	+∞	pmax	1	0	0	0	0
+Norm	–∞	operand_a	1	0	0	0	0
+Norm	+NaN	operand_a	1	0	0	0	0

Table 5-5. Floating-point results summary—min, max (continued)

Operand A	Operand B	Result	FINV	FOVF	FUNF	FDBZ	FINX
+Norm	-NaN	operand_a	1	0	0	0	0
+Norm	denorm	operand_a	1	0	0	0	0
+Norm	zero	operand_a	0	0	0	0	0
+Norm	Norm	_Calc_	0	0	0	0	0
-Norm	+∞	pmax	1	0	0	0	0
-Norm	–∞	operand_a	1	0	0	0	0
-Norm	+NaN	operand_a	1	0	0	0	0
-Norm	-NaN	operand_a	1	0	0	0	0
-Norm	denorm	bzero	1	0	0	0	0
-Norm	zero	bzero	0	0	0	0	0
-Norm	Norm	_Calc_	0	0	0	0	0
Min							
+∞	+∞	pmax	1	0	0	0	0
+∞	–∞	nmax	1	0	0	0	0
+∞	+NaN	pmax	1	0	0	0	0
+∞	-NaN	pmax	1	0	0	0	0
+∞	denorm	bzero	1	0	0	0	0
+∞	zero	bzero	1	0	0	0	0
+∞	Norm	operand_b	1	0	0	0	0
–∞	+∞	nmax	1	0	0	0	0
–∞	–∞	nmax	1	0	0	0	0
–∞	+NaN	nmax	1	0	0	0	0
–∞	-NaN	nmax	1	0	0	0	0
–∞	denorm	nmax	1	0	0	0	0
–∞	zero	nmax	1	0	0	0	0
–∞	Norm	nmax	1	0	0	0	0
+NaN	+∞	pmax	1	0	0	0	0
+NaN	–∞	nmax	1	0	0	0	0
+NaN	+NaN	pmax	1	0	0	0	0
+NaN	-NaN	nmax	1	0	0	0	0
+NaN	denorm	bzero	1	0	0	0	0
+NaN	zero	bzero	1	0	0	0	0

Table 5-5. Floating-point results summary—min, max (continued)

Operand A	Operand B	Result	FINV	FOVF	FUNF	FDBZ	FINX
+NaN	Norm	operand_b	1	0	0	0	0
-NaN	+∞	pmax	1	0	0	0	0
-NaN	–∞	nmax	1	0	0	0	0
-NaN	+NaN	nmax	1	0	0	0	0
-NaN	-NaN	nmax	1	0	0	0	0
-NaN	denorm	bzero	1	0	0	0	0
-NaN	zero	bzero	1	0	0	0	0
-NaN	Norm	operand_b	1	0	0	0	0
+denorm	+∞	azero	1	0	0	0	0
+denorm	–∞	nmax	1	0	0	0	0
+denorm	+NaN	azero	1	0	0	0	0
+denorm	-NaN	azero	1	0	0	0	0
+denorm	denorm	bzero	1	0	0	0	0
+denorm	zero	bzero	1	0	0	0	0
+denorm	+Norm	azero	1	0	0	0	0
+denorm	-Norm	operand_b	1	0	0	0	0
-denorm	+∞	azero	1	0	0	0	0
-denorm	–∞	nmax	1	0	0	0	0
-denorm	+NaN	azero	1	0	0	0	0
-denorm	-NaN	azero	1	0	0	0	0
-denorm	denorm	azero	1	0	0	0	0
-denorm	zero	azero	1	0	0	0	0
-denorm	+Norm	azero	1	0	0	0	0
-denorm	-Norm	operand_b	1	0	0	0	0
+zero	+∞	azero	1	0	0	0	0
+zero	–∞	nmax	1	0	0	0	0
+zero	+NaN	azero	1	0	0	0	0
+zero	-NaN	azero	1	0	0	0	0
+zero	denorm	bzero	1	0	0	0	0
+zero	zero	bzero	0	0	0	0	0
+zero	+Norm	azero	0	0	0	0	0
+zero	-Norm	operand_b	0	0	0	0	0

Table 5-5. Floating-point results summary—min, max (continued)

Operand A	Operand B	Result	FINV	FOVF	FUNF	FDBZ	FINX
-zero	+∞	azero	1	0	0	0	0
-zero	–∞	nmax	1	0	0	0	0
-zero	+NaN	azero	1	0	0	0	0
-zero	-NaN	azero	1	0	0	0	0
-zero	denorm	azero	1	0	0	0	0
-zero	zero	azero	0	0	0	0	0
-zero	+Norm	azero	0	0	0	0	0
-zero	-Norm	operand_b	0	0	0	0	0
+Norm	+∞	operand_a	1	0	0	0	0
+Norm	–∞	nmax	1	0	0	0	0
+Norm	+NaN	operand_a	1	0	0	0	0
+Norm	-NaN	operand_a	1	0	0	0	0
+Norm	denorm	bzero	1	0	0	0	0
+Norm	zero	bzero	0	0	0	0	0
+Norm	Norm	_Calc_	0	0	0	0	0
-Norm	+∞	operand_a	1	0	0	0	0
-Norm	–∞	nmax	1	0	0	0	0
-Norm	+NaN	operand_a	1	0	0	0	0
-Norm	-NaN	operand_a	1	0	0	0	0
-Norm	denorm	operand_a	1	0	0	0	0
-Norm	zero	operand_a	0	0	0	0	0
-Norm	Norm	_Calc_	0	0	0	0	0

Table 5-6. Floating-point results summary — convert to unsigned

Operand B	integer result efsctui[z]	Fractional result efsctuf	FINV	FOVF	FUNF	FDBZ	FINX
+ ∞	0xFFFF_FFFF	0xFFFF_FFFF	1	0	0	0	0
- ∞	zero	zero	1	0	0	0	0
+NaN	zero	zero	1	0	0	0	0
-NaN	zero	zero	1	0	0	0	0
denorm	zero	zero	1	0	0	0	0
zero	zero	zero	0	0	0	0	0

Table 5-6. Floating-point results summary — convert to unsigned (continued)

Operand B	integer result efsctui[z]	Fractional result efsctuf	FINV	FOVF	FUNF	FDBZ	FINX
+norm	_Calc_	_Calc_	*	0	0	0	*
-norm	zero	zero	0	0	0	0	0

Table 5-7. Floating-point results summary —convert to signed

Operand B	integer result efsctsi[z]	Fractional result efsctsf	FINV	FOVF	FUNF	FDBZ	FINX
+ ∞	0x7FFF_FFFF	0x7FFF_FFFF	1	0	0	0	0
- ∞	0x8000_0000	0x8000_0000	1	0	0	0	0
+NaN	zero	zero	1	0	0	0	0
-NaN	zero	zero	1	0	0	0	0
denorm	zero	zero	1	0	0	0	0
zero	zero	zero	0	0	0	0	0
+norm	_Calc_	_Calc_	*	0	0	0	*
-norm	_Calc_	_Calc_	*	0	0	0	*

Table 5-8. Floating-point results summary — convert from unsigned

Operand B	integer source efscfui	Fractional source efscfuf	FINV	FOVF	FUNF	FDBZ	FINX
zero	zero	zero	0	0	0	0	0
norm	_Calc_	_Calc_	0	0	0	0	*

Table 5-9. Floating-point results summary — convert from signed

Operand B	integer source efscfsi	Fractional source efscfsf	FINV	FOVF	FUNF	FDBZ	FINX
zero	zero	zero	0	0	0	0	0
norm	_Calc_	_Calc_	0	0	0	0	*

Table 5-10. Floating-point results summary — fabs, fnabs, fneg

Operand A	fabs	fnabs	fneg	FINV	FOVF	FUNF	FDBZ	FINX
∞	+ ∞	- ∞	-A	1	0	0	0	0
NaN	Sign bit cleared	Sign bit set	-A	1	0	0	0	0
denorm	Sign bit cleared	Sign bit set	-A	1	0	0	0	0

Table 5-10. Floating-point results summary — fabs, fnabs, fneg

Operand A	fabs	fnabs	fneg	FINV	FOVF	FUNF	FDBZ	FINX
zero	zero	zero	zero	0	0	0	0	0
norm	norm	norm	norm	0	0	0	0	0

Table 5-11. Floating-point results summary — convert from half-precision

Operand B	e[v]fscfh	FINV	FOVF	FUNF	FDBZ	FINX
∞	bmax	1	0	0	0	0
NaN	bmax	1	0	0	0	0
denorm	bzero	1	0	0	0	0
zero	bzero	0	0	0	0	0
+norm	_Calc_	0	0	0	0	*
-norm	_Calc_	0	0	0	0	*

Table 5-12. Floating-point results summary — convert to half-precision

Operand B	e[v]fscth	FINV	FOVF	FUNF	FDBZ	FINX
∞	bmax_{hp}	1	0	0	0	0
NaN	bmax_{hp}	1	0	0	0	0
denorm	bzero	1	0	0	0	0
zero	bzero	0	0	0	0	0
+norm	_Calc_	0	*	*	0	*
-norm	_Calc_	0	*	*	0	*

5.5 EFPU instruction timing

Instruction timing in number of processor clock cycles for EFPU instructions are shown in Table 5-13, and Table 5-14. Pipelined instructions are shown with cycles of total latency and throughput cycles. Divide instructions are not pipelined and block other instructions from executing during divide execution.

Instruction pipelining in the CPU is affected by the possibility of a floating-point instruction generating an exception. A load or store class instruction that follows an EFPU instruction will stall until it can be ensured that no previous instruction can generate a floating-point exception. This determination is based on which floating-point exception enable bits are set (FINVE, FOVFE, FUNFE, FDBZE, and FINXE) and at what point in the FPU pipeline an exception can be guaranteed to not occur. Invalid input operands are detected in the first stage of the pipeline, while underflow, overflow, and inexactness are determined later in the pipeline. Best overall performance occurs when either floating-point exceptions are disabled, or when load and store class instructions are scheduled such that previous floating-point instructions have already resolved the possibility of exceptional results.

5.5.1 EFPU single-precision vector floating-point instruction timing

Instruction timing for EFPU vector floating-point instructions is shown in Table 5-13. The table is sorted by opcode. The number of stall cycles for **evfsdiv** and **evfssqrt** is (latency) cycles.

Table 5-13. EFPU vector floating-point instruction timing

Instruction	Latency	Throughput	Comments
evfsabs	4	1	—
evfsadd	4	1	—
evfsaddx	4	1	—
evfsaddsub	4	1	—
evfsaddsubx	4	1	—
evfscfh	4	1	—
evfscfsf	4	1	—
evfscfsi	4	1	—
evfscfuf	4	1	—
evfscfui	4	1	—
evfscmpeq	4	1	—
evfscmpgt	4	1	—
evfscmplt	4	1	—
evfscth	4	1	—
evfsctsf	4	1	—
evfsctsi	4	1	—
evfsctsiz	4	1	—
evfsctuf	4	1	—
evfsctui	4	1	—
evfsctuiz	4	1	—
evfsdiff	4	1	—
evfsdiffsum	4	1	—
evfsdiv	13	13	blocking, no overlap with next inst.
evfsmax	4	1	—
evfsmin	4	1	—
evfsmadd	4	1[1]	dest also used as source
evfsmsub	4	1[1]	dest also used as source
evfsmul	4	1	—
evfsmule	4	1	—

Table 5-13. EFPU vector floating-point instruction timing (continued)

Instruction	Latency	Throughput	Comments
evfsmulo	4	1	—
evfsmulx	4	1	—
evfsnabs	4	1	—
evfsneg	4	1	—
evfsnmadd	4	1[1]	dest also used as source
evfsnmsub	4	1[1]	dest also used as source
evfssqrt	15	15	blocking, no overlap with next inst.
evfssub	4	1	—
evfssubx	4	1	—
evfssubadd	4	1	—
evfssubaddx	4	1	—
evfssum	4	1	—
evfssumdiff	4	1	—
evfststeq	4	1	—
evfststgt	4	1	—
evfststlt	4	1	—

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

5.5.2 EFPU single-precision scalar floating-point instruction timing

Instruction timing for EFPU single-precision scalar floating-point instructions is shown in Table 5-14. The table is sorted by opcode.

Table 5-14. EFPU single-precision scalar floating-point instruction timing

Instruction	Latency	Throughput	Comments
efsabs	4	1	—
efsadd	4	1	—
efscfh	4	1	—
efscfsf	4	1	—
efscfsi	4	1	—
efscfuf	4	1	—
efscfui	4	1	—
efscmpeq	4	1	—

Table 5-14. EFPU single-precision scalar floating-point instruction timing (continued)

Instruction	Latency	Throughput	Comments
efscmpgt	4	1	—
efscmplt	4	1	—
efscth	4	1	—
efsctsf	4	1	—
efsctsi	4	1	—
efsctsiz	4	1	—
efsctuf	4	1	—
efsctui	4	1	—
efsctuiz	4	1	—
efsdiv	13	13	blocking, no execution overlap with next instruction
efsmadd	4	1[1]	dest also used as source
efsmsub	4	1[1]	dest also used as source
efsmax	4	1	—
efsmin	4	1	—
efsmul	4	1	—
efsnabs	4	1	—
efsneg	4	1	—
efsnmadd	4	1[1]	dest also used as source
efsnmsub	4	1[1]	dest also used as source
efssqrt	15	15	blocking, no overlap with next inst.
efssub	4	1	—
efststeq	4	1	—
efststgt	4	1	—
efststlt	4	1	—

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

5.6 Instruction forms and opcodes

Table 5-15 gives the division of the opcode space for the EFPU instructions. This is the architectural assignment; not all instructions are implemented in all versions of the CPU.

Table 5-15. Opcode space division

Opcode bits		Instruction class
0–5	21–28	
4	0101 00xx	Embedded vector floating-point instructions
4	0101 010x	Embedded vector floating-point instructions
4	0101 0110	Embedded scalar floating-point single-precision instructions
4	0101 0111	Reserved (Embedded scalar floating-point double-precision instructions)[1]
4	0101 10xx	Embedded scalar floating-point single-precision instructions
4	0101 11xx	Reserved (Embedded scalar floating-point double-precision instructions)[1]

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

5.6.1 Opcodes for EFPU vector floating-point instructions

Table 5-16. Embedded vector floating-point instruction opcodes

Instruction	Opcode Bits						Comments
	0–5	6–10	11–15	16–20	21–24	25–31	
evfsadd	4	rD	rA	rB	0101	0000000	—
evfssub	4	rD	rA	rB	0101	0000001	rA - rB
evfsmadd	4	rD	rA	rB	0101	0000010	—
evfsmsub	4	rD	rA	rB	0101	0000011	—
evfsabs	4	rD	rA	00000	0101	0000100	—
evfsnabs	4	rD	rA	00000	0101	0000101	—
evfsneg	4	rD	rA	00000	0101	0000110	—
evfssqrt	4	rD	rA	00000	0101	0000111	—
evfsmul	4	rD	rA	rB	0101	0001000	—
evfsdiv	4	rD	rA	rB	0101	0001001	—
evfsnmadd	4	rD	rA	rB	0101	0001010	—
evfsnmsub	4	rD	rA	rB	0101	0001011	—
evfscmpgt	4	crfD 00	rA	rB	0101	0001100	—
evfscmplt	4	crfD 00	rA	rB	0101	0001101	—
evfscmpeq	4	crfD 00	rA	rB	0101	0001110	—
	4				0101	0001111	—
evfscfui	4	rD	00000	rB	0101	0010000	—
evfscfsi	4	rD	00000	rB	0101	0010001	—

Table 5-16. Embedded vector floating-point instruction opcodes (continued)

Instruction	Opcode Bits						Comments
	0–5	6–10	11–15	16–20	21–24	25–31	
evfscfh	4	rD	00100	rB	0101	0010001	—
evfscfuf	4	rD	00000	rB	0101	0010010	—
evfscfsf	4	rD	00000	rB	0101	0010011	—
evfsctui	4	rD	00000	rB	0101	0010100	—
evfsctsi	4	rD	00000	rB	0101	0010101	—
evfscth	4	rD	00100	rB	0101	0010101	—
evfsctuf	4	rD	00000	rB	0101	0010110	—
evfsctsf	4	rD	00000	rB	0101	0010111	—
evfsctuiz	4	rD	00000	rB	0101	0011000	—
	4				0101	0011001	—
evfsctsiz	4	rD	00000	rB	0101	0011010	—
	4				0101	0011011	—
evfststgt	4	crfD 00	rA	rB	0101	0011100	—
evfststlt	4	crfD 00	rA	rB	0101	0011101	—
evfststeq	4	crfD 00	rA	rB	0101	0011110	—
	4				0101	0011111	—
evfsmax	4	rD	rA	rB	0101	0100000	—
evfsmin	4	rD	rA	rB	0101	0100001	—
evfsaddsub	4	rD	rA	rB	0101	0100010	—
evfssubadd	4	rD	rA	rB	0101	0100011	rA - rB; rA + rB
evfssum	4	rD	rA	rB	0101	0100100	—
evfsdiff	4	rD	rA	rB	0101	0100101	—
evfssumdiff	4	rD	rA	rB	0101	0100110	—
evfsdiffsum	4	rD	rA	rB	0101	0100111	—
evfsaddx	4	rD	rA	rB	0101	0101000	—
evfssubx	4	rD	rA	rB	0101	0101001	—
evfsaddsubx	4	rD	rA	rB	0101	0101010	—
evfssubaddx	4	rD	rA	rB	0101	0101011	rA - rB; rA + rB
evfsmulx	4	rD	rA	rB	0101	0101100	—
	4	rD	rA	rB	0101	0101101	—

Table 5-16. Embedded vector floating-point instruction opcodes (continued)

Instruction	Opcode Bits						Comments
	0–5	6–10	11–15	16–20	21–24	25–31	
evfsmule	4	rD	rA	rB	0101	0101110	—
evfsmulo	4	rD	rA	rB	0101	0101111	—

5.6.2 Opcodes for EFPU scalar single-precision floating-point instructions

Table 5-17. Embedded scalar single-precision floating-point instruction opcodes

Instruction	Opcode Bits						Comments
	0–5	6–10	11–15	16–20	21–24	25–31	
efsmax	4	rD	rA	rB	0101	0110000	—
efsmin	4	rD	rA	rB	0101	0110001	—
efsadd	4	rD	rA	rB	0101	1000000	—
efssub	4	rD	rA	rB	0101	1000001	rA - rB
efsmadd	4	rD	rA	rB	0101	1000010	—
efsmsub	4	rD	rA	rB	0101	1000011	—
efsabs	4	rD	rA	00000	0101	1000100	—
efsnabs	4	rD	rA	00000	0101	1000101	—
efsneg	4	rD	rA	00000	0101	1000110	—
efssqrt	4	rD	rA	00000	0101	1000111	—
efsmul	4	rD	rA	rB	0101	1001000	—
efsdiv	4	rD	rA	rB	0101	1001001	—
efsnmadd	4	rD	rA	rB	0101	1001010	—
efsnmsub	4	rD	rA	rB	0101	1001011	—
efscmpgt	4	crfD 00	rA	rB	0101	1001100	—
efscmplt	4	crfD 00	rA	rB	0101	1001101	—
efscmpeq	4	crfD 00	rA	rB	0101	1001110	—
efscfd	4	rD	00000	rB	0101	1001111	optional, not implemented
efscfui	4	rD	00000	rB	0101	1010000	—
efscfsi	4	rD	00000	rB	0101	1010001	—
efscfh	4	rD	00100	rB	0101	1010001	—

Table 5-17. Embedded scalar single-precision floating-point instruction opcodes (continued)

Instruction	Opcode Bits						Comments
	0–5	6–10	11–15	16–20	21–24	25–31	
efscfuf	4	rD	00000	rB	0101	1010010	—
efscfsf	4	rD	00000	rB	0101	1010011	—
efsctui	4	rD	00000	rB	0101	1010100	—
efsctsi	4	rD	00000	rB	0101	1010101	—
efscth	4	rD	00100	rB	0101	1010101	—
efsctuf	4	rD	00000	rB	0101	1010110	—
efsctsf	4	rD	00000	rB	0101	1010111	—
efsctuiz	4	rD	00000	rB	0101	1011000	—
	4				0101	1011001	—
efsctsiz	4	rD	00000	rB	0101	1011010	—
	4				0101	1011011	—
efststgt	4	crfD 00	rA	rB	0101	1011100	—
efststlt	4	crfD 00	rA	rB	0101	1011101	—
efststeq	4	crfD 00	rA	rB	0101	1011110	—
	4				0101	1011111	—

Chapter 6
Signal Processing Extension APU (SPE APU)

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

6.1 Nomenclature and conventions

Several conventions regarding nomenclature are used in this chapter:

- Due to historical precedent, the terms SPE and SIMD are sometimes used interchangeably
- Bits 0 to 31 of a 64-bit register are referenced as field 0, upper half, or high-order element of the register. Bits 32–63 are referred to as field 1, lower half, or lower-order element of the register. Each half is an element of a GPR.
- Mnemonics for SPE APU instructions generally begin with the letters 'ev' (vector).

6.2 SPE programming model

The e200z759n3 core provides a register file with thirty-two 64-bit registers. The Power Architecture 32-bit Book E instructions operate on the lower (least significant) 32 bits of the 64-bit register. New SPE instructions are defined that view the 64-bit register as being composed of a vector of two 32-bit elements, and some of the instructions also read or write 16-bit elements. These new instructions can also be used to perform scalar operations by ignoring the results of the upper 32-bit half of the register file. Some instructions are defined that produce a 64-bit scalar result. Vector fixed-point instructions operate on a vector of two 32-bit or four 16-bit fixed-point numbers resident in the 64-bit GPRs. The SPE and Book E instructions issue from a single instruction stream.

There are no record forms of SPE instructions. Vector compare instructions store the result of the comparison into the condition register (CR). The meaning of the CR bits are now overloaded for the vector operations. Vector compare instructions specify a CR field, two source registers and the type of compare: greater than, less than, or equal. Two bits in the CR field are written with the result of the vector compare, one for each element. The remaining two bits reflect the 'and'ing and 'or'ing of the vector compare results.

A partially visible accumulator register is architected for the SPE integer and fractional multiply accumulate forms of instructions. Its usage is described in Section 6.2.2, Accumulator.

6.2.1 SPE Status and Control Register (SPEFSCR)

The e200z759n3 core implements the SPEFSCR register for status reporting and control of SPE instructions. This register is also used by the Embedded Floating-Point APUs. Status and control bits are shared for floating-point operations and SPE operations. The SPEFSCR register is implemented as special purpose register (SPR) number 512 and is read and written by the **mfspr** and **mtspr** instructions. The SPEFSCR is shown in Figure 6-1.

SOVH	OVH	FGH	FXH	FINVH	FDBZH	FUNFH	FOVFH	0		FINXS	FINVS	FDBZS	FUNFS	FOVFS	MODE	SOV	OV	FG	FX	FINV	FDBZ	FUNF	FOVF	0	FINXE	FINVE	FDBZE	FUNFE	FOVFE	FRMC	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

SPR — 512; Read/Write; Reset — 0x0

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

The SPEFSCR bits are defined in Table 6-1.

Table 6-1. SPEFCR field descriptions

Bits	Name	Description
0 (32)	SOVH	Summary Integer Overflow High The SOVH bit is set to 1 whenever an instruction sets OVH. The SOVH bit remains set until it is cleared by a **mtspr** instruction specifying the SPEFSCR register.
1 (33)	OVH	Integer Overflow High The OVH bit is set to 1 whenever an integer or fractional SPE instruction signals an overflow in the upper half of the result.
2 (34)	FGH	Embedded Floating-Point Guard bit High Defined by Embedded Floating-Point APUs.
3 (35)	FXH	Embedded Floating-Point Inexact bit High Defined by Embedded Floating-Point APUs.
4 (36)	FINVH	Embedded Floating-Point Invalid Operation / Input error High Defined by Embedded Floating-Point APUs.
5 (37)	FDBZH	Embedded Floating-Point Divide by Zero High Defined by Embedded Floating-Point APUs.
6 (38)	FUNFH	Embedded Floating-Point Underflow High Defined by Embedded Floating-Point APUs.
7 (39)	FOVFH	Embedded Floating-Point Overflow High Defined by Embedded Floating-Point APUs.
8:9 (40:41)	—	Reserved
10 (42)	FINXS	Embedded Floating-Point Inexact Sticky Flag Defined by Embedded Floating-Point APUs.
11 (43)	FINVS	Embedded Floating-Point Invalid Operation Sticky Flag Defined by Embedded Floating-Point APUs.
12 (44)	FDBZS	Embedded Floating-Point Divide by Zero Sticky Flag Defined by Embedded Floating-Point APUs.
13 (45)	FUNFS	Embedded Floating-Point Underflow Sticky Flag Defined by Embedded Floating-Point APUs.
14 (46)	FOVFS	Embedded Floating-Point Overflow Sticky Flag Defined by Embedded Floating-Point APUs.
15 (47)	MODE	Embedded Floating-Point Operating Mode Defined by Embedded Floating-Point APUs.

Table 6-1. SPEFCR field descriptions (continued)

Bits	Name	Description
16 (48)	SOV	Summary Integer Overflow The SOV bit is set to 1 whenever an instruction sets OV. The SOV bit remains set until it is cleared by a **mtspr** instruction specifying the SPEFSCR register.
17 (49)	OV	Integer Overflow The OV bit is set to 1 whenever an integer or fractional SPE instruction signals an overflow in the low element result.
18 (50)	FG	Embedded Floating-Point Guard bit (low/scalar) Defined by Embedded Floating-Point APUs.
19 (51)	FX	Embedded Floating-Point Inexact bit (low/scalar) Defined by Embedded Floating-Point APUs.
20 (52)	FINV	Embedded Floating-Point Invalid Operation / Input error (low/scalar) Defined by Embedded Floating-Point APUs.
21 (53)	FDBZ	Embedded Floating-Point Divide by Zero (low/scalar) Defined by Embedded Floating-Point APUs.
22 (54)	FUNF	Embedded Floating-Point Underflow (low/scalar) Defined by Embedded Floating-Point APUs.
23 (55)	FOVF	Embedded Floating-Point Overflow (low/scalar) Defined by Embedded Floating-Point APUs.
24 (56)	—	Reserved
25 (57)	FINXE	Embedded Floating-Point Round (Inexact) Exception Enable Defined by Embedded Floating-Point APUs.
26 (58)	FINVE	Embedded Floating-Point Invalid Operation / Input Error Exception Enable Defined by Embedded Floating-Point APUs.
27 (59)	FDBZE	Embedded Floating-Point Divide by Zero Exception Enable Defined by Embedded Floating-Point APUs.
28 (60)	FUNFE	Embedded Floating-Point Underflow Exception Enable Defined by Embedded Floating-Point APUs.
29 (61)	FOVFE	Embedded Floating-Point Overflow Exception Enable Defined by Embedded Floating-Point APUs.
30:31 (62:63)	FRMC	Embedded Floating-Point Rounding Mode Control Defined by Embedded Floating-Point APUs.

6.2.2 Accumulator

The e200z759n3 core has a 64-bit architectural accumulator register that holds the results of the SPE multiply accumulate (MAC) fixed-point instructions. The accumulator allows back-to-back execution of dependent fixed-point MAC instructions, something that is found in the inner loops of DSP code such as filters. The accumulator is partially visible to the programmer in that its results do not have to be explicitly read to use them. Instead, they are always copied into a 64-bit destination GPR specified as part of the instruction. The accumulator however, has to be explicitly cleared when starting a new MAC loop. Based upon the type of instruction, an accumulator can hold either a single 64-bit value or a vector of two 32-bit elements.

An example of a MAC instruction is **evmhossfaaw r**D**,r**A**,r**B. In this instruction, the least significant 16 bits of **r**A and **r**B are multiplied for both elements of the vector (see Figure "evmhossfaaw" on page 358), the result is shifted left one bit and added to the accumulator, and the result is possibly saturated to 32 bits in case of overflow. The final result is placed both in the accumulator and also in **r**D. Thus the result of this instruction can be used by accessing **r**D.

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

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

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

6.2.2.1 Context switch

When a context switch occurs, the OS process must explicitly save the accumulator as part of the context of the swapped-out task and then explicitly load the accumulator from the context of the new task that is being swapped in. When the old task is restarted, its accumulator must be restored before restarting the task.

6.2.3 GPRs and *PowerPC Book E* instructions

The e200z759n3 core implements the 32-bit forms of the Book E instructions. All 32-bit *PowerPC Book E* instructions operate upon the lower half of the 64-bit GPR. These instructions do not affect the upper half of a GPR.

6.2.4 SPE available bit in MSR

MSR_{SPE} is defined as the SPE available bit. If this bit is clear and software attempts to execute any of the SPE instructions other than the s **brinc** instruction (which does not affect the upper 32 bits of a GPR), the SPE APU Unavailable exception is taken. If this bit is set, software can execute any of the SPE instructions.

6.2.5 SPE exception bit in ESR

ESR_{SPE} is defined as the SPE exception bit. This bit is set whenever the processor takes an exception related to the execution of the SPE APU instructions.

6.2.6 SPE exceptions

The architecture defines the following SPE APU exceptions:

- SPE APU Unavailable exception
- SPE Vector Alignment exception — not used by e200z759n3

Interrupt vector offset registers (IVOR) IVOR32 (SPE / Embedded Floating Point Unavailable Interrupt) and IVOR5 (Alignment Interrupt), are used by the interrupt model. The SPR number for IVOR32 is 528, IVOR5 is defined by Book E. These registers are privileged.

6.2.6.1 SPE APU Unavailable exception

The SPE APU Unavailable exception is taken if MSR_{SPE} is cleared and execution of a SPE APU instruction other than the **brinc** instruction is attempted. When the SPE APU Unavailable exception occurs, the processor suppresses execution of the instruction causing the exception. The SRR0, SRR1, MSR, and ESR registers are modified as follows:

- SRR0 is set to the effective address of the instruction causing the exception.
- SRR1 is set to the contents of the MSR at the time of the exception.
- $MSR_{CE,ME,DE}$ are unchanged. All other bits are cleared.
- The ESR_{SPE} bit is set. All other ESR bits are cleared.

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

6.2.7 Exception priorities

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

1. SPE APU Unavailable exception

6.3 Integer SPE simple instructions

brinc

Bit Reversed Increment

brinc rD,rA,rB

0 – 5	6 – 10	11 – 15	16 – 20	21 – 31
4	rD	RA	RB	0 1 0 0 0 0 0 1 1 1 1

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

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

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

rB contains a bitmask that is based upon the number of points in an FFT. To access a buffer containing n byte sized data that is to be accessed with bit-reversed addressing, the mask has $\log_2$n '1's in the lsb positions and '0's in the remaining most significant position. If however, the data size is a multiple of a half word or a word, the mask is constructed so that the '1's are shifted left by $\log_2$ (size of the data) and '0's are placed in the lsb positions. Table 6-2 shows example values of masks for different data sizes and number of data.

Table 6-2. Data samples and sizes

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

NOTE

An implementation can restrict the number of bits specified in a mask. In the e200z759n3 implementation, the number of bits is 16, which allows the user to perform bit-reversed address computations for 65536 byte sized samples.

evabs evabs

Vector Absolute Value

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

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

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

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

evaddiw evaddiw

Vector Add Immediate Word

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

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

```
RD0:31 = RB0:31 + EXTZ(UIMM)                                  // Modulo sum
RD32:63 = RB32:63 + EXTZ(UIMM)                                // Modulo sum
```

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

evaddw evaddw

Vector Add Word

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

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

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

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

evand

evand

Vector AND

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

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

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

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

evandc

Vector AND with Complement

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

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

```
RD0:31 = RA0:31 & (¬RB0:31)                        // Bitwise ANDC
RD32:63 = RA32:63 & (¬RB32:63)                     // Bitwise ANDC
```

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

evcmpeq

Vector Compare Equal

evcmpeq **crf**D,**r**A,**r**B

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

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

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

evcmpgts evcmpgts

Vector Compare Greater Than Signed

evcmpgts **crf**D,**r**A,**r**B

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

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

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

Vector Compare Greater Than Unsigned

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

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

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

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

evcmplts

Vector Compare Less Than Signed

evcmplts **crf**D,**r**A,**r**B

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

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

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

evcmpltu

Vector Compare Less Than Unsigned

evcmpltu **crf**D,**r**A,**r**B

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

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

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

evcntlsw

evcntlsw

Vector Count Leading Sign Bits Word

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

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

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

evcntlzw evcntlzw

Vector Count Leading Zeros Word

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

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

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

evdivws evdivws

Vector Divide Word Signed

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

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

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

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

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

evdivwu evdivwu

Vector Divide Word Unsigned

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

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

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

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

eveqv   eveqv

Vector Equivalent

eveqv   rD,rA,rB

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

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

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

evextsb evextsb

Vector Extend Sign Byte

evextsb **r**D,**r**A

0 – 5	6 – 10	11 – 15	16 – 20	21 – 31
4	RD	RA	0000 0	010 0000 1010

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

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

evextsh evextsh

Vector Extend Sign Half Word

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

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

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

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

evmergehi

Vector Merge High

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

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

```
RD0:31 = RA0:31
RD32:63 = RB0:31
```

The high-order elements of **r**A and **r**B are merged and placed into **r**D as shown in Figure 6-2.

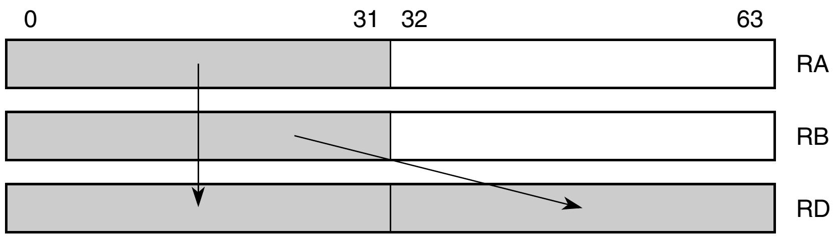


Figure 6-2. High order element merging with evmergehi

NOTE

A vector splat high can be performed by specifying the same register in **r**A and **r**B.

evmergehilo evmergehilo

Vector Merge High/Low

evmergehilo **r**D,**r**A,**r**B

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	010 0010 1110

```
RD0:31 = RA0:31
RD32:63 = RB32:63
```

The high-order element of **r**A and the low-order element of **r**B are merged and placed into **r**D as shown in Figure 6-3.

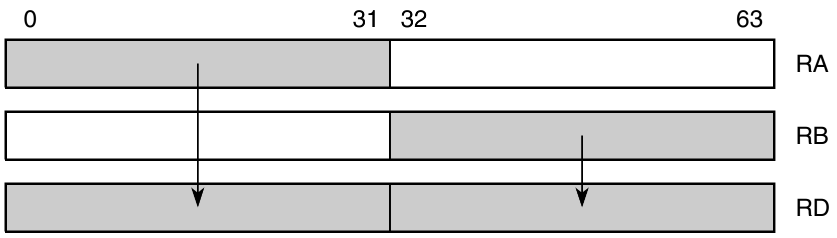


Figure 6-3. High order element merging with evmergehilo

Vector Merge Low

evmergelo **r**D,**r**A,**r**B

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

```
RD0:31 = RA32:63
RD32:63 = RB32:63
```

The low-order elements of **r**A and **r**B are merged and placed in **r**D as shown in Figure 6-4.

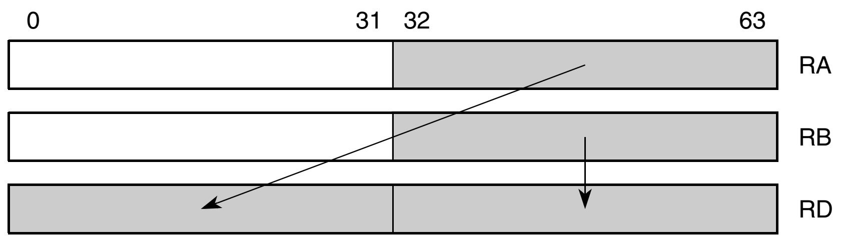


Figure 6-4. Low order element merging evmergelo

NOTE

A vector splat low can be performed by specifying the same register in **r**A and **r**B.

evmergelohi

Vector Merge Low/High

evmergelohi **r**D,**r**A,**r**B

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	010 0010 1111

```
RD0:31 = RA32:63
RD32:63 = RB0:31
```

The low-order element of **r**A and the high-order element of **r**B are merged and placed into **r**D as shown in Figure 6-5.

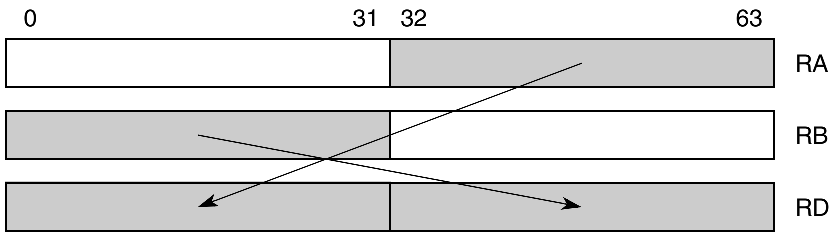


Figure 6-5. Low order element merging evmergelohi

NOTE

A vector swap can be performed by specifying the same register in **r**A and **r**B.

evnand

Vector NAND

evnand **r**D,**r**A,**r**B

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

```
$RD_{0:31}$ = ¬($RA_{0:31}$ & $RB_{0:31}$)                              // Bitwise NAND
$RD_{32:63}$ = ¬($RA_{32:63}$ & $RB_{32:63}$)                           // Bitwise NAND
```

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

evneg evneg

Vector Negate

evneg **r**D,**r**A

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

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

The negative value of each element of **r**A is placed in **r**D. The negative value of 0x8000_0000 (most negative number) returns 0x8000_0000. No overflow is detected.

evnor evnor

Vector NOR

evnor **r**D,**r**A,**r**B

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	010 0001 1000

```
$RD_{0:31}$ = ¬($RA_{0:31}$ | $RB_{0:31}$)                // Bitwise NOR
$RD_{32:63}$ = ¬($RA_{32:63}$ | $RB_{32:63}$)             // Bitwise NOR
```

Performs a bitwise NOR of each element of **r**A and **r**B and places the result into the corresponding element of **r**D.

evor evor

Vector OR

evor **r**D,**r**A,**r**B

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

```
$RD_{0:31}$ = $RA_{0:31}$ | $RB_{0:31}$                //Bitwise OR
$RD_{32:63}$ = $RA_{32:63}$ | $RB_{32:63}$             // Bitwise OR
```

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

Vector OR with Complement

evorc **r**D,**r**A,**r**B

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

```
$RD_{0:31} = RA_{0:31} | (\neg RB_{0:31})$          // Bitwise ORC
$RD_{32:63} = RA_{32:63} | (\neg RB_{32:63})$       // Bitwise ORC
```

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

evrlw

Vector Rotate Left Word

evrlw **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	010 0010 1000

```
nh = $RB_{27:31}$
nl = $RB_{59:63}$
$RD_{0:31}$ = ROTL($RA_{0:31}$, nh)
$RD_{32:63}$ = ROTL($RA_{32:63}$, nl)
```

Rotates left each of the elements of **r**A by amounts specified in **r**B and places the results into **r**D. The rotate amounts are specified by 5 bit fields in **r**B. Separate rotate values for each element of **r**A are specified in bit positions **r**$B_{27:31}$ and **r**$B_{59:63}$.

evrlwi evrlwi

Vector Rotate Left Word Immediate

evrlwi **r**D,**r**A,UIMM

0 5	6 10	11 15	16 20	21 31
4	RD	RA	UIMM	010 0010 1010

```
n = UIMM
$RD_{0:31}$ = ROTL($RA_{0:31}$, n)
$RD_{32:63}$ = ROTL($RA_{32:63}$, n)
```

Rotates left both elements of **r**A by an amount specified by the 5-bit UIMM immediate value and places the results into **r**D.

evrndw

Vector Round Word

evrndw **r**D,**r**A

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

```
RD0:31 = (RA0:31+0x00008000) & 0xFFFF0000                    // Modulo sum
RD32:63 = (RA32:63+0x00008000) & 0xFFFF0000                  // Modulo sum
```

Rounds the 32-bit elements of **r**A into 16 bits and places the results into **r**D. The resulting 16 bits of each element are placed in the most significant 16 bits of each element of **r**D, zeroing out the low order 16 bits of each element.

evsel evsel

Vector Select

evsel **r**D,**r**A,**r**B,**crf**S

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

```
ch = CR$_{crfS*4}$
cl = CR$_{crfS*4+1}$
if (ch == 1) then RD$_{0:31}$ = RA$_{0:31}$
else RD$_{0:31}$ = RB$_{0:31}$
if (cl == 1) then RD$_{32:63}$ = RA$_{32:63}$
else RD$_{32:63}$ = RB$_{32:63}$
```

If the msb if the **crf**S field of CR is set, the high-order element of **r**A is placed in the high-order element of **r**D; otherwise, the high-order element of **r**B is placed into the higher order element of **r**D. If the next most significant bit in the **crf**S field of CR is set, the low-order element of **r**A is placed in the low-order element of **r**D, otherwise, the low-order element of **r**B is placed into the lower order element of **r**D. This is shown in Figure 6-6.

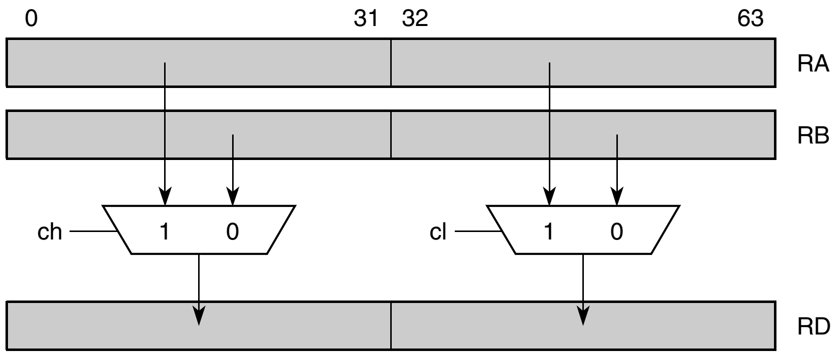


Figure 6-6. evsel

evslw evslw

Vector Shift Left Word

evslw **r**D,**r**A,**r**B

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	010 0010 0100

```
nh = RB$_{26:31}$
nl = RB$_{58:63}$
RD$_{0:31}$ = SL(RA$_{0:31}$, nh)
RD$_{32:63}$ = SL(RA$_{32:63}$, nl)
```

Shifts left each element of **r**A by amounts specified in **r**B and places the results into **r**D. The separate shift amounts for each element are specified by 6-bit fields in **r**B in bit positions 26:31 and 58:63. Shift amounts from 32 to 63 give a zero result.

evslwi evslwi

Vector Shift Left Word Immediate

evslwi **r**D,**r**A,UIMM

0 5	6 10	11 15	16 20	21 31
4	RD	RA	UIMM	010 0010 0110

```
n = UIMM
$RD_{0:31}$ = SL($RA_{0:31}$, n)
$RD_{32:63}$ = SL($RA_{32:63}$, n)
```

Shifts left each element of **r**A by the 5-bit UIMM value and places the results into **rD.**

evsplatfi evsplatfi

Vector Splat Fractional Immediate

evsplatfi **r**D,SIMM

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

```
RD0:31 = SIMM || 27 0
RD32:63 = SIMM || 27 0
```

The 5-bit SIMM value is padded with trailing zeros and placed into both elements of **r**D as shown in Figure 6-7. The SIMM value is placed in bit positions **r**D$_{0:4}$ and **r**D$_{32:36}$.

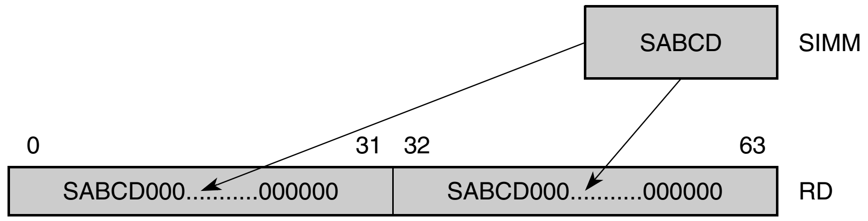


Figure 6-7. Splat for evsplatfi

evsplati

Vector Splat Immediate

evsplati rD,SIMM

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

```
RD$_{0:31}$ = EXTS(SIMM)
RD$_{32:63}$ = EXTS(SIMM)
```

The 5-bit SIMM immediate value is sign-extended and placed into both elements of **r**D as shown in Figure 6-8.

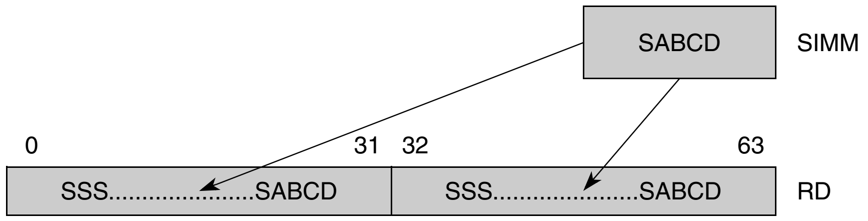

SABCD
SIMM
0
31
32
63
SSS......................SABCD
SSS......................SABCD
RD

Figure 6-8. Sign-extend in evsplati

evsrwis evsrwis

Vector Shift Right Word Immediate Signed

evsrwis **r**D,**r**A,UIMM

0 5	6 10	11 15	16 20	21 31
4	RD	RA	UIMM	010 0010 0011

```
n = UIMM
RD$_{0:31}$ = EXTS(RA$_{0:31-n}$)
RD$_{32:63}$ = EXTS(RA$_{32:63-n}$)
```

Shifts right arithmetically each element of **r**A by the 5-bit UIMM value and places the results into **r**D. The sign bit of each source element in **r**A is extended right into the most significant bit positions of each result element.

evsrwiu evsrwiu

Vector Shift Right Word Immediate Unsigned

evsrwiu **r**D,**r**A,UIMM

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

```
n = UIMM
RD_{0:31} = EXTZ(RA_{0:31-n})
RD_{32:63} = EXTZ(RA_{32:63-n})
```

Shifts right logically each element of **r**A by the 5-bit UIMM value and places the results into **r**D. ‘0’ bits are shifted in to the most significant bit positions of each result element.

evsrws

Vector Shift Right Word Signed

evsrws **r**D,**r**A,**r**B

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

```
nh = RB$_{26:31}$
nl = RB$_{58:63}$
RD$_{0:31}$ = EXTS(RA$_{0:31-nh}$)
RD$_{32:63}$ = EXTS(RA$_{32:63-nl}$)
```

Shifts right arithmetically each element of **r**A by an amount specified in **r**B and places the results into **r**D. Separate shift amounts for each element are specified by 6-bit fields in **r**B that occupy bit positions 26:31 and 58:63. The sign bit of each source element in **r**A is extended right into the most significant bit positions of each result element.

Shift amounts from 32 to 63 give a result of 32 sign bits.

evsrwu

Vector Shift Right Word Unsigned

evsrwu **r**D,**r**A,**r**B

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

```
nh = $RB_{26:31}$
nl = $RB_{58:63}$
$RD_{0:31}$ = EXTZ($RA_{0:31-nh}$)
$RD_{32:63}$ = EXTZ($RA_{32:63-nl}$)
```

Shifts right logically each element of **r**A by amounts specified in **r**B and places the results into **r**D. Separate shift amounts for each element are specified by 6-bit fields in **r**B that occupy bit positions 26:31 and 58:63. Zero bits are shifted in to the most significant bit positions.

Shift amounts from 32 to 63 give a zero result.

evsubfw

Vector Subtract from Word

evsubfw **r**D,**r**A,**r**B

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

```
RD0:31 = RB0:31 - RA0:31                                    // Modulo sum
RD32:63 = RB32:63 - RA32:63                                 // Modulo sum
```

Each element of **r**A is subtracted from the corresponding element of **r**B and the results are placed into the corresponding elements of **r**D.

evsubifw evsubifw

Vector Subtract Immediate from Word

evsubifw **r**D,UIMM,**r**B

0–5	6–10	11–15	16–20	21–31
4	RD	UIMM	RB	010 0000 0110

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

The 5-bit UIMM value is zero-extended and subtracted from each element of **r**B and the results are placed into the corresponding elements of **r**D. Note that the same value is subtracted from each element.

evxor evxor

Vector XOR

evxor rD,rA,rB

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

```
RD0:31 = RA0:31 ⊕ RB0:31                                    // Bitwise XOR
RD32:63 = RA32:63 ⊕ RB32:63                                 // Bitwise XOR
```

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

6.4 Integer SPE multiply, multiply-accumulate, and operation to accumulator instructions (complex integer instructions)

A number of forms of multiply and multiply-accumulate operations are supported in the SPE APU, as are add and subtract to accumulator operations. The SPE supports signed and unsigned forms, and optional fractional forms. For all of these instructions, the fractional form does not apply to unsigned forms because integer and fractional forms are identical for unsigned operands. Table 6-3 defines mnemonic extensions for these instructions.

Table 6-3. Mnemonic extensions for multiply-accumulate instructions

Extension	Meaning	Comments
Multiply form		
he	halfword even	$16 \times 16 \rightarrow 32$
heg	halfword even guarded	$16 \times 16 \rightarrow 32$, 64-bit final accum result
ho	halfword odd	$16 \times 16 \rightarrow 32$
hog	halfword odd guarded	$16 \times 16 \rightarrow 32$, 64-bit final accum result
w	word	$32 \times 32 \rightarrow 64$
wh	word high	$32 \times 32 \rightarrow 32$ high order 32 bits of product
wl	word low	$32 \times 32 \rightarrow 32$ low order 32 bits of product
Data type		
smf	signed modulo fractional	Wrap, no saturate
smi	signed modulo integer	Wrap, no saturate

Table 6-3. Mnemonic extensions for multiply-accumulate instructions (continued)

Extension	Meaning	Comments
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)

6.4.1 Multiply halfword instructions

The following instructions perform 16x16 multiplies from the odd or even half of elements, with and without accumulates, using signed or unsigned integer or fractional operands, and with optional saturation.

evmhegsmfaa evmhegsmfaa

Multiply Half Words, Even, Guarded, Signed, Modulo, Fractional and Accumulate

evmhegsmfaa **r**D,**r**A,**r**B (O=0, F=1, S=1)

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

```
$prod_{0:31}$ = $rA_{32:47}$ * $rB_{32:47}$
$temp1_{0:63}$ = EXTS($prod_{0:31}$ || 0)
$temp2_{0:64}$ = $ACC_{0:63}$ + $temp1_{0:63}$
$rD_{0:63}$ = $ACC_{0:63}$ = $temp2_{1:64}$
```

The low even-numbered signed fractional halfword element in **r**A is multiplied by the corresponding signed fractional halfword element in **r**B. The 32-bit intermediate product is sign-extended to 64 bits and then shifted left by one bit and added to the contents of the 64-bit accumulator to form a 65-bit intermediate sum. The lower 64 bits of the intermediate sum are placed back into the accumulator and also written into **r**D.

NOTE

This is a modulo sum. There is no check for overflow and no saturation is performed. An overflow from the 64-bit sum, if one occurs, is not recorded into SPEFSCR.

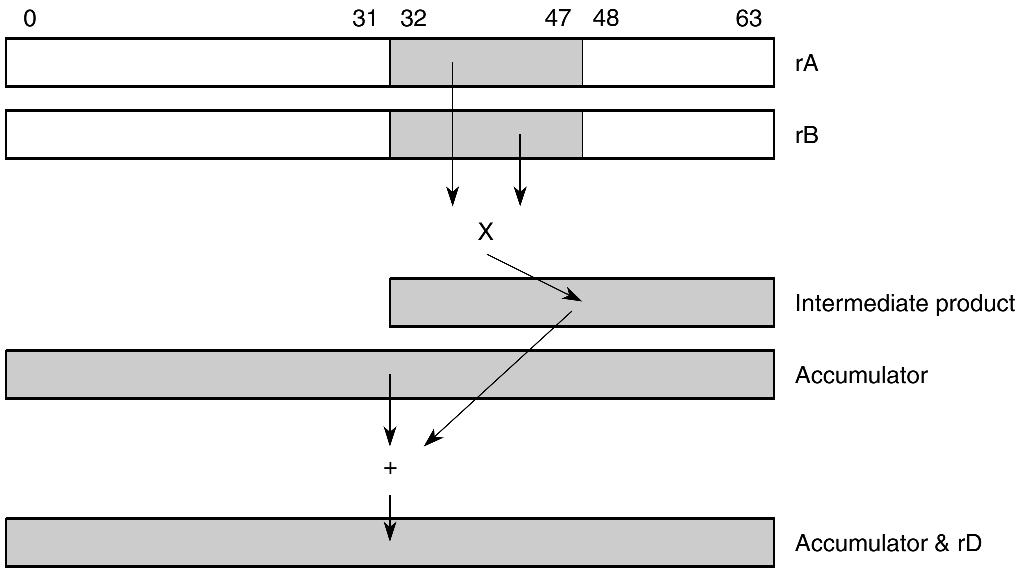


Figure 6-9. evmhegsmfaa

evmhegsmfan evmhegsmfan

Multiply Half Words, Even, Guarded, Signed, Modulo, Fractional and Accumulate Negative

evmhegsmfan **r**D,**r**A,**r**B (O=0, F=1, S=1)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	101 1010 1011

```
$prod_{0:31}$ = $rA_{32:47}$ * $rB_{32:47}$
$temp1_{0:63}$ = EXTS($prod_{0:31}$ || 0)
$temp2_{0:64}$ = $ACC_{0:63}$ - $temp1_{0:63}$
$rD_{0:63}$ = $ACC_{0:63}$ = $temp2_{1:64}$
```

The low even-numbered signed fractional halfword element in **r**A is multiplied by the corresponding signed fractional halfword element in **r**B. The 32-bit intermediate product is sign-extended to 64 bits and then shifted left by one bit and subtracted from the contents of the 64-bit accumulator to form a 65-bit intermediate difference. The lower 64 bits of the intermediate difference is placed back into the accumulator and also written into **r**D.

NOTE

This is a modulo difference. There is no check for overflow and no saturation is performed. An overflow from the 64-bit difference, if one occurs, is not recorded into SPEFSCR.

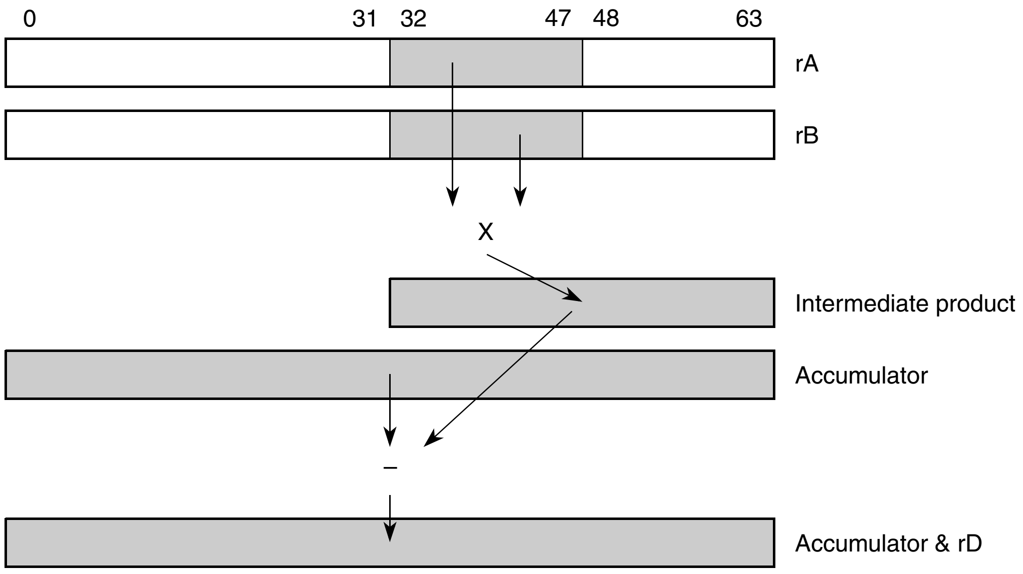


Figure 6-10. evmhegsmfan

evmhegsmiaa

Multiply Half Words, Even, Guarded, Signed, Modulo, Integer and Accumulate

evmhegsmiaa **r**D,**r**A,**r**B (O=0, F=0, S=1)

0 – 5	6 – 10	11 – 15	16 – 20	21 – 31
4	RD	RA	RB	101 0010 1001

```
$prod_{0:31}$ = $rA_{32:47}$ *si $rB_{32:47}$
$temp1_{0:63}$ = EXTS($prod_{0:31}$)
$temp2_{0:64}$ = $ACC_{0:63}$ + $temp1_{0:63}$
$rD_{0:63}$ = $ACC_{0:63}$ = $temp2_{1:64}$
```

The low even-numbered signed integer halfword element in **r**A is multiplied by the corresponding signed integer halfword element in **r**B. The intermediate product is sign-extended to 64 bits and added to the contents of the 64-bit accumulator to form a 65-bit intermediate sum. The lower 64 bits of the intermediate sum is placed back into the accumulator and also written into **r**D.

NOTE

This is a modulo sum. There is no check for overflow and no saturation is performed. An overflow from the 64-bit sum, if one occurs, is not recorded into SPEFSCR.

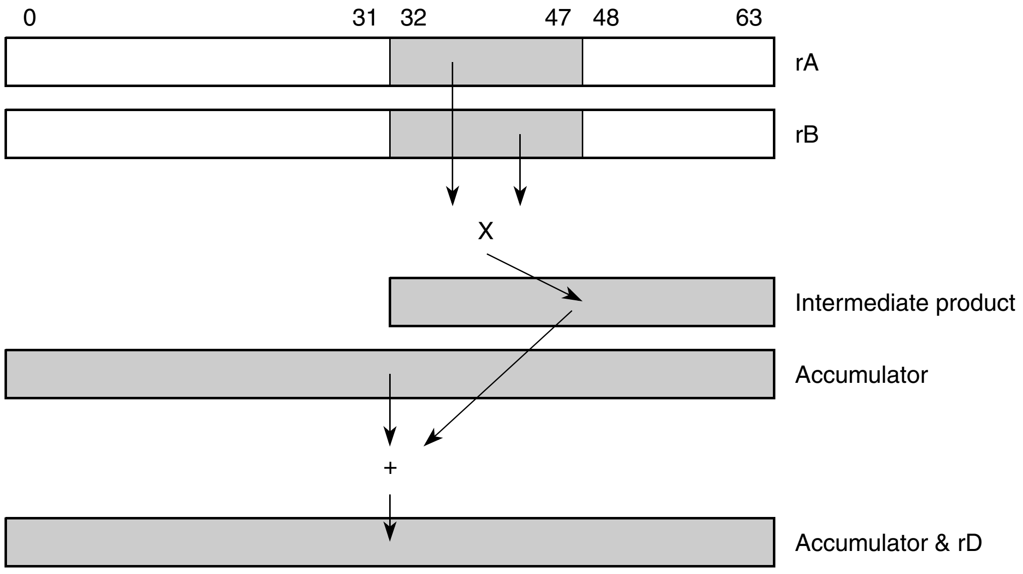


Figure 6-11. evmhegsmiaa

evmhegsmian evmhegsmian

Multiply Half Words, Even, Guarded, Signed, Modulo, Integer and Accumulate Negative

evmhegsmian **r**D,**r**A,**r**B (O=0, F=0, S=1)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	101 1010 1001

```
$prod_{0:31}$ = $rA_{32:47}$ *si $rB_{32:47}$
$temp1_{0:63}$ = EXTS($prod_{0:31}$)
$temp2_{0:64}$ = $ACC_{0:63}$ - $temp1_{0:63}$
$rD_{0:63}$ = $ACC_{0:63}$ = $temp2_{1:64}$
```

The low even-numbered signed integer halfword element in **r**A is multiplied by the corresponding signed integer halfword element in **r**B. The intermediate product is sign-extended to 64 bits and subtracted from the contents of the 64-bit accumulator to form a 65-bit intermediate difference. The lower 64 bits of the intermediate difference is placed back into the accumulator and also written into **r**D.

NOTE

This is a modulo difference. There is no check for overflow and no saturation is performed. An overflow from the 64-bit difference, if one occurs, is not recorded into SPEFSCR.

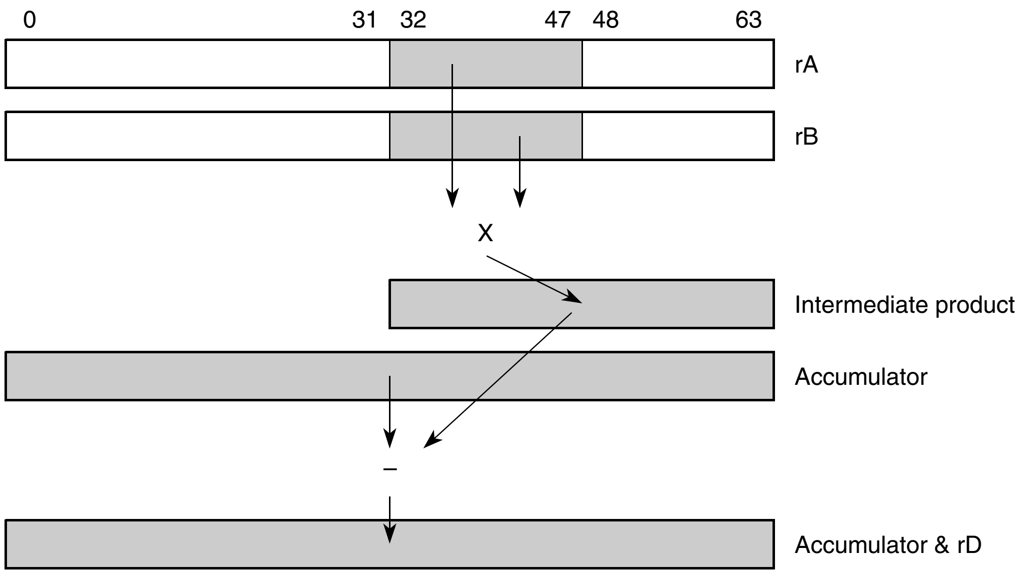


Figure 6-12. evmhegsmian

evmhegumiaa

Multiply Half Words, Even, Guarded, Unsigned, Modulo, Integer and Accumulate

evmhegumiaa **r**D,**r**A,**r**B (O=0, F=0, S=0)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	101 0010 1000

```
$prod_{0:31}$ = $rA_{32:47}$ *ui $rB_{32:47}$
$temp1_{0:63}$ = EXTZ($prod_{0:31}$)
$temp2_{0:64}$ = $ACC_{0:63}$ + $temp1_{0:63}$
$rD_{0:63}$ = $ACC_{0:63}$ = $temp2_{1:64}$
```

The low even-numbered unsigned integer halfword element in **r**A is multiplied by the corresponding unsigned integer halfword element in **r**B. The intermediate product is sign-extended to 64 bits and added to the contents of the 64-bit accumulator to form a 65-bit intermediate sum. The lower 64 bits of the intermediate sum is placed back into the accumulator and also written into **r**D.

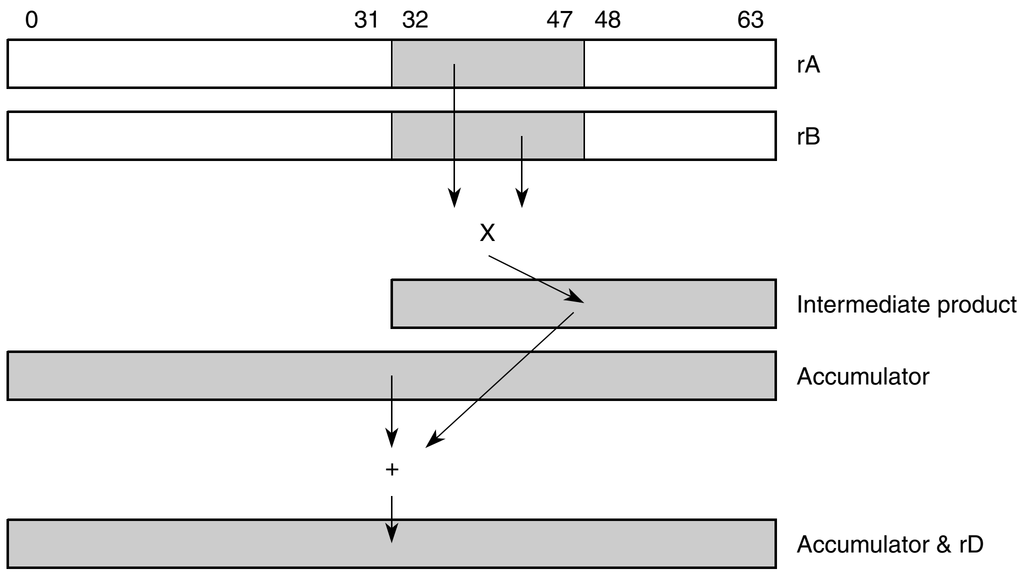


Figure 6-13. evmhegumiaa

evmhegumian evmhegumian

Multiply Half Words, Even, Guarded, Unsigned, Modulo, Integer and Accumulate Negative

evmhegumian **r**D,**r**A,**r**B (O=0, F=0, S=0)

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

```
prod$_{0:31}$ = rA$_{32:47}$ *ui rB$_{32:47}$
temp1$_{0:63}$ = EXTZ(prod$_{0:31}$)
temp2$_{0:64}$ = ACC$_{0:63}$ - temp1$_{0:63}$
rD$_{0:63}$ = ACC$_{0:63}$ = temp2$_{1:64}$
```

The low even-numbered unsigned integer halfword element in **r**A is multiplied by the corresponding unsigned integer halfword element in **r**B. The intermediate product is sign-extended to 64 bits and subtracted from the contents of the 64-bit accumulator to form a 65-bit intermediate difference. The lower 64 bits of the intermediate difference is placed back into the accumulator and also written into **r**D.

NOTE

This is a modulo difference. There is no check for overflow and no saturation is performed. An overflow from the 64-bit difference, if one occurs, is not recorded into SPEFSCR.

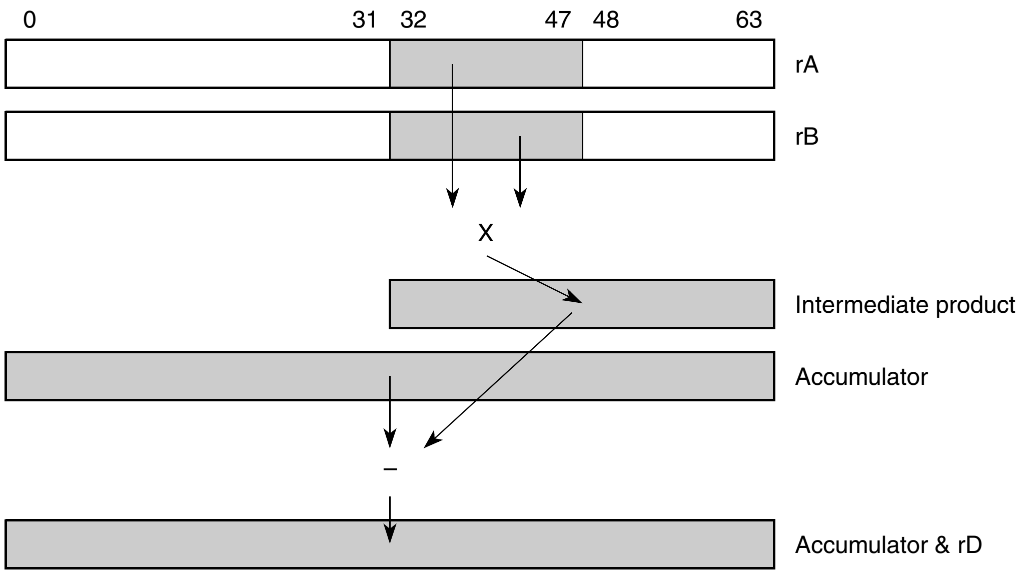


Figure 6-14. evmhegumian

evmhesmf evmhesmf

Vector Multiply Half Words, Even, Signed, Modulo, Fractional

evmhesmf **r**D,**r**A,**r**B (M=1, O=0, F=1, S=1, A=0)

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

```
$prod_{0:31}$ = $rA_{0:15}$ * $rB_{0:15}$
$prod_{32:63}$ = $rA_{32:47}$ * $rB_{32:47}$
$temp1_{0:32}$ = $prod_{0:31}$ || 0
$temp2_{0:32}$ = $prod_{32:63}$ || 0
$rD_{0:31}$ = $temp1_{1:32}$
$rD_{32:63}$ = $temp2_{1:32}$
```

Each even-numbered signed fractional halfword element in **r**A is multiplied by the corresponding signed fractional halfword element in **r**B. The two 32-bit signed fractional products are shifted left by one bit to remove the redundant sign bit, and are then placed into the two word elements of **r**D.

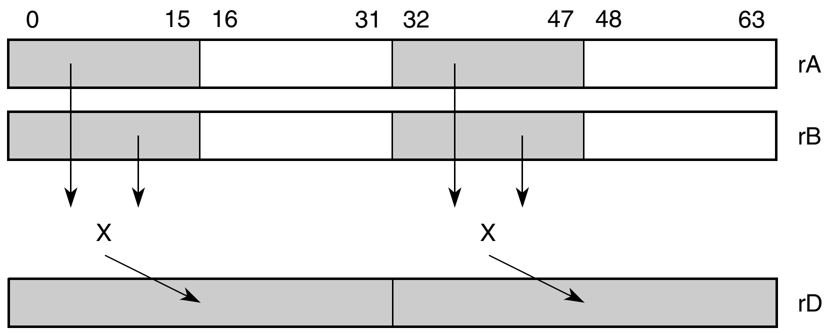


Figure 6-15. evmhesmf

evmhesmfa

Vector Multiply Half Words, Even, Signed, Modulo, Fractional, to Accumulator

evmhesmfa **r**D,**r**A,**r**B (M=1, O=0, F=1, S=1, A=1)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	100 0010 1011

```
prod$_{0:31}$ = rA$_{0:15}$ * rB$_{0:15}$
prod$_{32:63}$ = rA$_{32:47}$ * rB$_{32:47}$
temp1$_{0:32}$ = prod$_{0:31}$ || 0
temp2$_{0:32}$ = prod$_{32:63}$ || 0
rD$_{0:31}$ = temp1$_{1:32}$
rD$_{32:63}$ = temp2$_{1:32}$
ACC$_{0:63}$ = rD$_{0:63}$
```

Each even-numbered signed fractional halfword element in **r**A is multiplied by the corresponding signed fractional halfword element in **r**B. The two 32-bit signed fractional products are shifted left by one bit to remove the redundant sign bit, and are then placed into the two word elements of **r**D. The result in **r**D is also placed in the accumulator.

Other registers altered: ACC

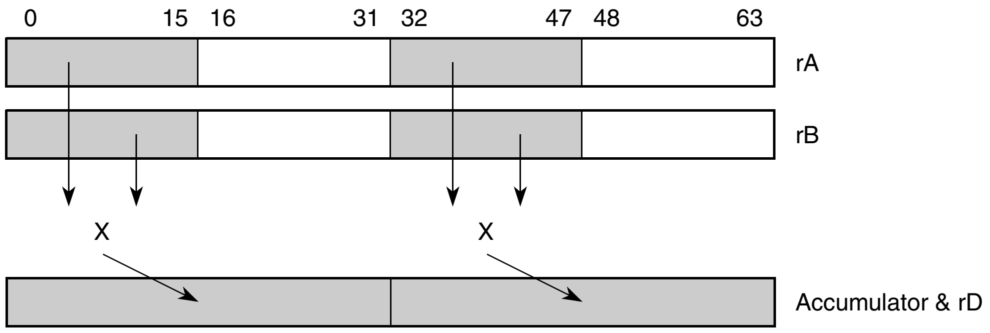


Figure 6-16. evmhesmfa

Vector Multiply Half Words, Even, Signed, Modulo, Fractional and Accumulate into Words

evmhesmfaaw **r**D,**r**A,**r**B (M=1, O=0, F=1, S=1)

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

```
temp1$_{0:32}$ = (rA$_{0:15}$ * rB$_{0:15}$) || 0
temp2$_{0:32}$ = (rA$_{32:47}$ * rB$_{32:47}$) || 0
temp3$_{0:32}$ = ACC$_{0:31}$ + temp1$_{1:32}$
temp4$_{0:32}$ = ACC$_{32:63}$ + temp2$_{1:32}$
ACC$_{0:31}$ = rD$_{0:31}$ = temp3$_{1:32}$
ACC$_{32:63}$ = rD$_{32:63}$ = temp4$_{1:32}$
```

For each word element in the accumulator the following operations are performed in the order shown:

Each even-numbered signed fractional halfword element in **r**A is multiplied by the corresponding signed fractional halfword element in **r**B.

The intermediate 32-bit product is shifted left by one bit to remove the redundant sign bit, and is then added to the contents of the accumulator word to form a 33-bit intermediate sum. The low 32 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word.

Other registers altered: ACC

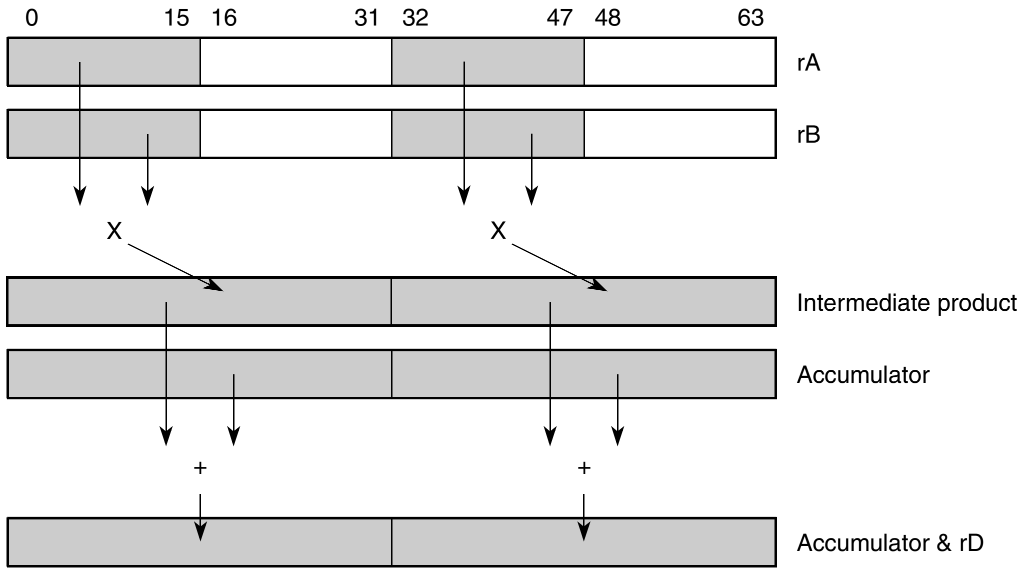


Figure 6-17. evmhesmfaaw

Vector Multiply Half Words, Even, Signed, Modulo, Fractional and Accumulate Negative into Words

evmhesmfanw **r**D,**r**A,**r**B (M=1, O=0, F=1, S=1)

0 … 5	6 … 10	11 … 15	16 … 20	21 … 31
4	RD	RA	RB	101 1000 1011

```
temp1_{0:32} = (rA_{0:15} * rB_{0:15}) || 0
temp2_{0:32} = (rA_{32:47} * rB_{32:47}) || 0
temp3_{0:32} = ACC_{0:31} - temp1_{1:32}
temp4_{0:32} = ACC_{32:63} - temp2_{1:32}
ACC_{0:31} = rD_{0:31} = temp3_{1:32}
ACC_{32:63} = rD_{32:63} = temp4_{1:32}
```

For each word element in the accumulator the following operations are performed in the order shown:

Each even-numbered signed fractional halfword element in **r**A is multiplied by the corresponding signed fractional halfword element in **r**B.

The intermediate 32-bit product is shifted left by one bit to remove the redundant sign bit, and is then subtracted from the contents of the accumulator word to form a 33-bit intermediate difference. The low 32 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word.

Other registers altered: ACC

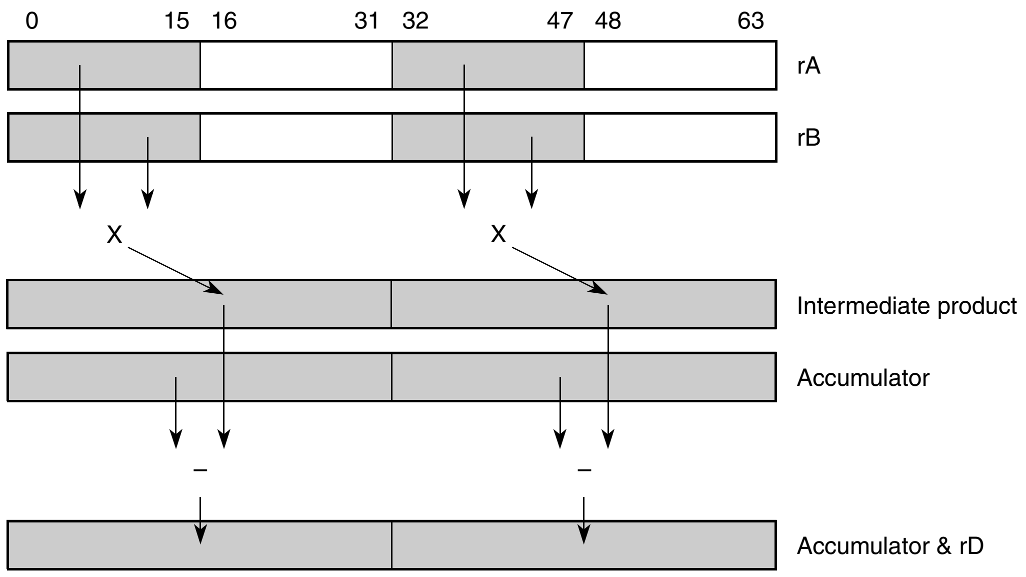


Figure 6-18. evmhesmfanw

evmhesmi evmhesmi

Vector Multiply Half Words, Even, Signed, Modulo, Integer

evmhesmi **r**D,**r**A,**r**B (M=1, O=0, F=0, S=1, A=0)

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

```
rD0:31 = rA0:15 *si rB0:15
rD32:63 = rA32:47 *si rB32:47
```

Each even-numbered signed integer halfword element in **r**A is multiplied by the corresponding signed integer halfword element in **r**B. The two 32-bit signed integer products are placed into the two word elements of **r**D.

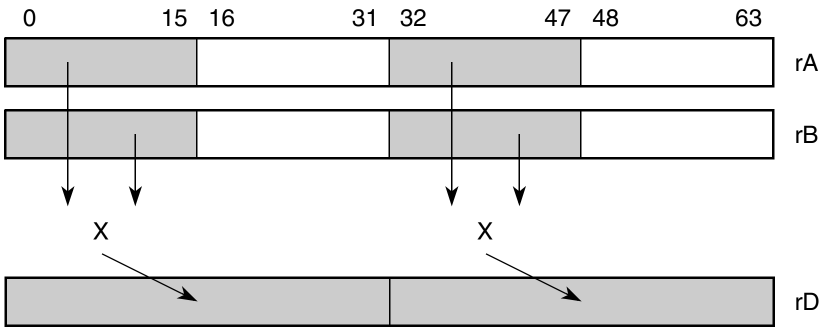


Figure 6-19. evmhesmi

Vector Multiply Half Words, Even, Signed, Modulo, Integer, to Accumulator

evmhesmia **r**D,**r**A,**r**B (M=1, O=0, F=0, S=1, A=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 0010 1001

```
$rD_{0:31}$ = $rA_{0:15}$ *si $rB_{0:15}$
$rD_{32:63}$ = $rA_{32:47}$ *si $rB_{32:47}$
$ACC_{0:63}$ = $rD_{0:63}$
```

Each even-numbered signed integer halfword element in **r**A is multiplied by the corresponding signed integer halfword element in **r**B. The two 32-bit signed integer products are placed into the two word elements of **r**D. The result in **r**D is also placed in the accumulator.

Other registers altered: ACC

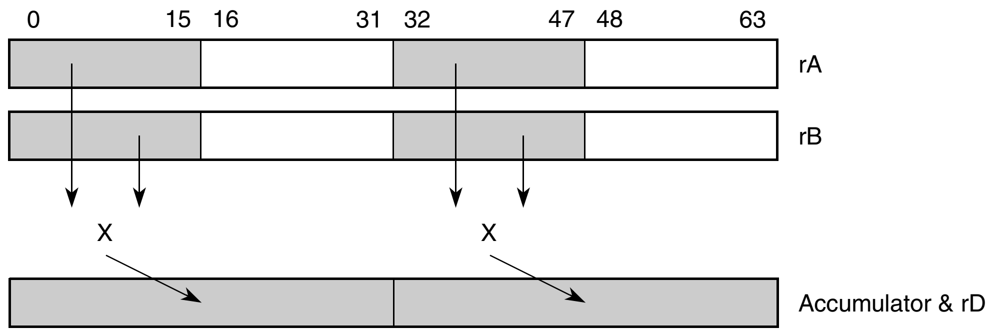


Figure 6-20. evmhesmia

evmhesmiaaw evmhesmiaaw

Vector Multiply Half Words, Even, Signed, Modulo, Integer and Accumulate into Words

evmhesmiaaw **r**D,**r**A,**r**B (M=1, O=0, F=0, S=1)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	101 0000 1001

```
temp1$_{0:31}$ = rA$_{0:15}$ *si rB$_{0:15}$
temp2$_{0:31}$ = rA$_{32:47}$ *si rB$_{32:47}$
temp3$_{0:32}$ = ACC$_{0:31}$ + temp1$_{0:31}$
temp4$_{0:32}$ = ACC$_{32:63}$ + temp2$_{0:31}$
ACC$_{0:31}$ = rD$_{0:31}$ = temp3$_{1:32}$
ACC$_{32:63}$ = rD$_{32:63}$ = temp4$_{1:32}$
```

For each word element in the accumulator the following operations are performed in the order shown:

Each even-numbered signed integer halfword element in **r**A is multiplied by the corresponding signed integer halfword element in **r**B.

The intermediate 32-bit product is added to the contents of the accumulator word to form a 33-bit intermediate sum. The low 32 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word.

Other registers altered: ACC

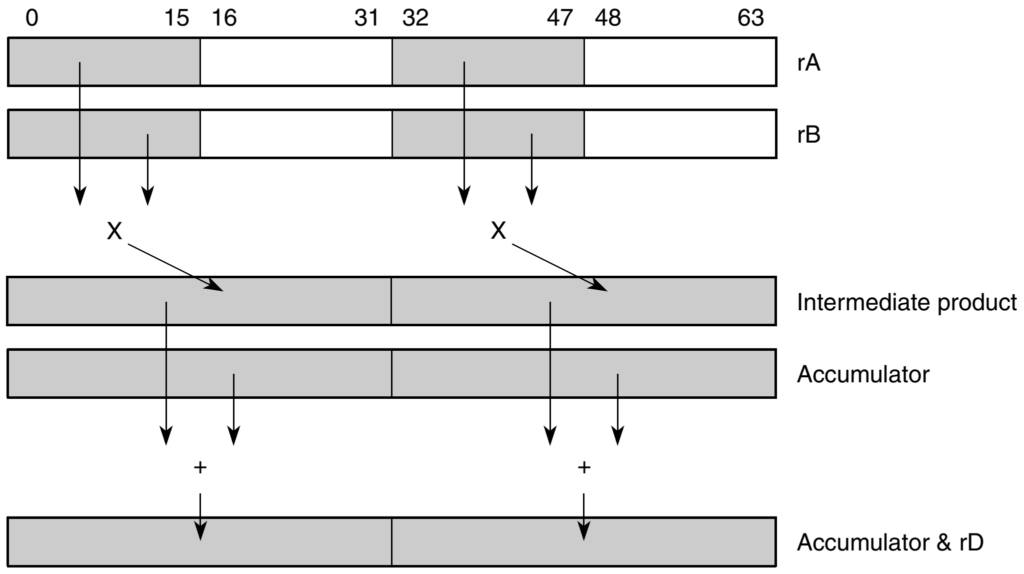


Figure 6-21. evmhesmiaaw

evmhesmianw evmhesmianw

Vector Multiply Half Words, Even, Signed, Modulo, Integer and Accumulate Negative into Words

evmhesmianw **r**D,**r**A,**r**B (M=1, O=0, F=0, S=1)

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

```
temp1$_{0:31}$ = rA$_{0:15}$ *si rB$_{0:15}$
temp2$_{0:31}$ = rA$_{32:47}$ *si rB$_{32:47}$
temp3$_{0:32}$ = ACC$_{0:31}$ - temp1$_{0:31}$
temp4$_{0:32}$ = ACC$_{32:63}$ - temp2$_{0:31}$
ACC$_{0:31}$ = rD$_{0:31}$ = temp3$_{1:32}$
ACC$_{32:63}$ = rD$_{32:63}$ = temp4$_{1:32}$
```

For each word element in the accumulator the following operations are performed in the order shown:

Each even-numbered signed integer halfword element in **r**A is multiplied by the corresponding signed integer halfword element in **r**B.

The intermediate 32-bit product is subtracted from the contents of the accumulator word to form a 33-bit intermediate difference. The low 32 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word.

Other registers altered: ACC

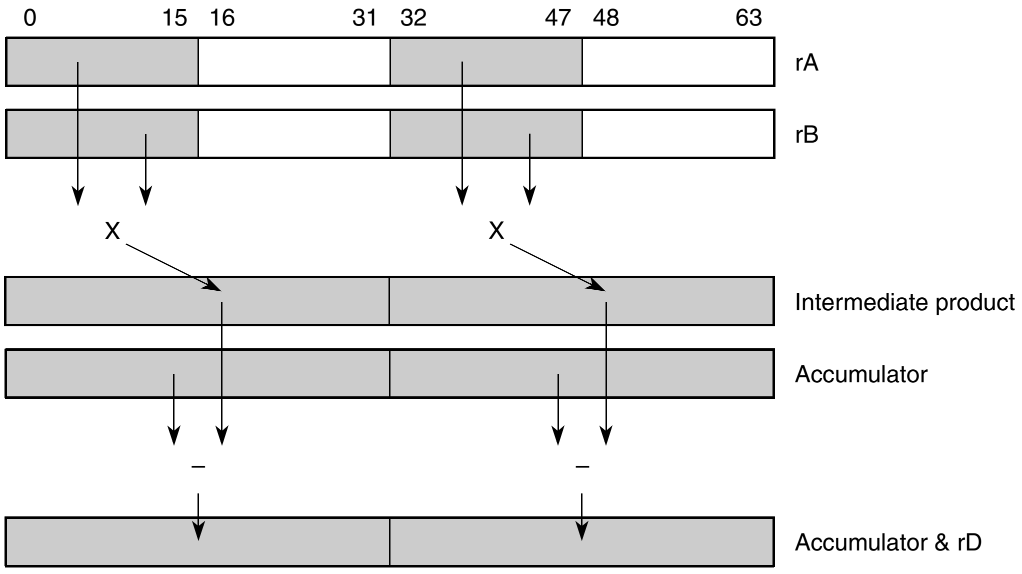


Figure 6-22. evmhesmianw

evmhessf evmhessf

Vector Multiply Half Words, Even, Signed, Saturate, Fractional

evmhessf **r**D,**r**A,**r**B (M=0, O=0, F=1, S=1, A=0)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	100 0000 0011

```
temp1$_{0:32}$ = (rA$_{0:15}$ * rB$_{0:15}$) || 0
temp2$_{0:32}$ = (rA$_{32:47}$ * rB$_{32:47}$) || 0
movh = temp1$_0$ $\oplus$ temp1$_1$
movl = temp2$_0$ $\oplus$ temp2$_1$
rD$_{0:31}$ = SATURATE(movh, 0x7FFFFFFF, temp1$_{1:32}$)
rD$_{32:63}$ = SATURATE(movl, 0x7FFFFFFF, temp2$_{1:32}$)
SPEFSCR$_{OVH}$ = movh
SPEFSCR$_{OV}$ = movl
SPEFSCR$_{SOVH}$ = SPEFSCR$_{SOVH}$ | movh
SPEFSCR$_{SOV}$ = SPEFSCR$_{SOV}$ | movl
```

Each even-numbered signed fractional halfword element in **r**A is multiplied by the corresponding signed fractional halfword element in **r**B. The two 32-bit signed fractional products are shifted left one bit to eliminate the redundant sign bit, and are then placed into the two word elements of **r**D. If the inputs are –1.0 and –1.0 the result is saturated to the most positive signed fraction (`0x7FFFFFFF`). If saturation occurs, the appropriate overflow and summary overflow bits are recorded in SPEFSCR.

Other registers altered: SPEFSCR

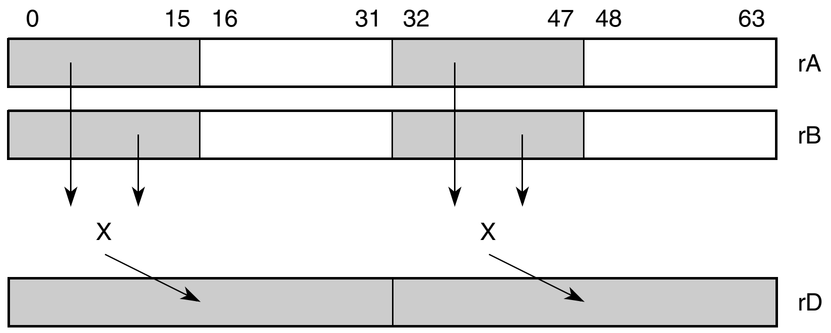


Figure 6-23. evmhessf

evmhessfa evmhessfa

Vector Multiply Half Words, Even, Signed, Saturate, Fractional, to Accumulator

evmhessfa **r**D,**r**A,**r**B (M=0, O=0, F=1, S=1, A=1)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	100 0010 0011

```
temp1$_{0:32}$ = (rA$_{0:15}$ * rB$_{0:15}$) || 0
temp2$_{0:32}$ = (rA$_{32:47}$ * rB$_{32:47}$) || 0
movh = temp1$_0$ ⊕ temp1$_1$
movl = temp2$_0$ ⊕ temp2$_1$
rD$_{0:31}$ = SATURATE(movh, 0x7FFFFFFF, temp1$_{1:32}$)
rD$_{32:63}$ = SATURATE(movl, 0x7FFFFFFF, temp2$_{1:32}$)
ACC$_{0:63}$ = rD$_{0:63}$
SPEFSCR$_{OVH}$ = movh
SPEFSCR$_{OV}$ = movl
SPEFSCR$_{SOVH}$ = SPEFSCR$_{SOVH}$ | movh
SPEFSCR$_{SOV}$ = SPEFSCR$_{SOV}$ | movl
```

Each even-numbered signed fractional halfword element in **r**A is multiplied by the corresponding signed fractional halfword element in **r**B. The two 32-bit signed fractional products are shifted left one bit to eliminate the redundant sign bit, and are then placed into the two word elements of **r**D. If the inputs are –1.0 and –1.0 the result is saturated to the most positive signed fraction (`0x7FFFFFFF`). The result in **r**D is also placed in the accumulator. If saturation occurs, the appropriate overflow and summary overflow bits are recorded in SPEFSCR.

Other registers altered: SPEFSCR, ACC

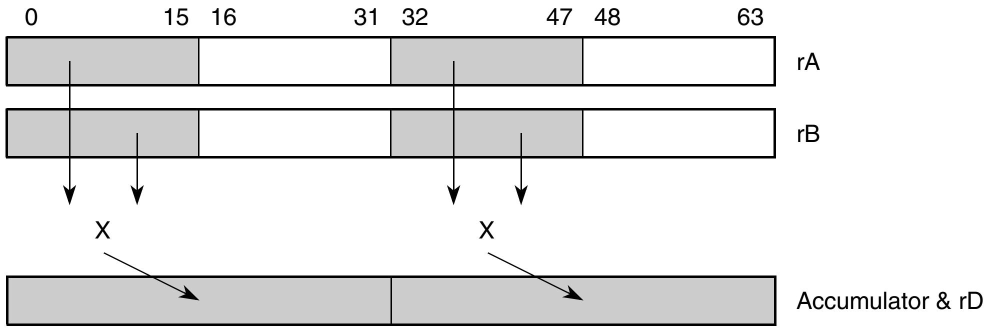


Figure 6-24. evmhessfa

evmhessfaaw

Vector Multiply Half Words, Even, Signed, Saturate, Fractional and Accumulate into Words

evmhessfaaw **r**D,**r**A,**r**B (M=0, O=0, F=1, S=1)

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

```
$temp1_{0:32}$ = ($rA_{0:15}$ * $rB_{0:15}$) || 0
$temp2_{0:32}$ = ($rA_{32:47}$ * $rB_{32:47}$) || 0
movh = $temp1_0 \oplus temp1_1$
movl = $temp2_0 \oplus temp2_1$
$temp3_{0:31}$ = SATURATE(movh, 0x7FFFFFFF, $temp1_{1:32}$)
$temp4_{0:31}$ = SATURATE(movl, 0x7FFFFFFF, $temp2_{1:32}$)
$temp5_{0:32}$ = {$ACC_0$,$ACC_{0:31}$} + {$temp3_0$,$temp3_{0:31}$}
$temp6_{0:32}$ = {$ACC_{32}$,$ACC_{32:63}$} + {$temp4_0$,$temp4_{0:31}$}
ovh = $temp5_0 \oplus temp5_1$
ovl = $temp6_0 \oplus temp6_1$
$rD_{0:31}$ = SATURATE_ACC(ovh, $temp5_0$, 0x80000000, 0x7FFFFFFF, $temp5_{1:32}$)
$rD_{32:63}$ = SATURATE_ACC(ovl, $temp6_0$, 0x80000000, 0x7FFFFFFF, $temp6_{1:32}$)
$ACC_{0:31}$ = $rD_{0:31}$
$ACC_{32:63}$ = $rD_{32:63}$
$SPEFSCR_{OVH}$ = movh | ovh
$SPEFSCR_{OV}$ = movl | ovl
$SPEFSCR_{SOVH}$ = $SPEFSCR_{SOVH}$ | movh | ovh
$SPEFSCR_{SOV}$ = $SPEFSCR_{SOV}$ | movl | ovl
```

For each word element in the accumulator the following operations are performed in the order shown:

Each even-numbered signed fractional halfword element in **r**A is multiplied by the corresponding signed fractional halfword element in **r**B. The two 32-bit signed fractional products are shifted left one bit to eliminate the redundant sign bit. If the inputs are –1.0 and –1.0 the intermediate result is saturated to the most positive signed fraction (`0x7FFFFFFF`).

The intermediate 32-bit product is added to the contents of the accumulator word to form an intermediate sum. If the intermediate sum has overflowed, the appropriate saturation value (`0x7FFFFFFF` if positive overflow or `0x80000000` if negative overflow) is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word.

If there is an overflow from either the multiply or the addition, the overflow information is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR, ACC

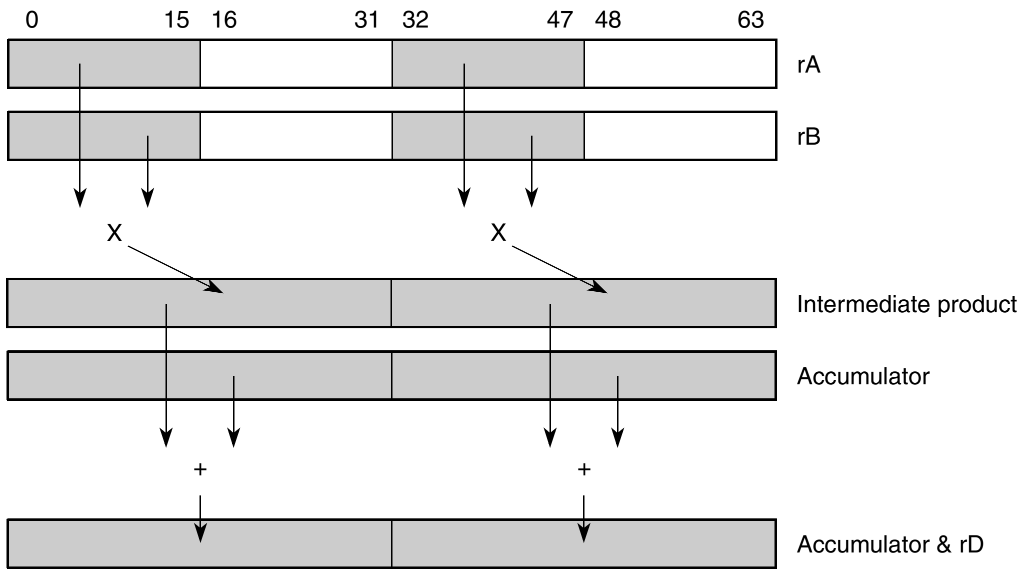


Figure 6-25. evmhessfaaw

evmhessfanw

Vector Multiply Half Words, Even, Signed, Saturate, Fractional and Accumulate Negative into Words

evmhessfanw **r**D,**r**A,**r**B (M=0, O=0, F=1, S=1)

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

```
temp1$_{0:32}$ = (rA$_{0:15}$ * rB$_{0:15}$) || 0
temp2$_{0:32}$ = (rA$_{32:47}$ * rB$_{32:47}$) || 0
movh = temp1$_0$ $\oplus$ temp1$_1$
movl = temp2$_0$ $\oplus$ temp2$_1$
temp3$_{0:31}$ = SATURATE(movh, 0x7FFFFFFF, temp1$_{1:32}$)
temp4$_{0:31}$ = SATURATE(movl, 0x7FFFFFFF, temp2$_{1:32}$)
temp5$_{0:32}$ = {ACC$_0$,ACC$_{0:31}$} - {temp3$_0$,temp3$_{0:31}$}
temp6$_{0:32}$ = {ACC$_{32}$,ACC$_{32:63}$} - {temp4$_0$,temp4$_{0:31}$}
ovh = temp5$_0$ $\oplus$ temp5$_1$
ovl = temp6$_0$ $\oplus$ temp6$_1$
rD$_{0:31}$ = SATURATE_ACC(ovh, temp5$_0$, 0x80000000, 0x7FFFFFFF, temp5$_{1:32}$)
rD$_{32:63}$ = SATURATE_ACC(ovl, temp6$_0$, 0x80000000, 0x7FFFFFFF, temp6$_{1:32}$)
ACC$_{0:31}$ = rD$_{0:31}$
ACC$_{32:63}$ = rD$_{32:63}$
SPEFSCR$_{OVH}$ = movh | ovh
SPEFSCR$_{OV}$ = movl | ovl
SPEFSCR$_{SOVH}$ = SPEFSCR$_{SOVH}$ | movh | ovh
SPEFSCR$_{SOV}$ = SPEFSCR$_{SOV}$ | movl | ovl
```

For each word element in the accumulator the following operations are performed in the order shown:

Each even-numbered signed fractional halfword element in **r**A is multiplied by the corresponding signed fractional halfword element in **r**B. The two 32-bit signed fractional products are shifted left one bit to eliminate the redundant sign bit. If the inputs are –1.0 and –1.0 the intermediate result is saturated to the most positive signed fraction (`0x7FFFFFFF`).

The intermediate 32-bit product is subtracted from the contents of the accumulator word to form an intermediate sum. If the intermediate difference has overflowed, the appropriate saturation value (`0x7FFFFFFF` if positive overflow or `0x80000000` if negative overflow) is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word.

If there is an overflow from either the multiply or the addition, the overflow information is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR, ACC

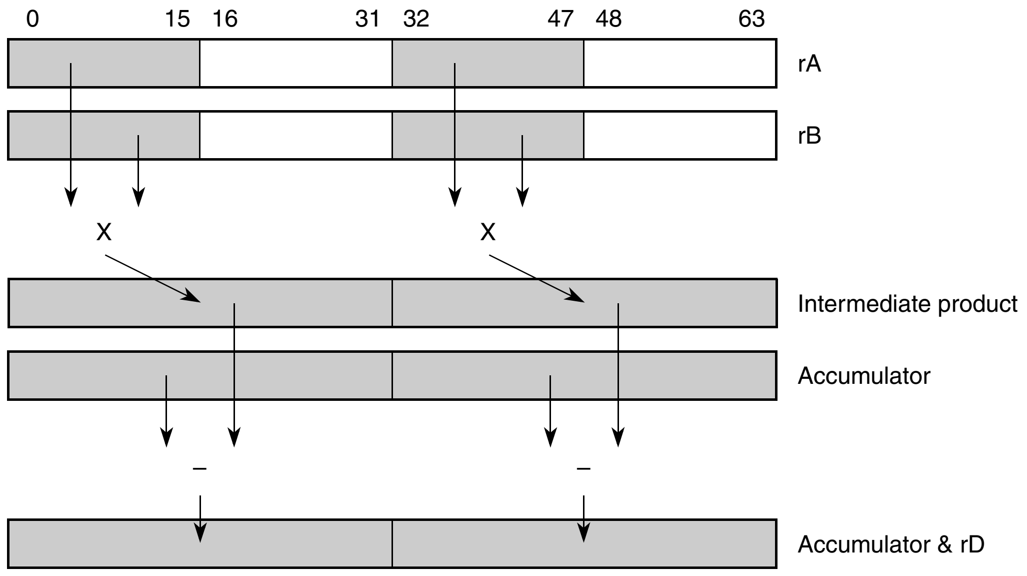


Figure 6-26. evmhessfanw

evmhessiaaw evmhessiaaw

Vector Multiply Half Words, Even, Signed, Saturate, Integer and Accumulate into Words

evmhessiaaw **r**D,**r**A,**r**B (M=0, O=0, F=0, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 0000 0001

```
temp1$_{0:31}$ = rA$_{0:15}$ *si rB$_{0:15}$
temp2$_{0:31}$ = rA$_{32:47}$ *si rB$_{32:47}$
temp3$_{0:32}$ = {ACC$_{0}$,ACC$_{0:31}$} + {temp1$_{0}$,temp1$_{0:31}$}
temp4$_{0:32}$ = {ACC$_{32}$,ACC$_{32:63}$} + {temp2$_{0}$,temp2$_{0:31}$}
ovh = temp3$_{0}$ ⊕ temp3$_{1}$
ovl = temp4$_{0}$ ⊕ temp4$_{1}$
rD$_{0:31}$ = SATURATE_ACC(ovh, temp3$_{0}$, 0x80000000, 0x7FFFFFFF, temp3$_{1:32}$)
rD$_{32:63}$ = SATURATE_ACC(ovl, temp4$_{0}$, 0x80000000, 0x7FFFFFFF, temp4$_{1:32}$)
ACC$_{0:31}$ = rD$_{0:31}$
ACC$_{32:63}$ = rD$_{32:63}$
SPEFSCR$_{OVH}$ = ovh
SPEFSCR$_{OV}$ = ovl
SPEFSCR$_{SOVH}$ = SPEFSCR$_{SOVH}$ | ovh
SPEFSCR$_{SOV}$ = SPEFSCR$_{SOV}$ | ovl
```

For each word element in the accumulator the following operations are performed in the order shown:

Each even-numbered signed integer halfword element in **r**A is multiplied by the corresponding signed integer halfword element in **r**B.

The intermediate 32-bit product is added to the contents of the accumulator word to form an intermediate sum. If the intermediate sum has overflowed, the appropriate saturation value (`0x7FFFFFFF` if positive overflow or `0x80000000` if negative overflow) is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word.

If there is an overflow from the addition, the overflow information is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR, ACC

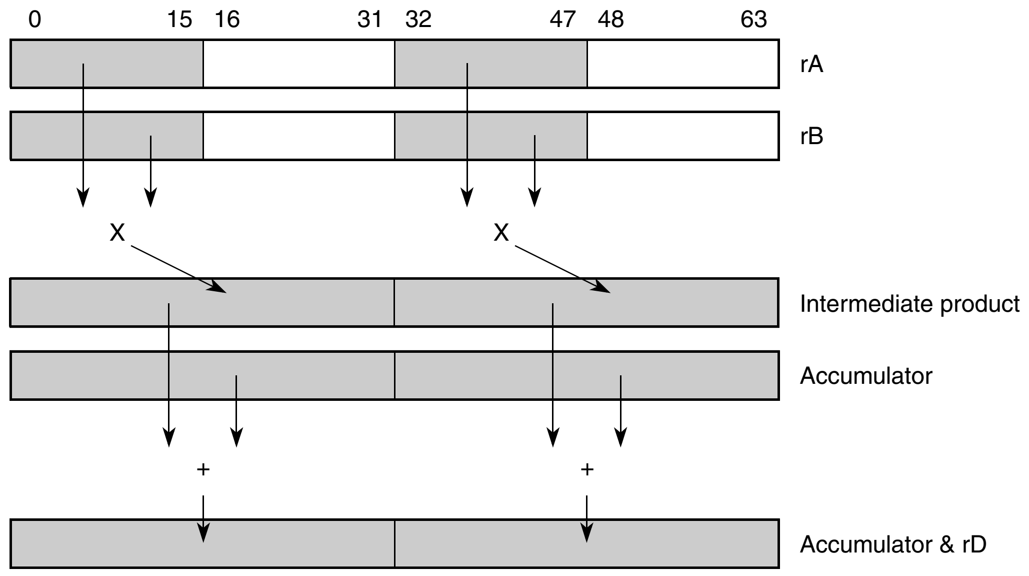


Figure 6-27. Even form of vector halfword multiply (evmhessiaaw)

evmhessianw

Vector Multiply Half Words, Even, Signed, Saturate, Integer and Accumulate Negative into Words

evmhessianw **r**D,**r**A,**r**B (M=0, O=0, F=0, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 1000 0001

```
$temp1_{0:31}$ = $rA_{0:15}$ *si $rB_{0:15}$
$temp2_{0:31}$ = $rA_{32:47}$ *si $rB_{32:47}$
    $temp3_{0:32}$ = {$ACC_0$,$ACC_{0:31}$} - {$temp1_0$,$temp1_{0:31}$}
    $temp4_{0:32}$ = {$ACC_{32}$,$ACC_{32:63}$} - {$temp2_0$,$temp2_{0:31}$}
    ovh = $temp3_0 \oplus temp3_1$
    ovl = $temp4_0 \oplus temp4_1$
    $rD_{0:31}$ = SATURATE_ACC(ovh, $temp3_0$, 0x80000000, 0x7FFFFFFF, $temp3_{1:32}$)
    $rD_{32:63}$ = SATURATE_ACC(ovl, $temp4_0$, 0x80000000, 0x7FFFFFFF, $temp4_{1:32}$)
    $ACC_{0:31}$ = $rD_{0:31}$
    $ACC_{32:63}$ = $rD_{32:63}$
    $SPEFSCR_{OVH}$ = ovh
    $SPEFSCR_{OV}$ = ovl
    $SPEFSCR_{SOVH}$ = $SPEFSCR_{SOVH}$ | ovh
    $SPEFSCR_{SOV}$ = $SPEFSCR_{SOV}$ | ovl
```

For each word element in the accumulator, the following operations are performed in the order shown:

Each even-numbered signed integer halfword element in **r**A is multiplied by the corresponding signed integer halfword element in **r**B.

The intermediate 32-bit product is subtracted from the contents of the accumulator word to form a 33-bit intermediate difference. If the intermediate difference has overflowed, the appropriate saturation value (`0x7FFFFFFF` if positive overflow or `0x80000000` if negative overflow) is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word.

If there is an overflow from the subtraction, the overflow information is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR, ACC

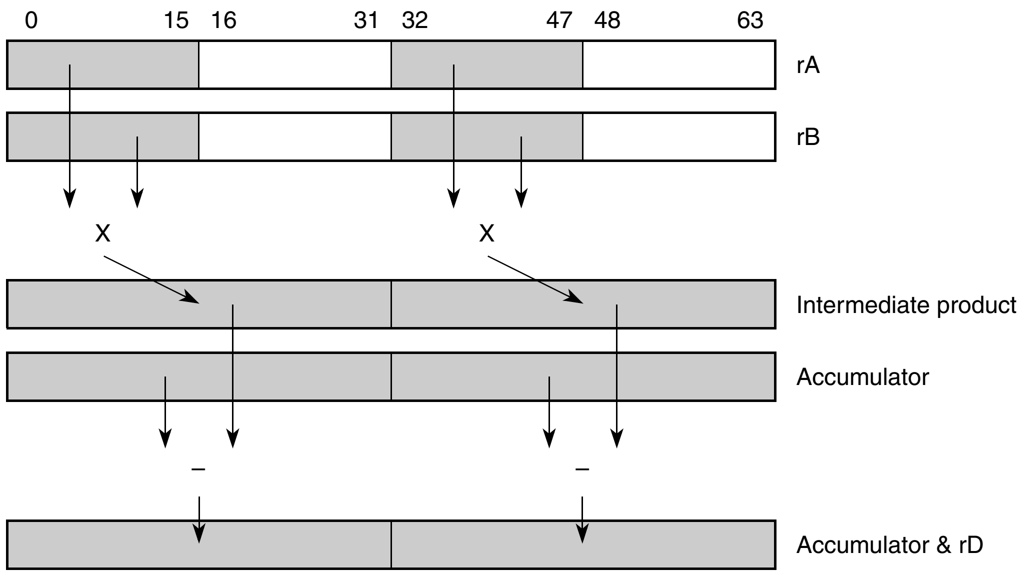


Figure 6-28. evmhessianw

evmheumi evmheumi

Vector Multiply Half Words, Even, Unsigned, Modulo, Integer

evmheumi **r**D,**r**A,**r**B (M=1, O=0, F=0, S=0, A=0)

0 … 5	6 … 10	11 … 15	16 … 20	21 … 31
4	RD	RA	RB	100 0000 1000

```
rD0:31 = rA0:15 *ui rB0:15
rD32:63 = rA32:47 *ui rB32:47
```

Each even-numbered unsigned integer halfword element in **r**A is multiplied by the corresponding unsigned integer halfword element in **r**B. The two 32-bit unsigned integer products are placed into the two word elements of **r**D.

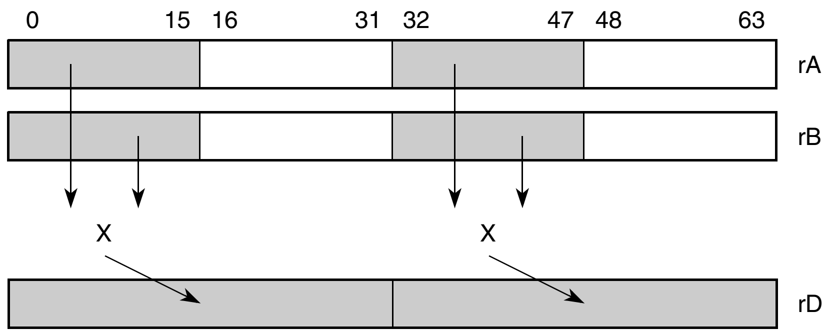


Figure 6-29. evmheumi — even multiply of two unsigned modulo integer elements

evmheumia evmheumia

Vector Multiply Half Words, Even, Unsigned, Modulo, Integer, to Accumulator

evmheumia **r**D,**r**A,**r**B (M=1, O=0, F=0, S=0, A=1)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	100 0010 1000

```
rD0:31 = rA0:15 *ui rB0:15
rD32:63 = rA32:47 *ui rB32:47
ACC0:63 = rD0:63
```

Each even-numbered unsigned integer halfword element in **r**A is multiplied by the corresponding unsigned integer halfword element in **r**B. The two 32-bit unsigned integer products are placed into the two word elements of **r**D. The result in **r**D is also placed in the accumulator.

Other registers altered: ACC

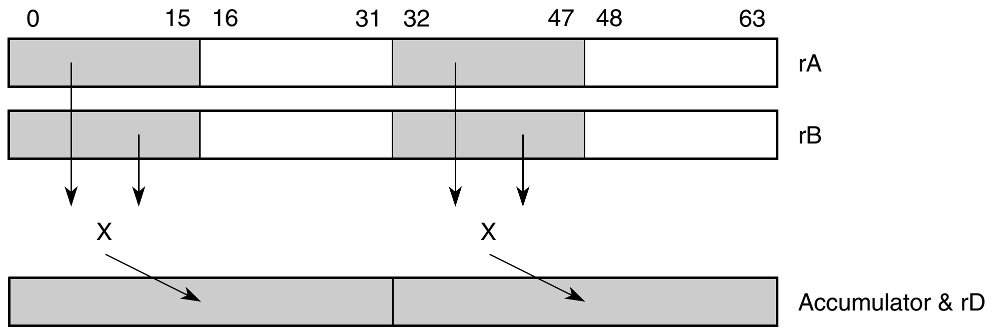


Figure 6-30. evmheumia

evmheumiaaw evmheumiaaw

Vector Multiply Half Words, Even, Unsigned, Modulo, Integer and Accumulate into Words

evmheumiaaw **r**D,**r**A,**r**B (M=1, O=0, F=0, S=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 0000 1000

```
temp1_{0:31} = rA_{0:15} *ui rB_{0:15}
temp2_{0:31} = rA_{32:47} *ui rB_{32:47}
temp3_{0:32} = ACC_{0:31} + temp1_{0:31}
temp4_{0:32} = ACC_{32:63} + temp2_{0:31}
ACC_{0:31} = rD_{0:31} = temp3_{1:32}
ACC_{32:63} = rD_{32:63} = temp4_{1:32}
```

For each word element in the accumulator the following operations are performed in the order shown:

Each even-numbered unsigned integer halfword element in **r**A is multiplied by the corresponding unsigned integer halfword element in **r**B. The intermediate 32-bit product is added to the contents of the accumulator word to form a 33-bit intermediate sum. The low 32 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word.

Other registers altered: ACC

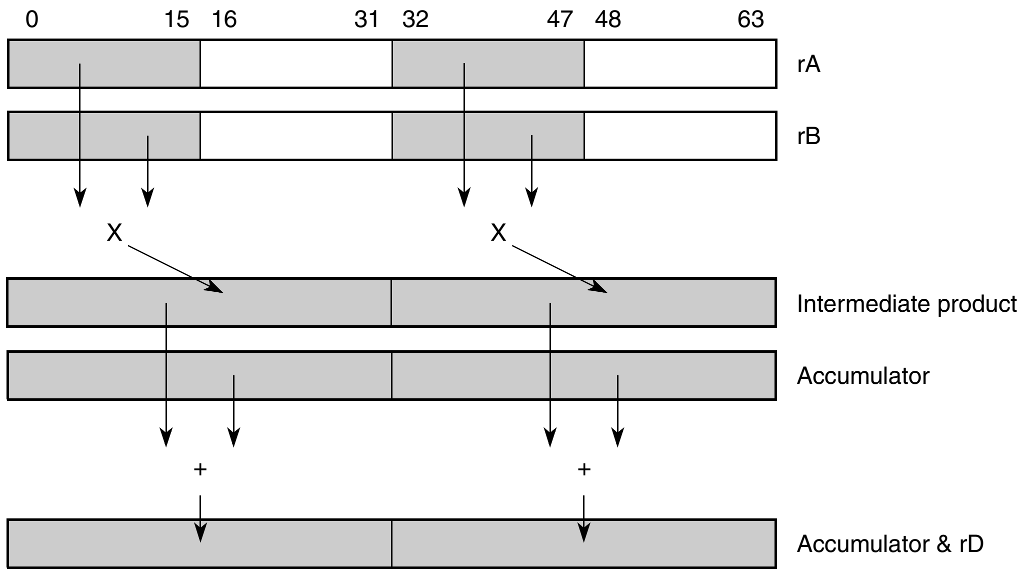


Figure 6-31. evmheumiaaw

evmheumianw evmheumianw

Vector Multiply Half Words, Even, Unsigned, Modulo, Integer and Accumulate Negative into Words

evmheumianw **r**D,**r**A,**r**B (M=1, O=0, F=0, S=0)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	101 1000 1000

```
$temp1_{0:31}$ = $rA_{0:15}$ *ui $rB_{0:15}$
$temp2_{0:31}$ = $rA_{32:47}$ *ui $rB_{32:47}$
$temp3_{0:32}$ = $ACC_{0:31}$ - $temp1_{0:31}$
$temp4_{0:32}$ = $ACC_{32:63}$ - $temp2_{0:31}$
$ACC_{0:31}$ = $rD_{0:31}$ = $temp3_{1:32}$
$ACC_{32:63}$ = $rD_{32:63}$ = $temp4_{1:32}$
```

For each word element in the accumulator the following operations are performed in the order shown:

Each even-numbered unsigned integer halfword element in **r**A is multiplied by the corresponding unsigned integer halfword element in **r**B.

The intermediate 32-bit product is subtracted from the contents of the accumulator word to form a 33-bit intermediate difference. The low 32 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word.

Other registers altered: ACC

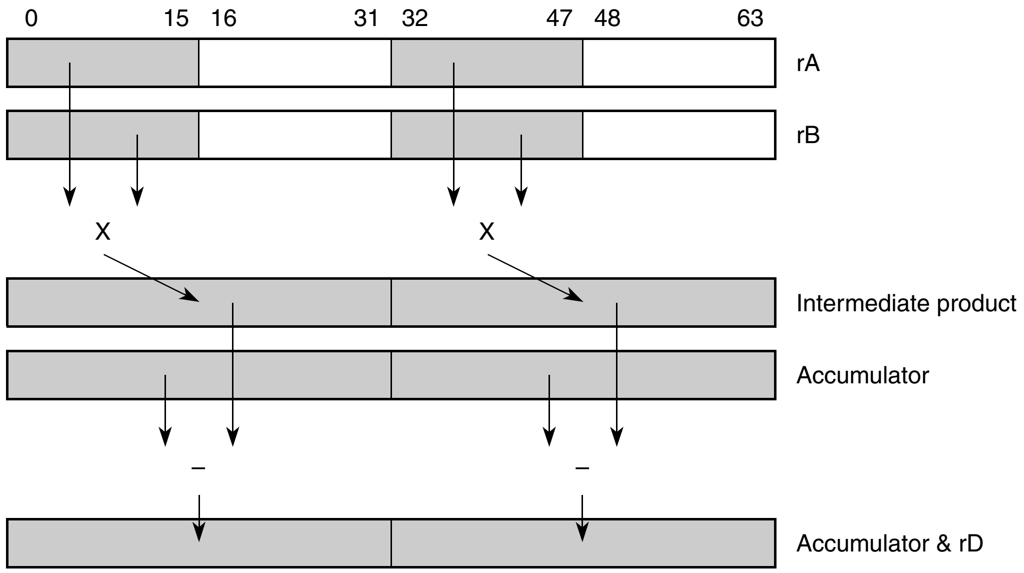


Figure 6-32. evmheumianw

evmheusiaaw evmheusiaaw

Vector Multiply Half Words, Even, Unsigned, Saturate, Integer and Accumulate into Words

evmheusiaaw **r**D,**r**A,**r**B (M=0, O=0, F=0, S=0)

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

```
temp1$_{0:31}$ = rA$_{0:15}$ *ui rB$_{0:15}$
temp2$_{0:31}$ = rA$_{32:47}$ *ui rB$_{32:47}$
temp3$_{0:32}$ = ACC$_{0:31}$ + temp1$_{0:31}$
temp4$_{0:32}$ = ACC$_{32:63}$ + temp2$_{0:31}$
ovh = temp3$_{0}$
ovl = temp4$_{0}$
rD$_{0:31}$ = SATURATE_ACC(ovh, 0xFFFFFFFF, temp3$_{1:32}$)
rD$_{32:63}$ = SATURATE_ACC(ovl, 0xFFFFFFFF, temp4$_{1:32}$)
ACC$_{0:31}$ = rD$_{0:31}$
ACC$_{32:63}$ = rD$_{32:63}$
SPEFSCR$_{OVH}$ = ovh
SPEFSCR$_{OV}$ = ovl
SPEFSCR$_{SOVH}$ = SPEFSCR$_{SOVH}$ | ovh
SPEFSCR$_{SOV}$ = SPEFSCR$_{SOV}$ | ovl
```

For each word element in the accumulator the following operations are performed in the order shown:

Each even-numbered unsigned integer halfword element in **r**A is multiplied by the corresponding unsigned integer halfword element in **r**B.

The intermediate 32-bit product is added to the contents of the accumulator word to form a 33-bit intermediate sum. If the intermediate sum has overflowed, the saturation value `0xFFFFFFFF` is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word.

If there is an overflow from the addition, the overflow information is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR, ACC

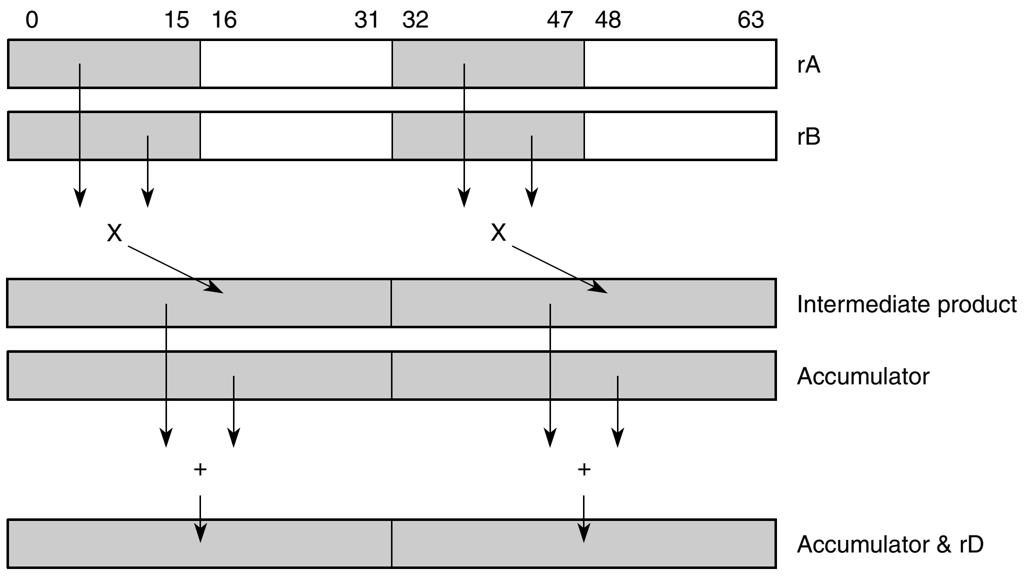


Figure 6-33. evmheusiaaw

evmheusianw evmheusianw

Vector Multiply Half Words, Even, Unsigned, Saturate, Integer and Accumulate Negative into Words

evmheusianw **r**D,**r**A,**r**B (M=0, O=0, F=0, S=0)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	101 1000 0000

```
$\text{temp1}_{0:31} = \text{rA}_{0:15} \text{ *ui } \text{rB}_{0:15}$
$\text{temp2}_{0:31} = \text{rA}_{32:47} \text{ *ui } \text{rB}_{32:47}$
$\text{temp3}_{0:32} = \text{ACC}_{0:31} - \text{temp1}_{0:31}$
$\text{temp4}_{0:32} = \text{ACC}_{32:63} - \text{temp2}_{0:31}$
$\text{ovh} = \text{temp3}_{0}$
$\text{ovl} = \text{temp4}_{0}$
$\text{rD}_{0:31} = \text{SATURATE\_ACC(ovh, 0x00000000, temp3}_{1:32}\text{)}$
$\text{rD}_{32:63} = \text{SATURATE\_ACC(ovl, 0x00000000, temp4}_{1:32}\text{)}$
$\text{ACC}_{0:31} = \text{rD}_{0:31}$
$\text{ACC}_{32:63} = \text{rD}_{32:63}$
$\text{SPEFSCR}_{\text{OVH}} = \text{ovh}$
$\text{SPEFSCR}_{\text{OV}} = \text{ovl}$
$\text{SPEFSCR}_{\text{SOVH}} = \text{SPEFSCR}_{\text{SOVH}} \mid \text{ovh}$
$\text{SPEFSCR}_{\text{SOV}} = \text{SPEFSCR}_{\text{SOV}} \mid \text{ovl}$
```

For each word element in the accumulator the following operations are performed in the order shown:

Each even-numbered unsigned integer halfword element in **r**A is multiplied by the corresponding unsigned integer halfword element in **r**B.

The intermediate 32-bit product is subtracted from the contents of the accumulator word to form a 33-bit intermediate difference. If the intermediate difference has underflowed (is negative), the saturation value `0x00000000` is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word.

If there is an underflow from the subtraction, the underflow information is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR, ACC

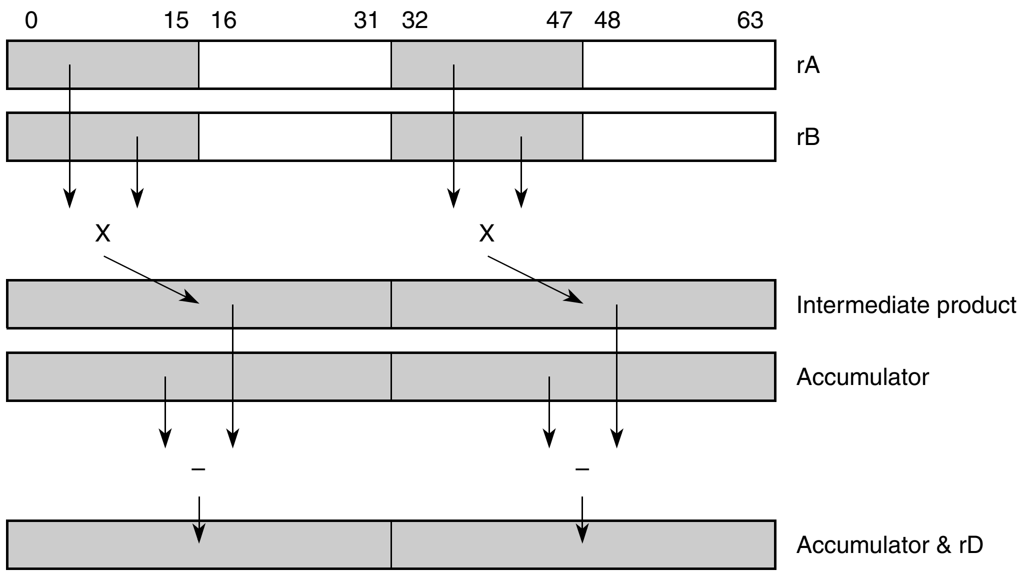


Figure 6-34. evmheusianw

evmhogsmfaa evmhogsmfaa

Multiply Half Words, Odd, Guarded, Signed, Modulo, Fractional and Accumulate

evmhogsmfaa **r**D,**r**A,**r**B (O=1, F=1, S=1)

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

```
$prod_{0:31} = rA_{48:63} * rB_{48:63}$
$temp1_{0:63} = EXTS(prod_{0:31} || 0)$
$temp2_{0:64} = ACC_{0:63} + temp1_{0:63}$
$rD_{0:63} = ACC_{0:63} = temp2_{1:64}$
```

The low odd-numbered signed fractional halfword element in **r**A is multiplied by the corresponding signed fractional halfword element in **r**B. The 32-bit intermediate product is sign-extended to 64 bits and then shifted left by one bit and added to the contents of the 64-bit accumulator to form a 65-bit intermediate sum. The lower 64 bits of the intermediate sum is placed back into the accumulator and also written into **r**D.

NOTE

This is a modulo sum. There is no check for overflow and no saturation is performed. An overflow from the 64-bit sum, if one occurs, is not recorded into SPEFSCR.

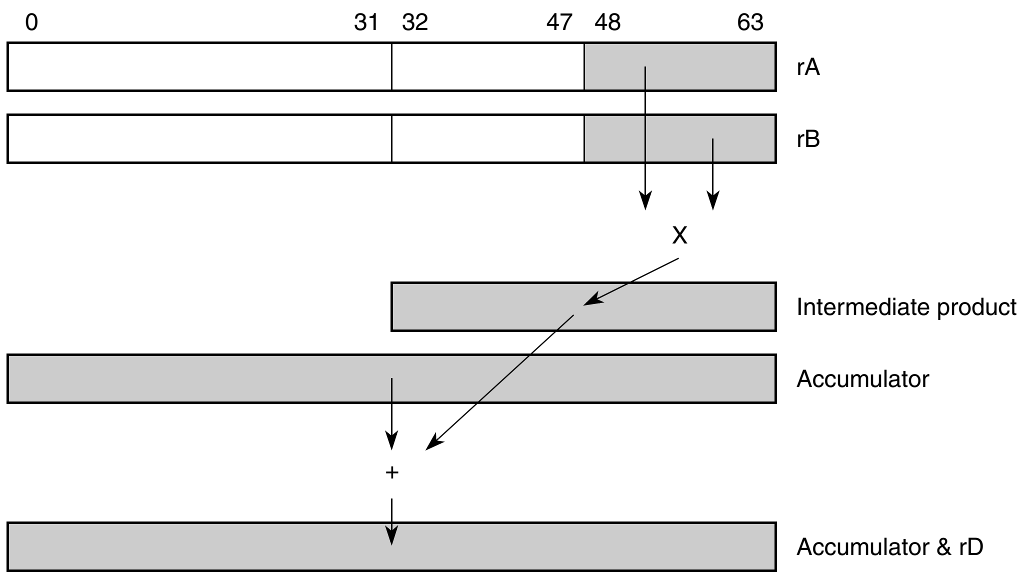


Figure 6-35. evmhogsmfaa

evmhogsmfan evmhogsmfan

Multiply Half Words, Odd, Guarded, Signed, Modulo, Fractional and Accumulate Negative

evmhogsmfan **r**D,**r**A,**r**B (O=1, F=1, S=1)

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

```
$prod_{0:31}$ = $rA_{48:63}$ * $rB_{48:63}$
$temp1_{0:63}$ = EXTS($prod_{0:31}$ || 0)
$temp2_{0:64}$ = $ACC_{0:63}$ - $temp1_{0:63}$
$rD_{0:63}$ = $ACC_{0:63}$ = $temp2_{1:64}$
```

The low odd-numbered signed fractional halfword element in **r**A is multiplied by the corresponding signed fractional halfword element in **r**B. The 32-bit intermediate product is sign-extended to 64 bits and then shifted left by one bit and subtracted from the contents of the 64-bit accumulator to form a 65-bit intermediate difference. The lower 64 bits of the intermediate difference is placed back into the accumulator and also written into **r**D.

NOTE

This is a modulo difference. There is no check for overflow and no saturation is performed. An overflow from the 64-bit difference, if one occurs, is not recorded into SPEFSCR.

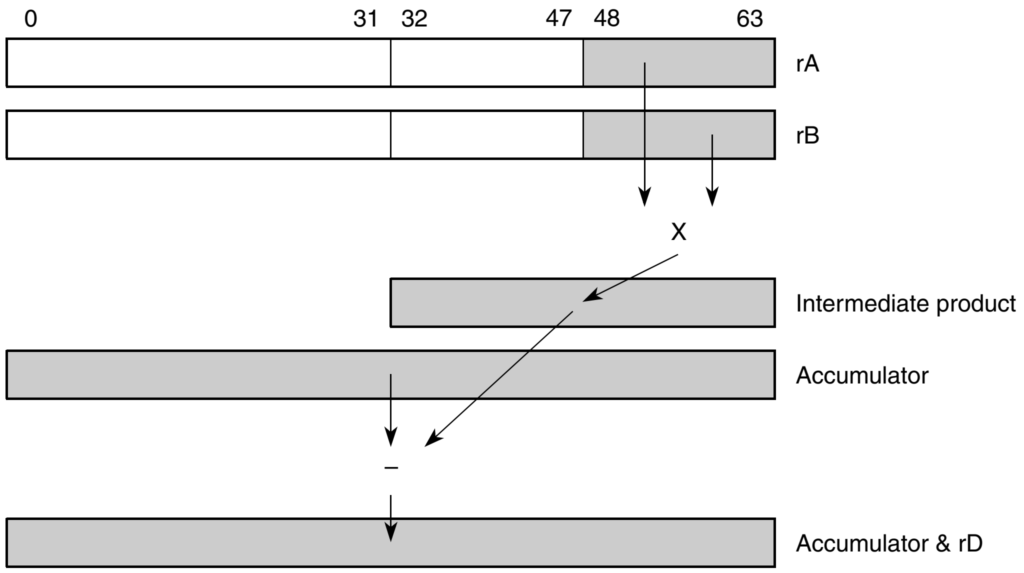


Figure 6-36. evmhogsmfan

evmhogsmiaa evmhogsmiaa

Multiply Half Words, Odd, Guarded, Signed, Modulo, Integer and Accumulate

evmhogsmiaa **r**D,**r**A,**r**B (O=1, F=0, S=1)

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

```
prod$_{0:31}$ = rA$_{48:63}$ *si rB$_{48:63}$
temp1$_{0:63}$ = EXTS(prod$_{0:31}$)
temp2$_{0:64}$ = ACC$_{0:63}$ + temp1$_{0:63}$
rD$_{0:63}$ = ACC$_{0:63}$ = temp2$_{1:64}$
```

The low odd-numbered signed integer halfword element in **r**A is multiplied by the corresponding signed integer halfword element in **r**B. The intermediate product is sign-extended to 64 bits and added to the contents of the 64-bit accumulator to form a 65-bit intermediate sum. The lower 64 bits of the intermediate sum is placed back into the accumulator and also written into **r**D.

NOTE

This is a modulo sum. There is no check for overflow and no saturation is performed. An overflow from the 64-bit sum, if one occurs, is not recorded into SPEFSCR.

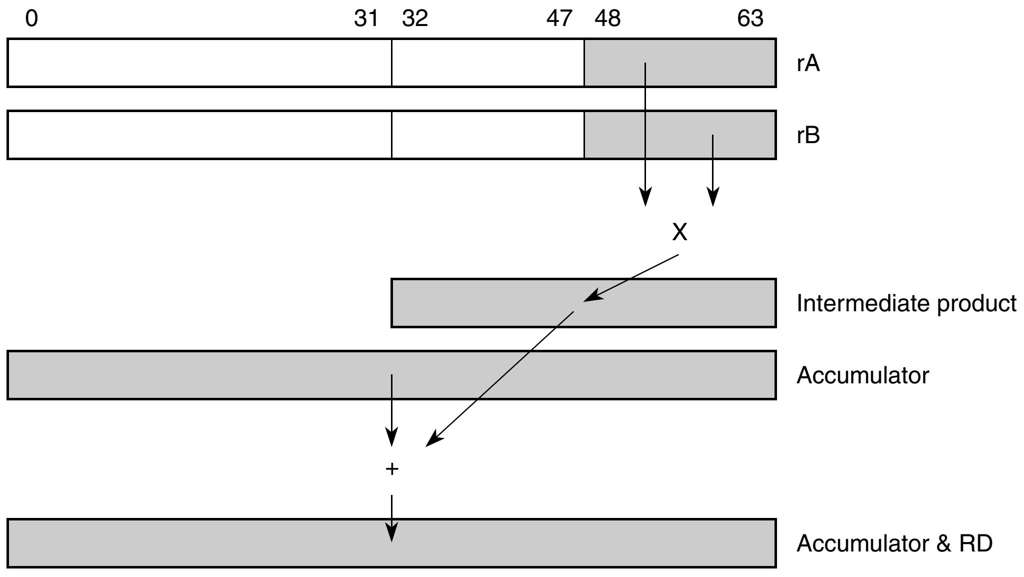


Figure 6-37. evmhogsmiaa

evmhogsmian evmhogsmian

Multiply Half Words, Odd, Guarded, Signed, Modulo, Integer and Accumulate Negative

evmhogsmian **r**D,**r**A,**r**B (O=1, F=0, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 1010 1101

```
$prod_{0:31}$ = $rA_{48:63}$ *si $rB_{48:63}$
$temp1_{0:63}$ = EXTS($prod_{0:31}$)
$temp2_{0:64}$ = $ACC_{0:63}$ - $temp1_{0:63}$
$rD_{0:63}$ = $ACC_{0:63}$ = $temp2_{1:64}$
```

The low odd-numbered signed integer halfword element in **r**A is multiplied by the corresponding signed integer halfword element in **r**B. The intermediate product is sign-extended to 64 bits and subtracted from the contents of the 64-bit accumulator to form a 65-bit intermediate difference. The lower 64 bits of the intermediate difference is placed back into the accumulator and also written into **r**D.

NOTE

This is a modulo difference. There is no check for overflow and no saturation is performed. An overflow from the 64-bit difference, if one occurs, is not recorded into SPEFSCR.

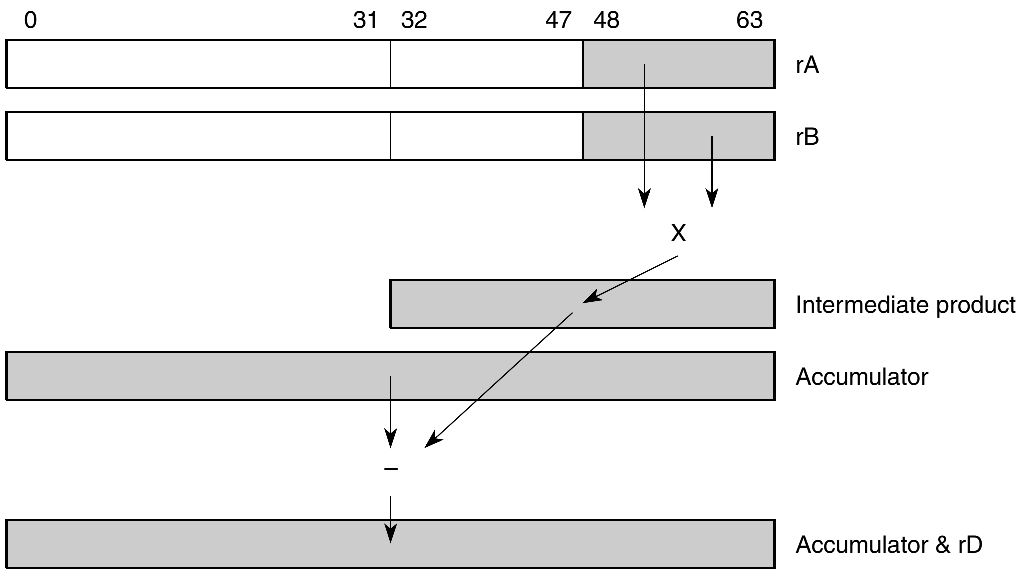


Figure 6-38. evmhogsmian

evmhogumiaa evmhogumiaa

Multiply Half Words, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate

evmhogumiaa **r**D,**r**A,**r**B (O=1, F=0, S=0)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	101 0010 1100

```
$prod_{0:31}$ = $rA_{48:63}$ *ui $rB_{48:63}$
$temp1_{0:63}$ = EXTZ($prod_{0:31}$)
$temp2_{0:64}$ = $ACC_{0:63}$ + $temp1_{0:63}$
$rD_{0:63}$ = $ACC_{0:63}$ = $temp2_{1:64}$
```

The low odd-numbered unsigned integer halfword element in **r**A is multiplied by the corresponding unsigned integer halfword element in **r**B. The intermediate product is sign-extended to 64 bits and added to the contents of the 64-bit accumulator to form a 65-bit intermediate sum. The lower 64 bits of the intermediate sum is placed back into the accumulator and also written into **r**D.

NOTE

This is a modulo sum. There is no check for overflow and no saturation is performed. An overflow from the 64-bit sum, if one occurs, is not recorded into SPEFSCR.

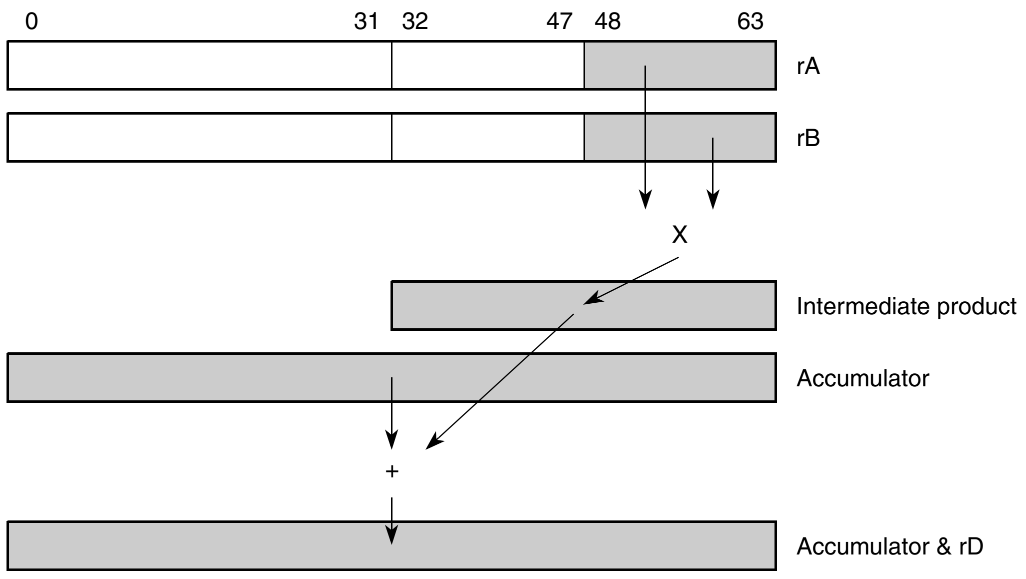


Figure 6-39. evmhogumiaa

evmhogumian   evmhogumian

Multiply Half Words, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate Negative

evmhogumian **r**D,**r**A,**r**B (O=1, F=0, S=0)

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

```
prod0:31 = rA48:63 *ui rB48:63
temp10:63 = EXTZ(prod0:31)
temp20:64 = ACC0:63 - temp10:63
```

The low odd-numbered unsigned integer halfword element in **r**A is multiplied by the corresponding unsigned integer halfword element in **r**B. The intermediate product is sign-extended to 64 bits and subtracted from the contents of the 64-bit accumulator to form a 65-bit intermediate difference. The lower 64 bits of the intermediate difference is placed back into the accumulator and also written into **r**D.

NOTE

This is a modulo difference. There is no check for overflow and no saturation is performed. An overflow from the 64-bit difference, if one occurs, is not recorded into SPEFSCR.

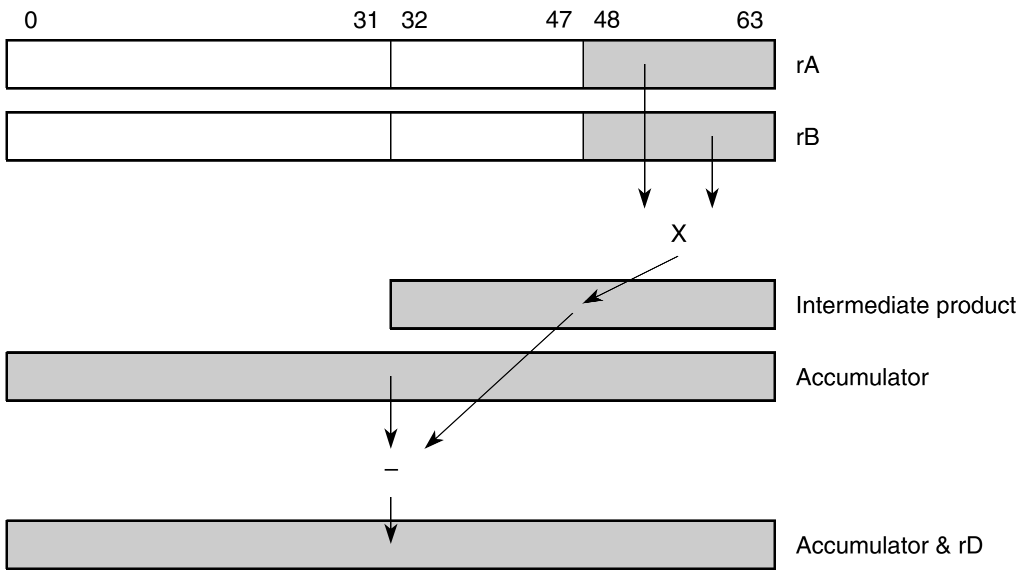


Figure 6-40. evmhogumian

evmhosmf evmhosmf

Vector Multiply Half Words, Odd, Signed, Modulo, Fractional

evmhosmf **r**D,**r**A,**r**B (M=1, O=1, F=1, S=1, A=0)

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

```
temp1$_{0:32}$ = rA$_{16:31}$ * rB$_{16:31}$
temp2$_{0:32}$ = rA$_{48:63}$ * rB$_{48:63}$
rD$_{0:31}$ = temp1$_{1:32}$
rD$_{32:63}$ = temp2$_{1:32}$
```

Each odd-numbered signed fractional halfword element in **r**A is multiplied by the corresponding signed fractional halfword element in **r**B. The two 32-bit signed fractional products are placed into the two word elements of **r**D.

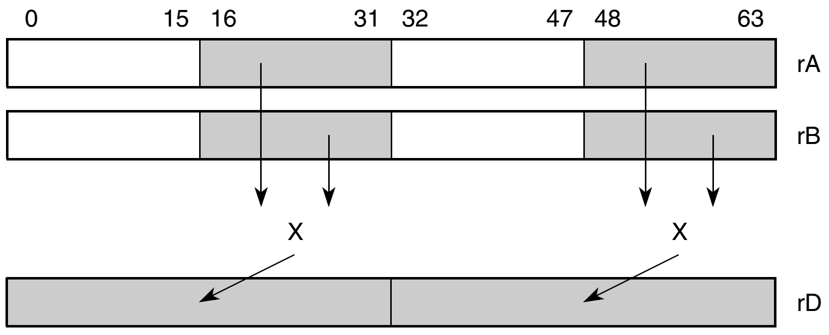


Figure 6-41. evmhosmf

evmhosmfa evmhosmfa

Vector Multiply Half Words, Odd, Signed, Modulo, Fractional, to Accumulator

evmhosmfa **r**D,**r**A,**r**B (M=1, O=1, F=1, S=1, A=1)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	100 0010 1111

```
$prod_{0:31} = rA_{16:31} * rB_{16:31}$
$prod_{32:63} = rA_{48:63} * rB_{48:63}$
$temp1_{0:32} = prod_{0:31} || 0$
$temp2_{0:32} = prod_{32:63} || 0$
$rD_{0:31} = temp1_{1:32}$
$rD_{32:63} = temp2_{1:32}$
$ACC_{0:63} = rD_{0:63}$
```

Each odd-numbered signed fractional halfword element in **r**A is multiplied by the corresponding signed fractional halfword element in **r**B. The two 32-bit signed fractional products are shifted left by one bit to remove the redundant sign bit, and are then placed into the two word elements of **r**D. The result in **r**D is also placed in the accumulator.

Other registers altered: ACC

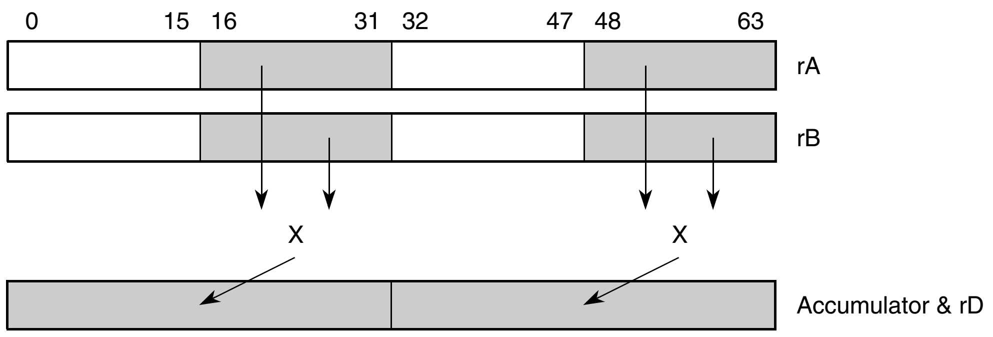


Figure 6-42. evmhosmfa

evmhosmfaaw

Vector Multiply Half Words, Odd, Signed, Modulo, Fractional and Accumulate into Words

evmhosmfaaw **r**D,**r**A,**r**B (M=1, O=1, F=1, S=1)

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

```
temp1$_{0:32}$ = (rA$_{16:31}$ * rB$_{16:31}$) || 0
temp2$_{0:32}$ = (rA$_{48:63}$ * rB$_{48:63}$) || 0
temp3$_{0:32}$ = ACC$_{0:31}$ + temp1$_{1:32}$
temp4$_{0:32}$ = ACC$_{32:63}$ + temp2$_{1:32}$
ACC$_{0:31}$ = rD$_{0:31}$ = temp3$_{1:32}$
ACC$_{32:63}$ = rD$_{32:63}$ = temp4$_{1:32}$
```

For each word element in the accumulator the following operations are performed in the order shown:

Each odd-numbered signed fractional halfword element in **r**A is multiplied by the corresponding signed fractional halfword element in **r**B.

The intermediate 32-bit product is shifted left by one bit to remove the redundant sign bit, and is then added to the contents of the accumulator word to form a 33-bit intermediate sum. The low 32 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word.

Other registers altered: ACC

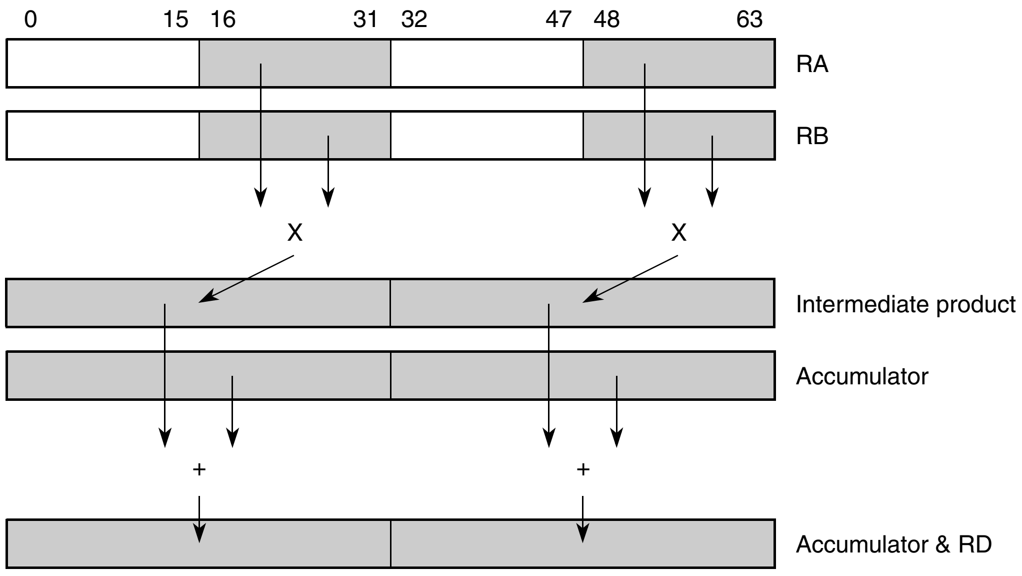


Figure 6-43. evmhosmfaaw

evmhosmfanw

Vector Multiply Half Words, Odd, Signed, Modulo, Fractional and Accumulate Negative into Words

evmhosmfanw **r**D,**r**A,**r**B (M=1, O=1, F=1, S=1)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	101 1000 1111

```
temp1$_{0:32}$ = (rA$_{16:31}$ * rB$_{16:31}$) || 0
temp2$_{0:32}$ = (rA$_{48:63}$ * rB$_{48:63}$) || 0
temp3$_{0:32}$ = ACC$_{0:31}$ - temp1$_{1:32}$
temp4$_{0:32}$ = ACC$_{32:63}$ - temp2$_{1:32}$
ACC$_{0:31}$ = rD$_{0:31}$ = temp3$_{1:32}$
ACC$_{32:63}$ = rD$_{32:63}$ = temp4$_{1:32}$
```

For each word element in the accumulator the following operations are performed in the order shown:

Each odd-numbered signed fractional halfword element in **r**A is multiplied by the corresponding signed fractional halfword element in **r**B.

The intermediate 32-bit product is shifted left by one bit to remove the redundant sign bit, and is then subtracted from the contents of the accumulator word to form a 33-bit intermediate difference. The low 32 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word.

Other registers altered: ACC

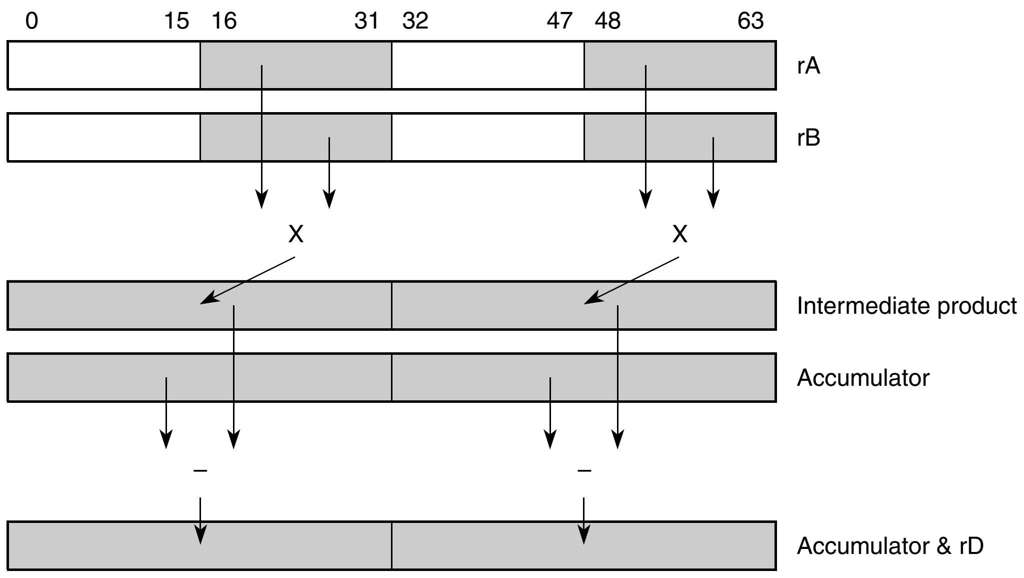


Figure 6-44. evmhosmfanw

evmhosmi evmhosmi

Vector Multiply Half Words, Odd, Signed, Modulo, Integer

evmhosmi **r**D,**r**A,**r**B (M=1, O=1, F=0, S=1, A=0)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	100 0000 1101

$rD_{0:31}$ = $rA_{16:31}$ *si $rB_{16:31}$
$rD_{32:63}$ = $rA_{48:63}$ *si $rB_{48:63}$

Each odd-numbered signed integer halfword element in **r**A is multiplied by the corresponding signed integer halfword element in **r**B. The two 32-bit signed integer products are placed into the two word elements of **r**D.

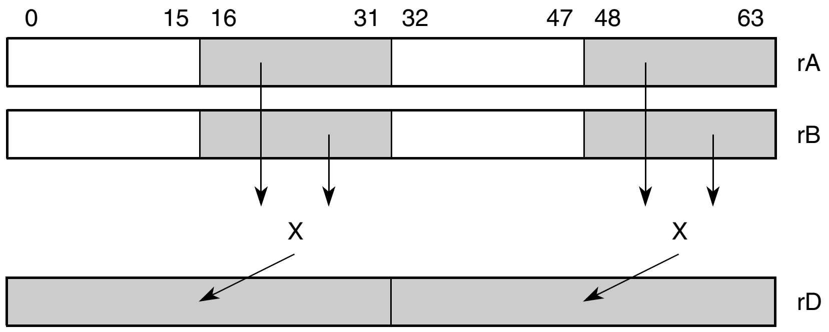


Figure 6-45. evmhosmi

evmhosmia evmhosmia

Vector Multiply Half Words, Odd, Signed, Modulo, Integer, to Accumulator

evmhosmia **r**D,**r**A,**r**B (M=1, O=1, F=0, S=1, A=1)

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

$rD_{0:31}$ = $rA_{16:31}$ *si $rB_{16:31}$
$rD_{32:63}$ = $rA_{48:63}$ *si $rB_{48:63}$
$ACC_{0:63}$ = $rD_{0:63}$

Each odd-numbered signed integer halfword element in **r**A is multiplied by the corresponding signed integer halfword element in **r**B. The two 32-bit signed integer products are placed into the two word elements of **r**D. The result in **r**D is also placed in the accumulator.

Other registers altered: ACC

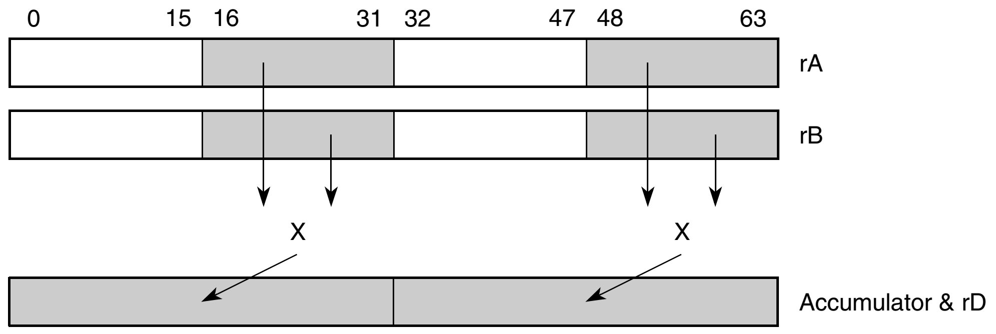


Figure 6-46. evmhosmia

evmhosmiaaw evmhosmiaaw

Vector Multiply Half Words, Odd, Signed, Modulo, Integer and Accumulate into Words

evmhosmiaaw **r**D,**r**A,**r**B (M=1, O=1, F=0, S=1)

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

```
temp1$_{0:31}$ = rA$_{16:31}$ *si rB$_{16:31}$
temp2$_{0:31}$ = rA$_{48:63}$ *si rB$_{48:63}$
temp3$_{0:32}$ = ACC$_{0:31}$ + temp1$_{0:31}$
temp4$_{0:32}$ = ACC$_{32:63}$ + temp2$_{0:31}$
ACC$_{0:31}$ = rD$_{0:31}$ = temp3$_{1:32}$
ACC$_{32:63}$ = rD$_{32:63}$ = temp4$_{1:32}$
```

For each word element in the accumulator the following operations are performed in the order shown:

Each odd-numbered signed integer halfword element in **r**A is multiplied by the corresponding signed integer halfword element in **r**B.

The intermediate 32-bit product is added to the contents of the accumulator word to form a 33-bit intermediate sum. The low 32 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word.

Other registers altered: ACC

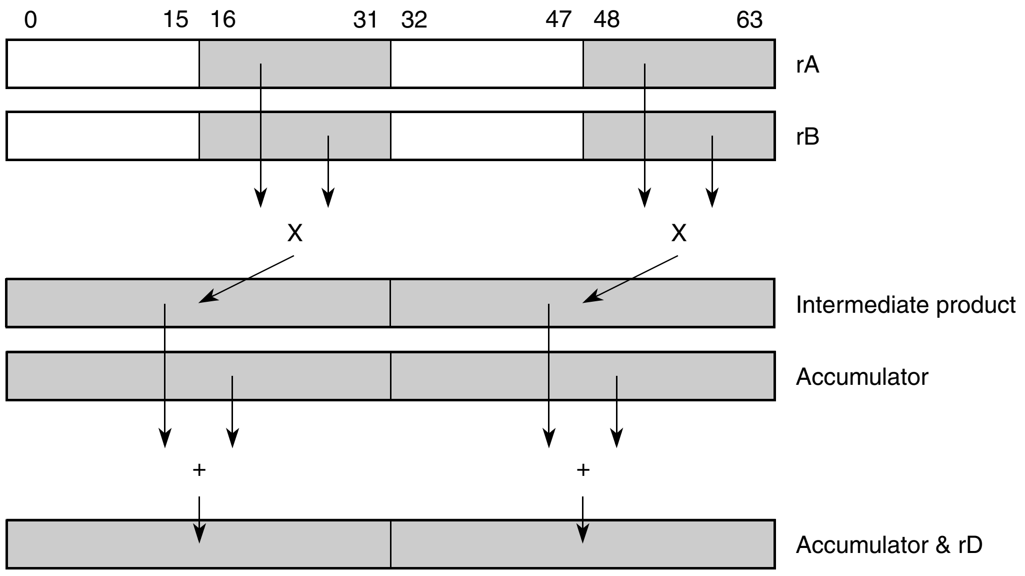


Figure 6-47. evmhosmiaaw

evmhosmianw evmhosmianw

Vector Multiply Half Words, Odd, Signed, Modulo, Integer and Accumulate Negative into Words

evmhosmianw **r**D,**r**A,**r**B (M=1, O=1, F=0, S=1)

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

```
$temp1_{0:31}$ = $rA_{16:31}$ *si $rB_{16:31}$
$temp2_{0:31}$ = $rA_{48:63}$ *si $rB_{48:63}$
$temp3_{0:32}$ = $ACC_{0:31}$ - $temp1_{0:31}$
$temp4_{0:32}$ = $ACC_{32:63}$ - $temp2_{0:31}$
$ACC_{0:31}$ = $rD_{0:31}$ = $temp3_{1:32}$
$ACC_{32:63}$ = $rD_{32:63}$ = $temp4_{1:32}$
```

For each word element in the accumulator the following operations are performed in the order shown:

Each odd-numbered signed integer halfword element in **r**A is multiplied by the corresponding signed integer halfword element in **r**B.

The intermediate 32-bit product is subtracted from the contents of the accumulator word to form a 33-bit intermediate difference. The low 32 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word.

Other registers altered: ACC

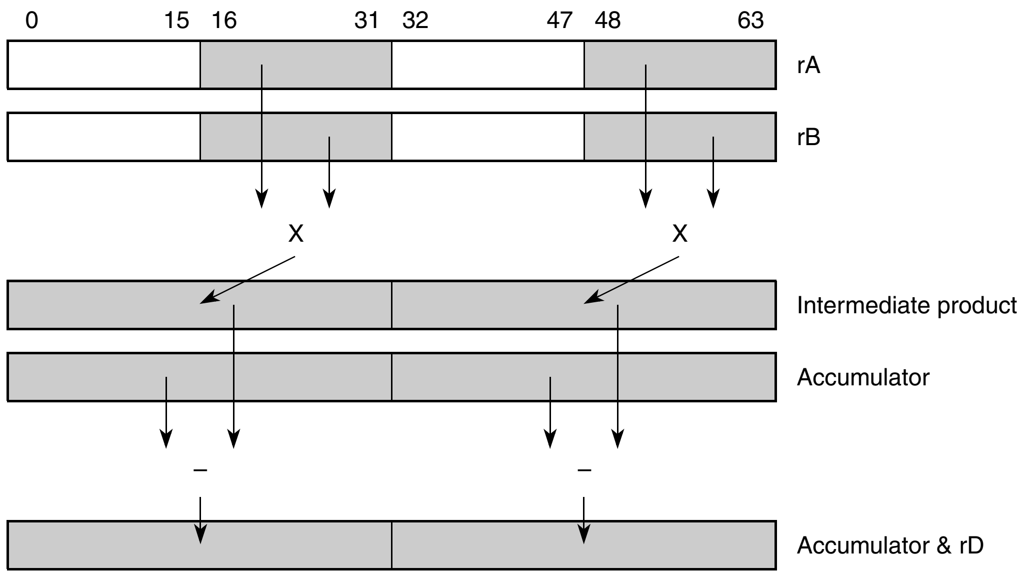


Figure 6-48. evmhosmianw

evmhossf

Vector Multiply Half Words, Odd, Signed, Saturate, Fractional

evmhossf **r**D,**r**A,**r**B (M=0, O=1, F=1, S=1, A=0)

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

```
temp1_{0:32} = (rA_{16:31} * rB_{16:31}) || 0
temp2_{0:32} = (rA_{48:63} * rB_{48:63}) || 0
movh = temp1_0 ⊕ temp1_1
movl = temp2_0 ⊕ temp2_1
rD_{0:31} = SATURATE(movh, 0x7FFFFFFF, temp1_{1:32})
rD_{32:63} = SATURATE(movl, 0x7FFFFFFF, temp2_{1:32})
SPEFSCR_{OVH} = movh
SPEFSCR_{OV} = movl
SPEFSCR_{SOVH} = SPEFSCR_{SOVH} | movh
SPEFSCR_{SOV} = SPEFSCR_{SOV} | movl
```

Each odd-numbered signed fractional halfword element in **r**A is multiplied by the corresponding signed fractional halfword element in **r**B. The two 32-bit signed fractional products are shifted left one bit to eliminate the redundant sign bit, and are then placed into the two word elements of **r**D. If the inputs are –1.0 and –1.0 the result is saturated to the most positive signed fraction (`0x7FFFFFFF`). If saturation occurs, the overflow and summary overflow bits are recorded.

Other registers altered: SPEFSCR

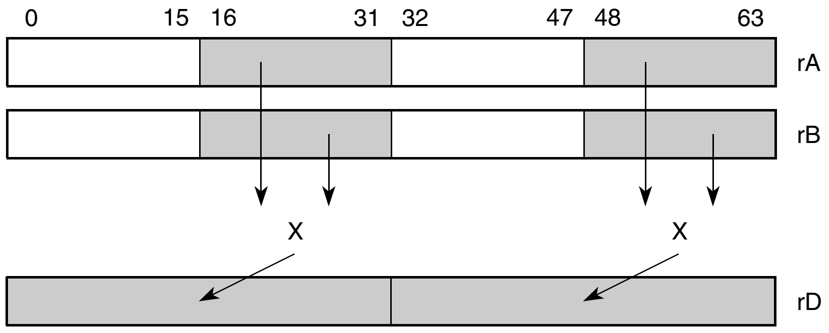


Figure 6-49. evmhossf

evmhossfa evmhossfa

Vector Multiply Half Words, Odd, Signed, Saturate, Fractional, to Accumulator

evmhossfa **r**D,**r**A,**r**B (M=0, O=1, F=1, S=1, A=1)

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

```
$temp1_{0:32}$ = ($rA_{16:31}$ * $rB_{16:31}$) || 0
$temp2_{0:32}$ = ($rA_{48:63}$ * $rB_{48:63}$) || 0
movh = $temp1_0 \oplus temp1_1$
movl = $temp2_0 \oplus temp2_1$
$rD_{0:31}$ = SATURATE(movh, 0x7FFFFFFF, $temp1_{1:32}$)
$rD_{32:63}$ = SATURATE(movl, 0x7FFFFFFF, $temp2_{1:32}$)
$ACC_{0:63}$ = $rD_{0:63}$
$SPEFSCR_{OVH}$ = movh
$SPEFSCR_{OV}$ = movl
$SPEFSCR_{SOVH}$ = $SPEFSCR_{SOVH}$ | movh
$SPEFSCR_{SOV}$ = $SPEFSCR_{SOV}$ | movl
```

Each odd-numbered signed fractional halfword element in **r**A is multiplied by the corresponding signed fractional halfword element in **r**B. The two 32-bit signed fractional products are shifted left one bit to eliminate the redundant sign bit, and are then placed into the two word elements of **r**D. If the inputs are –1.0 and –1.0 the result is saturated to the most positive signed fraction (`0x7FFFFFFF`). If saturation occurs, the overflow and summary overflow bits are recorded. The result in **r**D is also placed in the accumulator.

Other registers altered: SPEFSCR, ACC

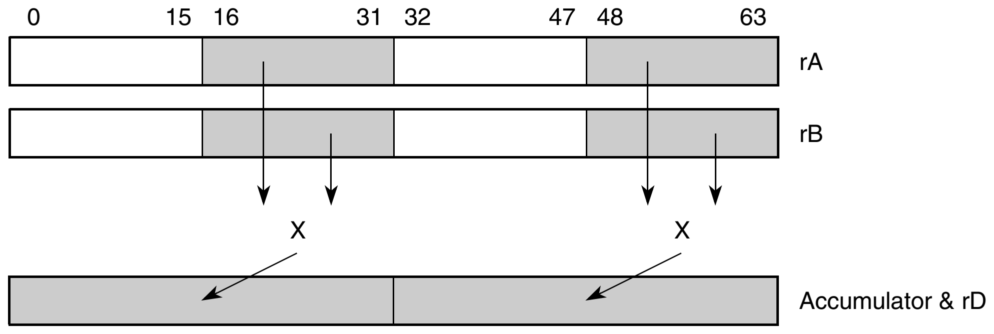


Figure 6-50. evmhossfa

evmhossfaaw

Vector Multiply Half Words, Odd, Signed, Saturate, Fractional and Accumulate into Words

evmhossfaaw **r**D,**r**A,**r**B (M=0, O=1, F=1, S=1)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	101 0000 0111

```
$temp1_{0:32}$ = ($rA_{16:31}$ * $rB_{16:31}$) || 0
$temp2_{0:32}$ = ($rA_{48:63}$ * $rB_{48:63}$) || 0
movh = $temp1_0 \oplus temp1_1$
movl = $temp2_0 \oplus temp2_1$
$temp3_{0:31}$ = SATURATE(movh, 0x7FFFFFFF, $temp1_{1:32}$)
$temp4_{0:31}$ = SATURATE(movl, 0x7FFFFFFF, $temp2_{1:32}$)
$temp5_{0:32}$ = {$ACC_0$,$ACC_{0:31}$} + {$temp3_0$,$temp3_{0:31}$}
$temp6_{0:32}$ = {$ACC_{32}$,$ACC_{32:63}$} + {$temp4_0$,$temp4_{0:31}$}
ovh = $temp5_0 \oplus temp5_1$
ovl = $temp6_0 \oplus temp6_1$
$rD_{0:31}$ = SATURATE_ACC(ovh, $temp5_0$, 0x80000000, 0x7FFFFFFF, $temp5_{1:32}$)
$rD_{32:63}$ = SATURATE_ACC(ovl, $temp6_0$, 0x80000000, 0x7FFFFFFF, $temp6_{1:32}$)
$ACC_{0:31}$ = $rD_{0:31}$
$ACC_{32:63}$ = $rD_{32:63}$
$SPEFSCR_{OVH}$ = movh | ovh
$SPEFSCR_{OV}$ = movl | ovl
$SPEFSCR_{SOVH}$ = $SPEFSCR_{SOVH}$ | movh | ovh
$SPEFSCR_{SOV}$ = $SPEFSCR_{SOV}$ | movl | ovl
```

Each odd-numbered signed fractional halfword element in **r**A is multiplied by the corresponding signed fractional halfword element in **r**B. The two 32-bit signed fractional products are shifted left one bit to eliminate the redundant sign bit. If the inputs are –1.0 and –1.0 the intermediate result is saturated to the most positive signed fraction (`0x7FFFFFFF`). The intermediate 32-bit products are added to the respective accumulator word to form an intermediate sum. If the intermediate sum has overflowed, the appropriate saturation value (`0x7FFFFFFF` if positive overflow or `0x80000000` if negative overflow) is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word.

If there is an overflow from either the multiply or the addition, the overflow information is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR, ACC

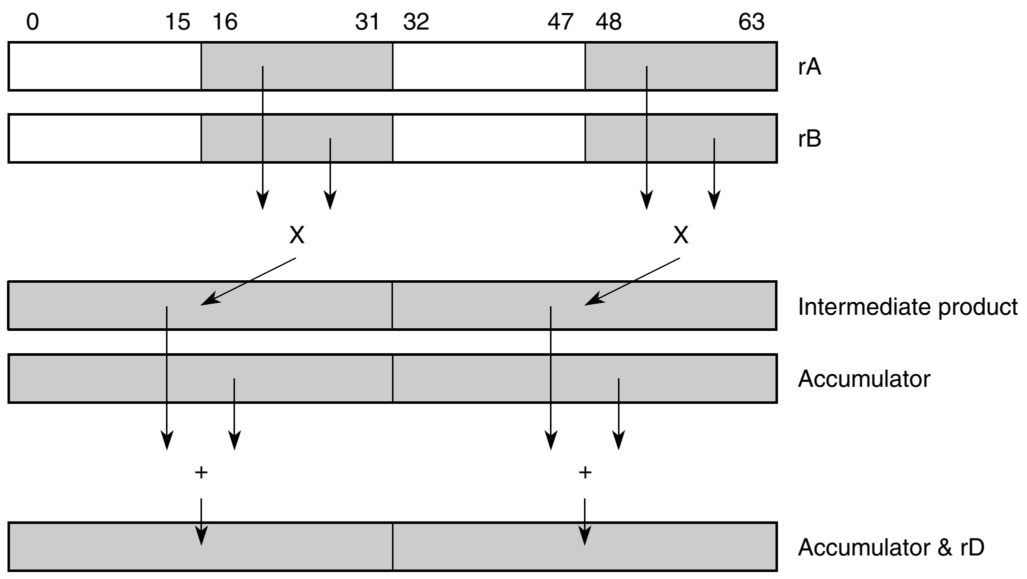


Figure 6-51. evmhossfaaw

evmhossfanw evmhossfanw

Vector Multiply Half Words, Odd, Signed, Saturate, Fractional and Accumulate Negative into Words

evmhossfanw **r**D,**r**A,**r**B (M=0, O=1, F=1, S=1)

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

```
temp1$_{0:32}$ = (rA$_{16:31}$ * rB$_{16:31}$) || 0
temp2$_{0:32}$ = (rA$_{48:63}$ * rB$_{48:63}$) || 0
movh = temp1$_0$ ⊕ temp1$_1$
movl = temp2$_0$ ⊕ temp2$_1$
temp3$_{0:31}$ = SATURATE(movh, 0x7FFFFFFF, temp1$_{1:32}$)
temp4$_{0:31}$ = SATURATE(movl, 0x7FFFFFFF, temp2$_{1:32}$)
temp5$_{0:32}$ = {ACC$_0$,ACC$_{0:31}$} - {temp3$_0$,temp3$_{0:31}$}
temp6$_{0:32}$ = {ACC$_{32}$,ACC$_{32:63}$} - {temp4$_0$,temp4$_{0:31}$}
ovh = temp5$_0$ ⊕ temp5$_1$
ovl = temp6$_0$ ⊕ temp6$_1$
rD$_{0:31}$ = SATURATE_ACC(ovh, temp5$_0$, 0x80000000, 0x7FFFFFFF, temp5$_{1:32}$)
rD$_{32:63}$ = SATURATE_ACC(ovl, temp6$_0$, 0x80000000, 0x7FFFFFFF, temp6$_{1:32}$)
ACC$_{0:31}$ = rD$_{0:31}$
ACC$_{32:63}$ = rD$_{32:63}$
SPEFSCR$_{OVH}$ = movh | ovh
SPEFSCR$_{OV}$ = movl | ovl
SPEFSCR$_{SOVH}$ = SPEFSCR$_{SOVH}$ | movh | ovh
SPEFSCR$_{SOV}$ = SPEFSCR$_{SOV}$ | movl | ovl
```

For each word element in the accumulator the following operations are performed in the order shown:

Each odd-numbered signed fractional halfword element in **r**A is multiplied by the corresponding signed fractional halfword element in **r**B. The two 32-bit signed fractional products are shifted left one bit to eliminate the redundant sign bit. If the inputs are –1.0 and –1.0 the intermediate result is saturated to the most positive signed fraction (`0x7FFFFFFF`).

The intermediate 32-bit product is subtracted from the contents of the accumulator word to form an intermediate difference. If the intermediate difference has overflowed, the appropriate saturation value (`0x7FFFFFFF` if positive overflow or `0x80000000` if negative overflow) is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word.

If there is an overflow from either the multiply or the subtraction, the overflow information is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR, ACC

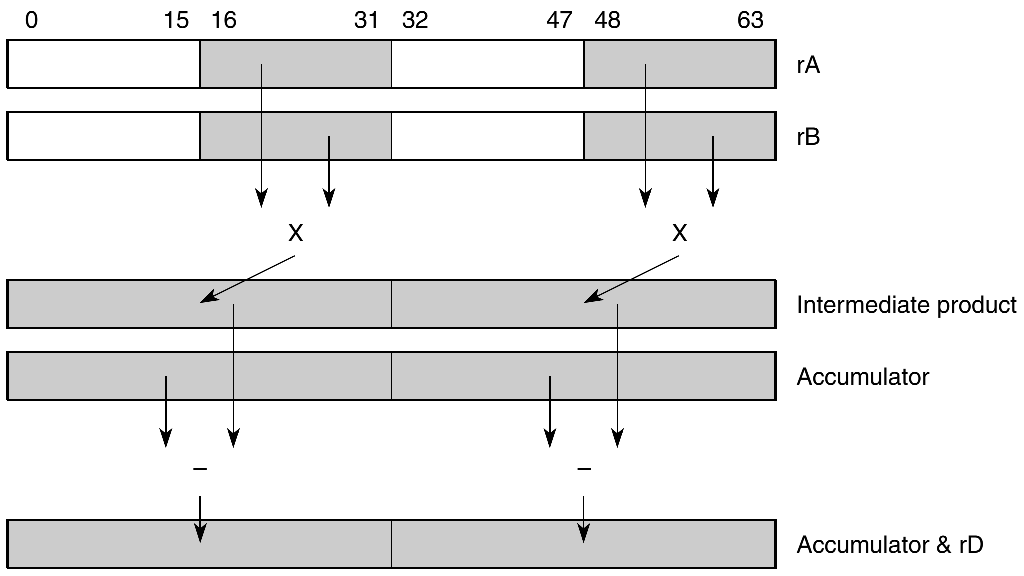


Figure 6-52. evmhossfanw

evmhossiaaw evmhossiaaw

Vector Multiply Half Words, Odd, Signed, Saturate, Integer and Accumulate into Words

evmhossiaaw **r**D,**r**A,**r**B (M=0, O=1, F=0, S=1)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	101 0000 0101

```
$temp1_{0:31}$ = $rA_{16:31}$ *si $rB_{16:31}$
$temp2_{0:31}$ = $rA_{48:63}$ *si $rB_{48:63}$
$temp3_{0:32}$ = {$ACC_0$,$ACC_{0:31}$} + {$temp1_0$,$temp1_{0:31}$}
$temp4_{0:32}$ = {$ACC_{32}$,$ACC_{32:63}$} + {$temp2_0$,$temp2_{0:31}$}
ovh = $temp3_0 \oplus temp3_1$
ovl = $temp4_0 \oplus temp4_1$
$rD_{0:31}$ = SATURATE_ACC(ovh, $temp3_0$, 0x80000000, 0x7FFFFFFF, $temp3_{1:32}$)
$rD_{32:63}$ = SATURATE_ACC(ovl, $temp4_0$, 0x80000000, 0x7FFFFFFF, $temp4_{1:32}$)
$ACC_{0:31}$ = $rD_{0:31}$
$ACC_{32:63}$ = $rD_{32:63}$
$SPEFSCR_{OVH}$ = ovh
$SPEFSCR_{OV}$ = ovl
$SPEFSCR_{SOVH}$ = $SPEFSCR_{SOVH}$ | ovh
$SPEFSCR_{SOV}$ = $SPEFSCR_{SOV}$ | ovl
```

For each word element in the accumulator the following operations are performed in the order shown:

Each odd-numbered signed integer halfword element in **r**A is multiplied by the corresponding signed integer halfword element in **r**B.

The intermediate 32-bit product is added to the contents of the accumulator word to form an intermediate sum. If the intermediate sum has overflowed, the appropriate saturation value (`0x7FFFFFFF` if positive overflow or `0x80000000` if negative overflow) is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word.

If there is an overflow from the addition, the overflow information is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR, ACC

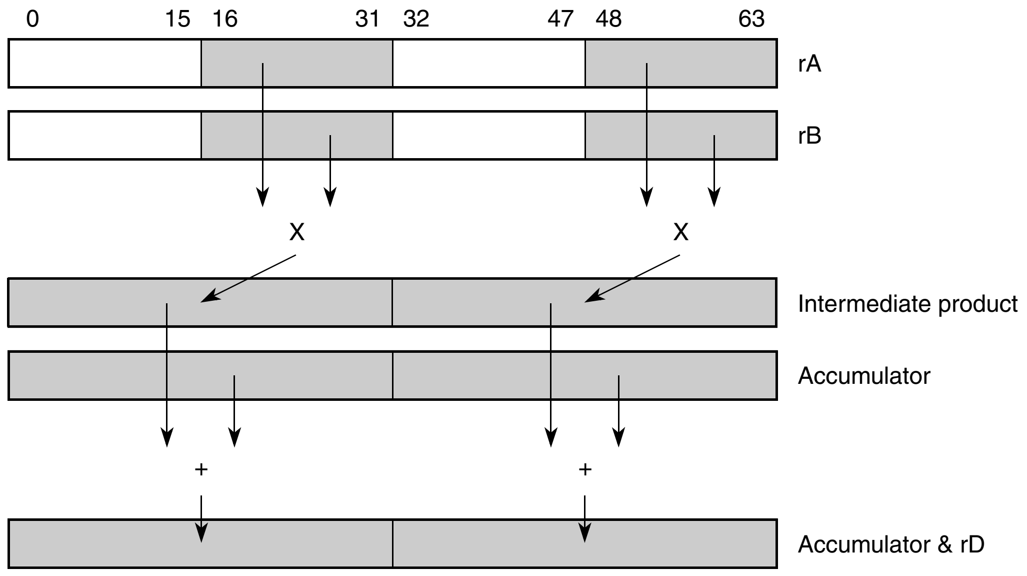


Figure 6-53. evmhossiaaw

evmhossianw evmhossianw

Vector Multiply Half Words, Odd, Signed, Saturate, Integer and Accumulate Negative into Words

evmhossianw **r**D,**r**A,**r**B (M=0, O=1, F=0, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 1000 0101

```
temp1_{0:31} = rA_{16:31} *si rB_{16:31}
temp2_{0:31} = rA_{48:63} *si rB_{48:63}
temp3_{0:32} = {ACC_0,ACC_{0:31}} - {temp1_0,temp1_{0:31}}
temp4_{0:32} = {ACC_32,ACC_{32:63}} - {temp2_0,temp2_{0:31}}
ovh = temp3_0 ⊕ temp3_1
ovl = temp4_0 ⊕ temp4_1
rD_{0:31} = SATURATE_ACC(ovh, temp3_0, 0x80000000, 0x7FFFFFFF, temp3_{1:32})
rD_{32:63} = SATURATE_ACC(ovl, temp4_0, 0x80000000, 0x7FFFFFFF, temp4_{1:32})
ACC_{0:31} = rD_{0:31}
ACC_{32:63} = rD_{32:63}
SPEFSCR_OVH = ovh
SPEFSCR_OV = ovl
SPEFSCR_SOVH = SPEFSCR_SOVH | ovh
SPEFSCR_SOV = SPEFSCR_SOV | ovl
```

For each word element in the accumulator the following operations are performed in the order shown:

Each odd-numbered signed integer halfword element in **r**A is multiplied by the corresponding signed integer halfword element in **r**B.

The intermediate 32-bit product is subtracted from the contents of the accumulator word to form an intermediate difference. If the intermediate difference has overflowed, the appropriate saturation value (`0x7FFFFFFF` if positive overflow or `0x80000000` if negative overflow) is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word.

If there is an overflow from the subtraction, the overflow information is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR, ACC

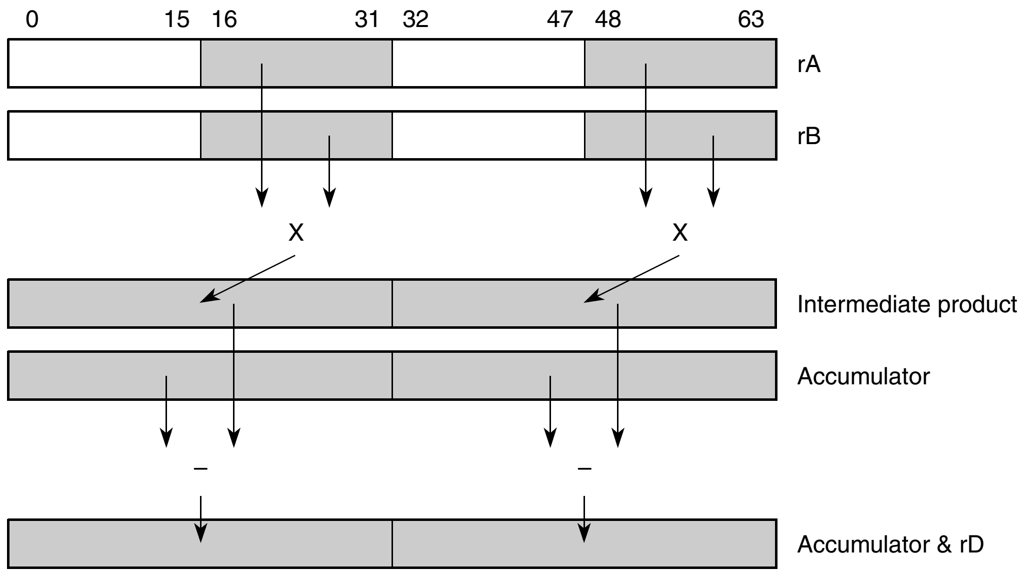


Figure 6-54. evmhossianw

evmhoumi evmhoumi

Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer

evmhoumi **r**D,**r**A,**r**B (M=1, O=1, F=0, S=0, A=0)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	100 0000 1100

$rD_{0:31}$ = $rA_{16:31}$ *ui $rB_{16:31}$
$rD_{32:63}$ = $rA_{48:63}$ *ui $rB_{48:63}$

Each odd-numbered unsigned integer halfword element in **r**A is multiplied by the corresponding unsigned integer halfword element in **r**B. The two 32-bit unsigned integer products are placed into the two word elements of **r**D.

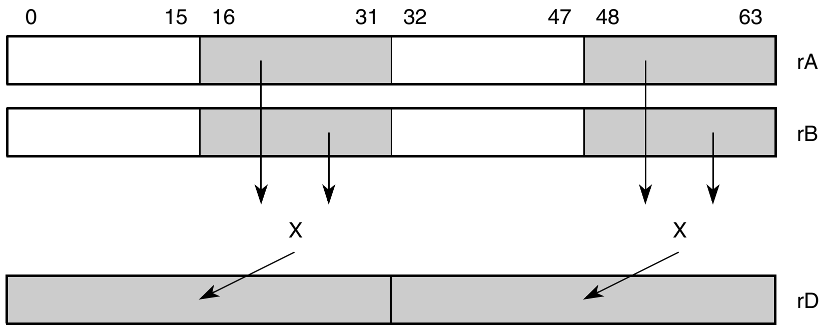


Figure 6-55. evmhoumi

evmhoumia evmhoumia

Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer, to Accumulator

evmhoumia **r**D,**r**A,**r**B (M=1, O=1, F=0, S=0, A=1)

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

```
rD0:31 = rA16:31 *ui rB16:31
rD32:63 = rA48:63 *ui rB48:63
ACC0:63 = rD0:63
```

Each odd-numbered unsigned integer halfword element in **r**A is multiplied by the corresponding unsigned integer halfword element in **r**B. The two 32-bit unsigned integer products are placed into the two word elements of **r**D. The result in **r**D is also placed in the accumulator.

Other registers altered: ACC

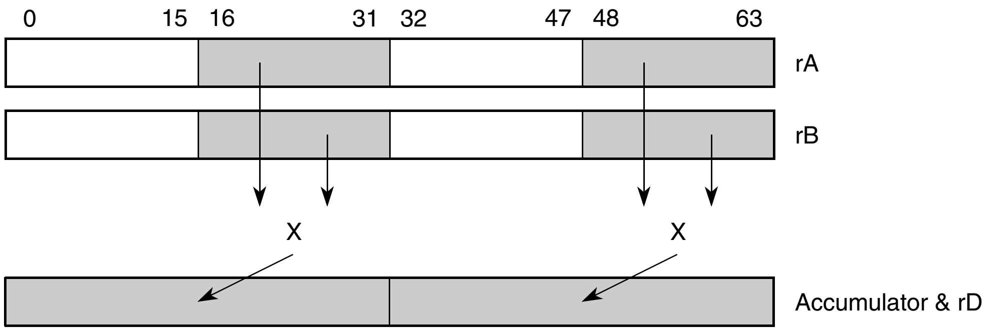


Figure 6-56. evmhoumia

evmhoumiaaw evmhoumiaaw

Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer and Accumulate into Words

evmhoumiaaw **r**D,**r**A,**r**B (M=1, O=1, F=0, S=0)

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

```
temp1_{0:31} = rA_{16:31} *ui rB_{16:31}
temp2_{0:31} = rA_{48:63} *ui rB_{48:63}
temp3_{0:32} = ACC_{0:31} + temp1_{0:31}
temp4_{0:32} = ACC_{32:63} + temp2_{0:31}
ACC_{0:31} = rD_{0:31} = temp3_{1:32}
ACC_{32:63} = rD_{32:63} = temp4_{1:32}
```

For each word element in the accumulator the following operations are performed in the order shown:

Each odd-numbered unsigned integer halfword element in **r**A is multiplied by the corresponding unsigned integer halfword element in **r**B.

The intermediate 32-bit product is added to the contents of the accumulator word to form a 33-bit intermediate sum. The low 32 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word.

Other registers altered: ACC

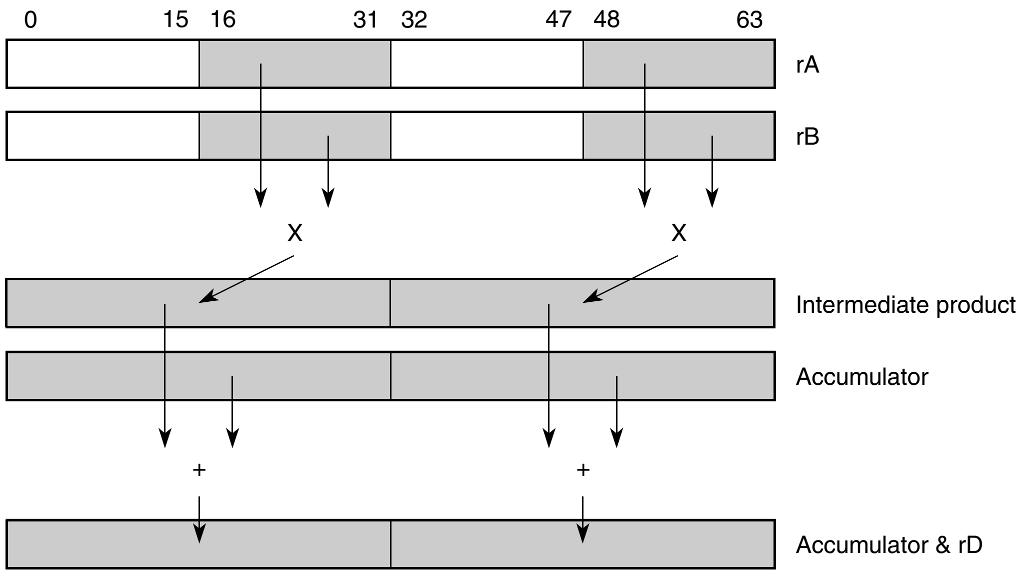


Figure 6-57. evmhoumiaaw

evmhoumianw evmhoumianw

Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer and Accumulate Negative into Words

evmhoumianw **r**D,**r**A,**r**B (M=1, O=1, F=0, S=0)

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

```
$temp1_{0:31}$ = $rA_{16:31}$ *ui $rB_{16:31}$
$temp2_{0:31}$ = $rA_{48:63}$ *ui $rB_{48:63}$
$temp3_{0:32}$ = $ACC_{0:31}$ - $temp1_{0:31}$
$temp4_{0:32}$ = $ACC_{32:63}$ - $temp2_{0:31}$
$ACC_{0:31}$ = $rD_{0:31}$ = $temp3_{1:32}$
$ACC_{32:63}$ = $rD_{32:63}$ = $temp4_{1:32}$
```

For each word element in the accumulator the following operations are performed in the order shown:

Each odd-numbered unsigned integer halfword element in **r**A is multiplied by the corresponding unsigned integer halfword element in **r**B.

The intermediate 32-bit product is subtracted from the contents of the accumulator word to form a 33-bit intermediate difference. The low 32 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word.

Other registers altered: ACC

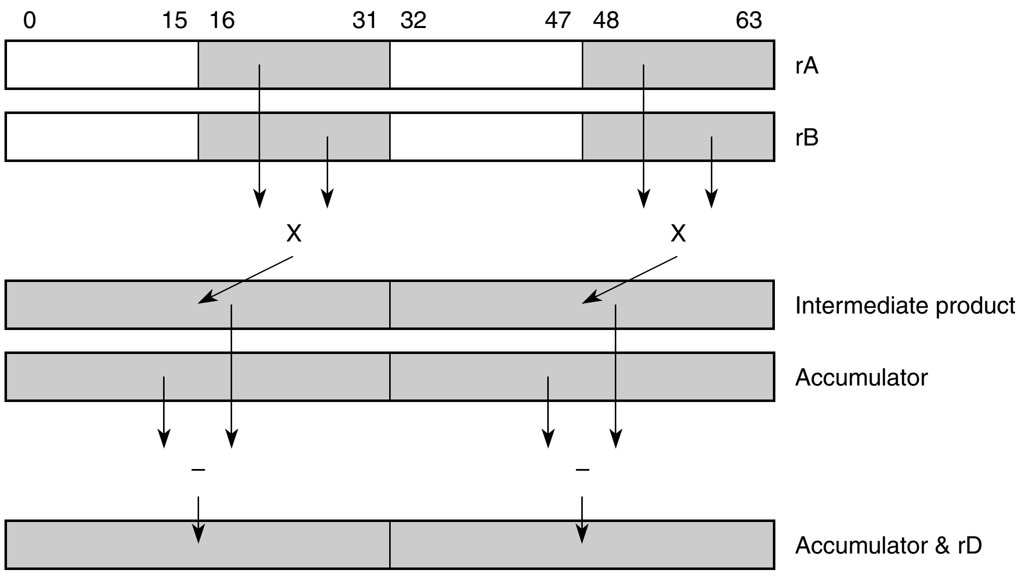


Figure 6-58. evmhoumianw

evmhousiaaw evmhousiaaw

Vector Multiply Half Words, Odd, Unsigned, Saturate, Integer and Accumulate into Words

evmhousiaaw **r**D,**r**A,**r**B (M=0, O=1, F=0, S=0)

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

```
$temp1_{0:31}$ = $rA_{16:31}$ *ui $rB_{16:31}$
$temp2_{0:31}$ = $rA_{48:63}$ *ui $rB_{48:63}$
$temp3_{0:32}$ = $ACC_{0:31}$ + $temp1_{0:31}$
$temp4_{0:32}$ = $ACC_{32:63}$ + $temp2_{0:31}$
ovh = $temp3_0$
ovl = $temp4_0$
$rD_{0:31}$ = SATURATE_ACC(ovh, 0xFFFFFFFF, $temp3_{1:32}$)
$rD_{32:63}$ = SATURATE_ACC(ovl, 0xFFFFFFFF, $temp4_{1:32}$)
$ACC_{0:31}$ = $rD_{0:31}$
$ACC_{32:63}$ = $rD_{32:63}$
$SPEFSCR_{OVH}$ = ovh
$SPEFSCR_{OV}$ = ovl
$SPEFSCR_{SOVH}$ = $SPEFSCR_{SOVH}$ | ovh
$SPEFSCR_{SOV}$ = $SPEFSCR_{SOV}$ | ovl
```

For each word element in the accumulator the following operations are performed in the order shown:

Each odd-numbered unsigned integer halfword element in **r**A is multiplied by the corresponding unsigned integer halfword element in **r**B.

The intermediate 32-bit product is added to the contents of the accumulator word to form a 33-bit intermediate sum. If the intermediate sum has overflowed, `0xFFFFFFFF` is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word.

If there is an overflow from the addition, the overflow information is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR, ACC

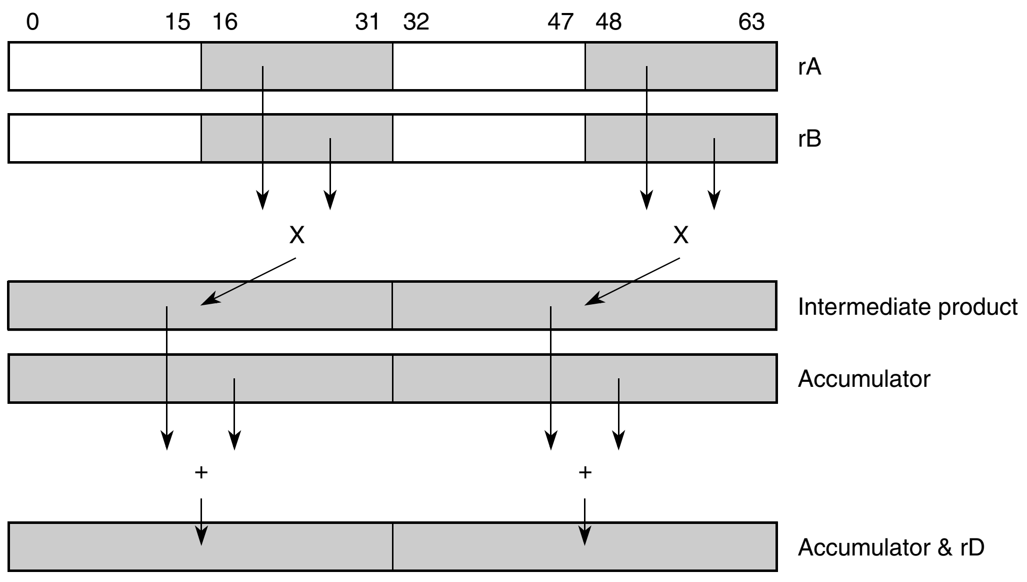


Figure 6-59. evmhousiaaw

evmhousianw

Vector Multiply Half Words, Odd, Unsigned, Saturate, Integer and Accumulate Negative into Words

evmhousianw **r**D,**r**A,**r**B (M=0, O=1, F=0, S=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 1000 0100

```
$temp1_{0:31}$ = $rA_{16:31}$ *ui $rB_{16:31}$
$temp2_{0:31}$ = $rA_{48:63}$ *ui $rB_{48:63}$
$temp3_{0:32}$ = $ACC_{0:31}$ - $temp1_{0:31}$
$temp4_{0:32}$ = $ACC_{32:63}$ - $temp2_{0:31}$
ovh = $temp3_{0}$
ovl = $temp4_{0}$
$rD_{0:31}$ = SATURATE_ACC(ovh, 0x00000000, $temp3_{1:32}$)
$rD_{32:63}$ = SATURATE_ACC(ovl, 0x00000000, $temp4_{1:32}$)
$ACC_{0:31}$ = $rD_{0:31}$
$ACC_{32:63}$ = $rD_{32:63}$
$SPEFSCR_{OVH}$ = ovh
$SPEFSCR_{OV}$ = ovl
$SPEFSCR_{SOVH}$ = $SPEFSCR_{SOVH}$ | ovh
$SPEFSCR_{SOV}$ = $SPEFSCR_{SOV}$ | ovl
```

For each word element in the accumulator the following operations are performed in the order shown:

Each odd-numbered unsigned integer halfword element in **r**A is multiplied by the corresponding unsigned integer halfword element in **r**B.

The intermediate 32-bit product is subtracted from the contents of the accumulator word to form a 33-bit intermediate difference. If the intermediate difference has underflowed (is negative), `0x00000000` is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word.

If there is an underflow from either subtraction, the underflow information is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR, ACC

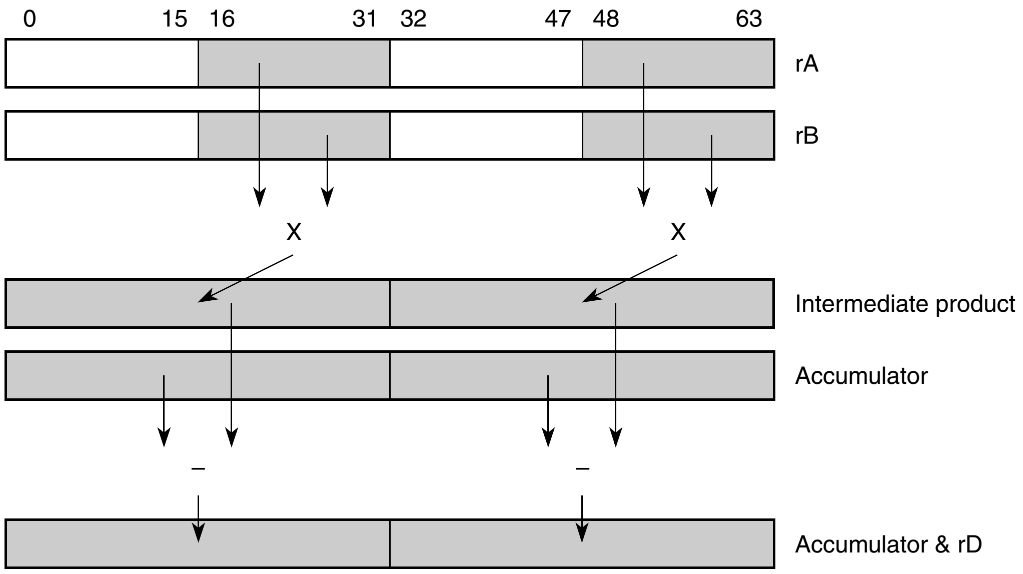


Figure 6-60. evmhousianw

6.4.2 Multiply words instructions

The following instructions perform 32x32 multiplies, returning either the higher or lower portion of the product, with and without accumulates, using signed or unsigned integer or fractional operands, with optional saturation.

evmwhsmf evmwhsmf

Vector Multiply Word High Signed, Modulo, Fractional

evmwhsmf **r**D,**r**A,**r**B (M=1, F=1, S=1,A=0)

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

```
temp1$_{0:63}$ = rA$_{0:31}$ * rB$_{0:31}$
temp2$_{0:63}$ = rA$_{32:63}$ * rB$_{32:63}$
rD$_{0:31}$ = temp1$_{1:32}$
rD$_{32:63}$ = temp2$_{1:32}$
```

Each signed fractional word element in **r**A is multiplied by the corresponding signed fractional word element in **r**B. Bits$_{1:32}$ of the two 64-bit signed fractional products (eliminating the redundant sign bit) are placed into the two word elements of **r**D.

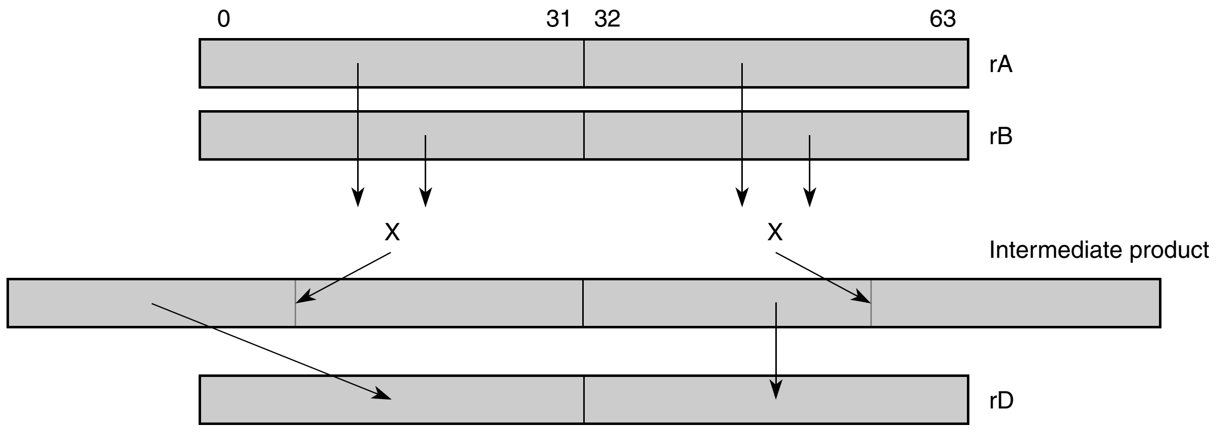


Figure 6-61. evmwhsmf

evmwhsmfa evmwhsmfa

Vector Multiply Word High Signed, Modulo, Fractional, to Accumulator

evmwhsmfa **r**D,**r**A,**r**B (M=1, F=1, S=1,A=1)

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

```
temp1_{0:64} = rA_{0:31} * rB_{0:31}
temp2_{0:64} = rA_{32:63} * rB_{32:63}
rD_{0:31} = temp1_{1:32}
rD_{32:63} = temp2_{1:32}
ACC_{0:63} = rD_{0:63}
```

Each signed fractional word element in **r**A is multiplied by the corresponding signed fractional word element in **r**B. Bits$_{1:32}$ of the two 64-bit signed fractional products (eliminating the redundant sign bit) are placed into the two word elements of **r**D. The result in **r**D is also placed in the accumulator.

Other registers altered: ACC

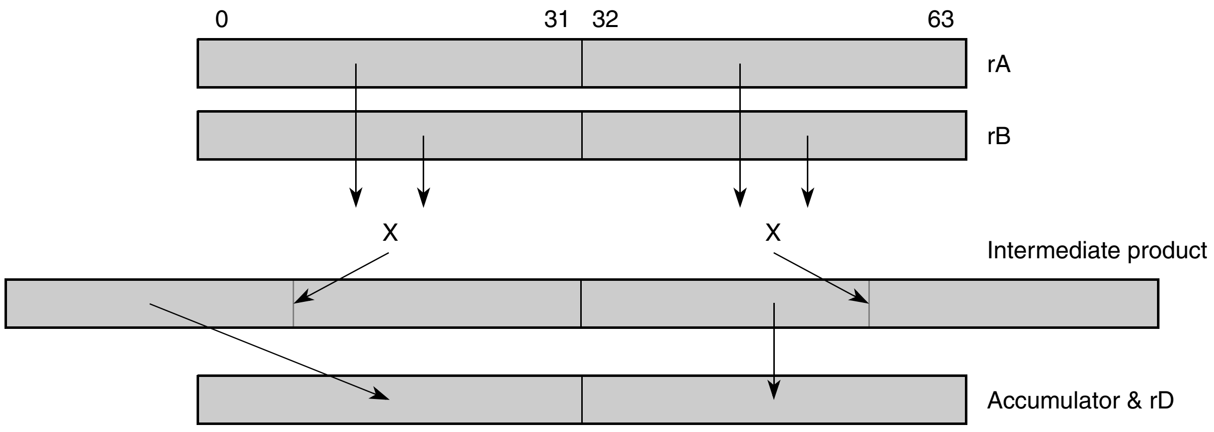


Figure 6-62. evmwhsmfa

evmwhsmi evmwhsmi

Vector Multiply Word High Signed, Modulo, Integer

evmwhsmi **r**D,**r**A,**r**B (M=1, F=0, S=1,A=0)

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

```
temp1_{0:63} = rA_{0:31} *si rB_{0:31}
temp2_{0:63} = rA_{32:63} *si rB_{32:63}
rD_{0:31} = temp1_{0:31}
rD_{32:63} = temp2_{0:31}
```

Each signed integer word element in **r**A is multiplied by the corresponding signed integer word element in **r**B. The upper 32 bits of the two 64-bit signed integer products are placed into the two word elements of **r**D.

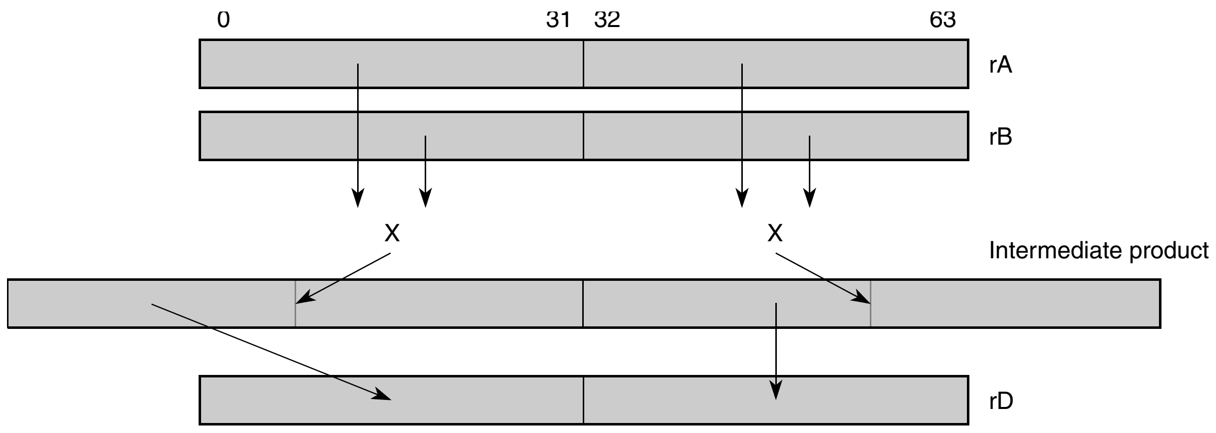


Figure 6-63. evmwhsmi

evmwhsmia   evmwhsmia

Vector Multiply Word High Signed, Modulo, Integer, to Accumulator

evmwhsmia **r**D,**r**A,**r**B (M=1, F=0, S=1,A=1)

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

```
$temp1_{0:63}$ = $rA_{0:31}$ *si $rB_{0:31}$
$temp2_{0:63}$ = $rA_{32:63}$ *si $rB_{32:63}$
$rD_{0:31}$ = $temp1_{0:31}$
$rD_{32:63}$ = $temp2_{0:31}$
$ACC_{0:63}$ = $rD_{0:63}$
```

Each signed integer word element in **r**A is multiplied by the corresponding signed integer word element in **r**B. The upper 32 bits of the two 64-bit signed integer products are placed into the two word elements of **r**D. The result in **r**D is also placed in the accumulator.

Other registers altered: ACC

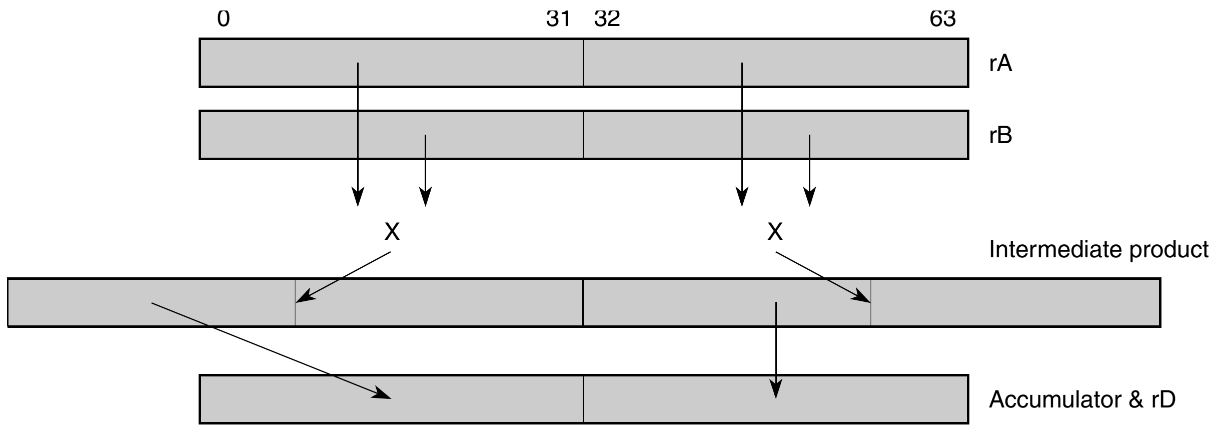


Figure 6-64. evmwhsmia

evmwhssf evmwhssf

Vector Multiply Word High Signed, Saturate, Fractional

evmwhssf **r**D,**r**A,**r**B (M=0, F=1, S=1,A=0)

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

```
$temp1_{0:63}$ = $rA_{0:31}$ * $rB_{0:31}$
$temp2_{0:63}$ = $rA_{32:63}$ * $rB_{32:63}$
movh = $temp1_0 \oplus temp1_1$
movl = $temp2_0 \oplus temp2_1$
$rD_{0:31}$ = SATURATE(movh, 0x7FFFFFFF, $temp1_{1:32}$)
$rD_{32:63}$ = SATURATE(movl, 0x7FFFFFFF, $temp2_{1:32}$)
$SPEFSCR_{OVH}$ = movh
$SPEFSCR_{OV}$ = movl
$SPEFSCR_{SOVH}$ = $SPEFSCR_{SOVH}$ | movh
$SPEFSCR_{SOV}$ = $SPEFSCR_{SOV}$ | movl
```

Each signed fractional word element in **r**A is multiplied by the corresponding signed fractional word element in **r**B. Bits$_{1:32}$ of the two 64-bit signed fractional products (eliminating the redundant sign bit) are placed into the two word elements of **r**D. If the inputs are –1.0 and –1.0 the result is saturated to the most positive signed fraction (`0x7FFFFFFF`). If saturation occurs the overflow and summary overflow bits are recorded.

Other registers altered: SPEFSCR

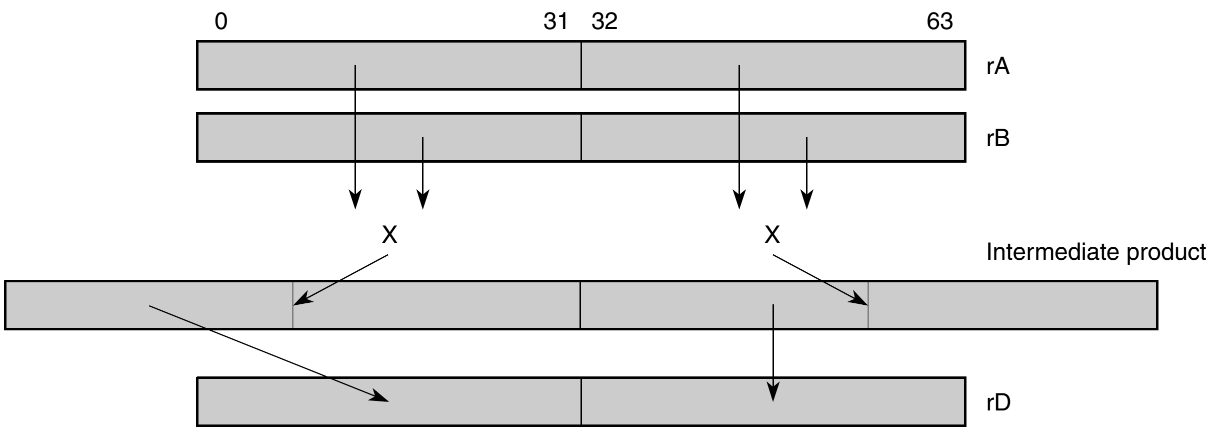


Figure 6-65. evmwhssf

evmwhssfa evmwhssfa

Vector Multiply Word High Signed, Saturate, Fractional, to Accumulator

evmwhssfa **r**D,**r**A,**r**B (M=0, F=1, S=1,A=1)

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

```
$temp1_{0:63}$ = $rA_{0:31}$ * $rB_{0:31}$
$temp2_{0:63}$ = $rA_{32:63}$ * $rB_{32:63}$
movh = $temp1_0 \oplus temp1_1$
movl = $temp2_0 \oplus temp2_1$
$rD_{0:31}$ = SATURATE(movh, 0x7FFFFFFF, $temp1_{1:32}$)
$rD_{32:63}$ = SATURATE(movl, 0x7FFFFFFF, $temp2_{1:32}$)
$ACC_{0:63}$ = $rD_{0:63}$
$SPEFSCR_{OVH}$ = movh
$SPEFSCR_{OV}$ = movl
$SPEFSCR_{SOVH}$ = $SPEFSCR_{SOVH}$ | movh
$SPEFSCR_{SOV}$ = $SPEFSCR_{SOV}$ | movl
```

Each signed fractional word element in **r**A is multiplied by the corresponding signed fractional word element in **r**B. Bits$_{1:32}$ of the two 64-bit signed fractional products (eliminating the redundant sign bit) are placed into the two word elements of **r**D. If the inputs are –1.0 and –1.0 the result is saturated to the most positive signed fraction (`0x7FFFFFFF`). If saturation occurs the overflow and summary overflow bits are recorded. The result in **r**D is also placed in the accumulator.

Other registers altered: SPEFSCR, ACC

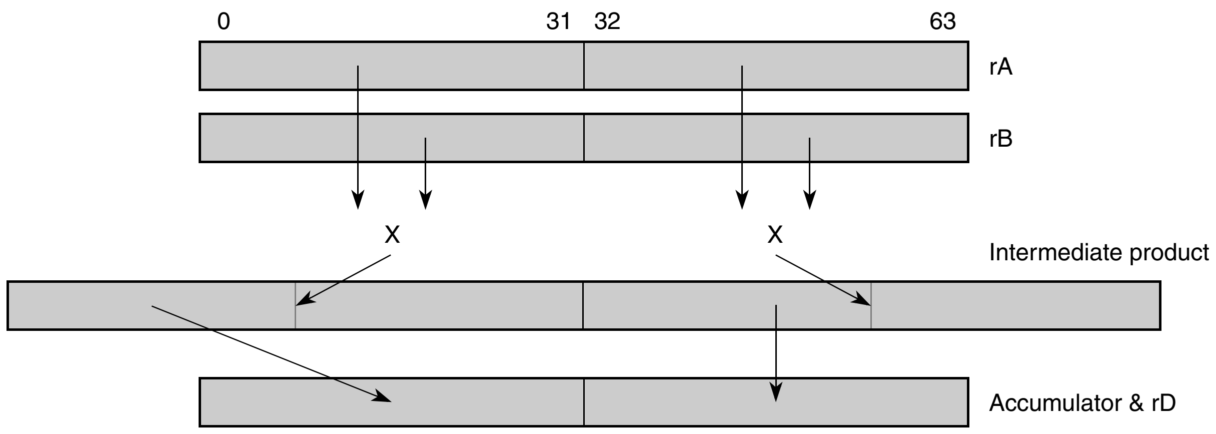


Figure 6-66. evmwhssfa

evmwhumi evmwhumi

Vector Multiply Word High Unsigned, Modulo, Integer

evmwhumi **r**D,**r**A,**r**B (M=1, F=0, S=0,A=0)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	100 0100 1100

```
temp1$_{0:63}$ = rA$_{0:31}$ *ui rB$_{0:31}$
temp2$_{0:63}$ = rA$_{32:63}$ *ui rB$_{32:63}$
rD$_{0:31}$ = temp1$_{0:31}$
rD$_{32:63}$ = temp2$_{0:31}$
```

Each unsigned integer word element in **r**A is multiplied by the corresponding unsigned integer word element in **r**B. The upper 32 bits of the two 64-bit unsigned integer products are placed into the two word elements of **r**D.

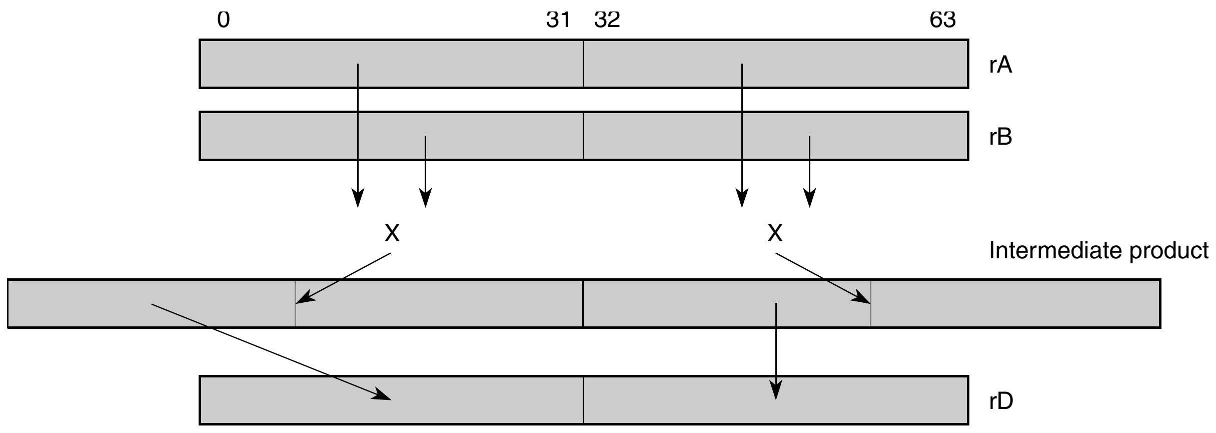


Figure 6-67. evmwhumi

evmwhumia evmwhumia

Vector Multiply Word High Unsigned, Modulo, Integer, to Accumulator

evmwhumia **r**D,**r**A,**r**B (M=1, F=0, S=0,A=1)

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

```
temp1$_{0:63}$ = rA$_{0:31}$ *ui rB$_{0:31}$
temp2$_{0:63}$ = rA$_{32:63}$ *ui rB$_{32:63}$
rD$_{0:31}$ = temp1$_{0:31}$
rD$_{32:63}$ = temp2$_{0:31}$
ACC$_{0:63}$ = rD$_{0:63}$
```

Each unsigned integer word element in **r**A is multiplied by the corresponding unsigned integer word element in **r**B. The upper 32 bits of the two 64-bit unsigned integer products are placed into the two word elements of **r**D. The result in **r**D is also placed in the accumulator.

Other registers altered: ACC

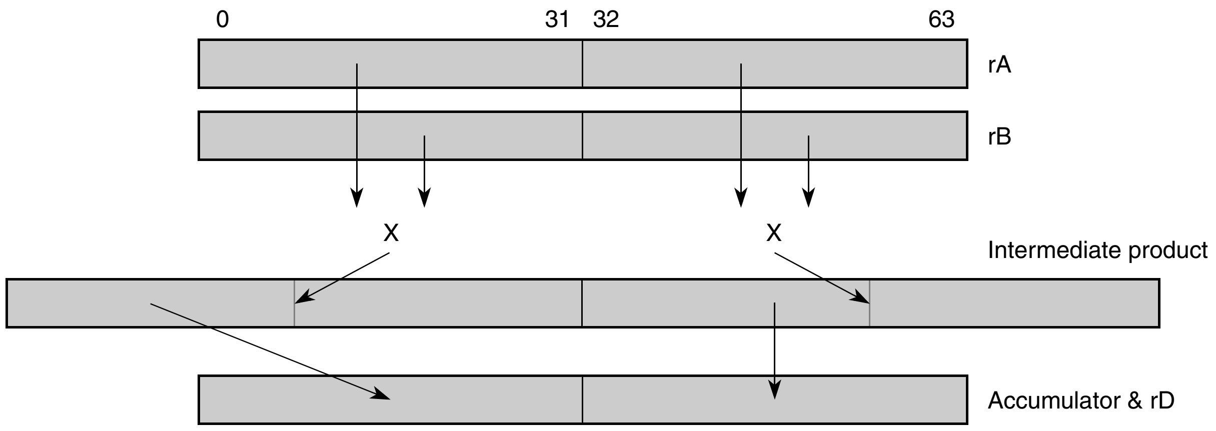


Figure 6-68. evmwhumia

evmwlsmiaaw evmwlsmiaaw

Vector Multiply Word Low Signed, Modulo, Integer and Accumulate in Words

evmwlsmiaaw **r**D,**r**A,**r**B (M=1, F=0, S=1)

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

```
temp1$_{0:63}$ = rA$_{0:31}$ *si rB$_{0:31}$
temp2$_{0:63}$ = rA$_{32:63}$ *si rB$_{32:63}$
rD$_{0:31}$ = ACC$_{0:31}$ + temp1$_{32:63}$
rD$_{32:63}$ = ACC$_{32:63}$ + temp2$_{32:63}$
ACC$_{0:63}$ = rD$_{0:63}$
```

For each word element in the accumulator the following operations are performed in the order shown:

Each signed integer word element in **r**A is multiplied by the corresponding signed integer word element in **r**B.

The low 32 bits of the 64-bit intermediate product are added to the contents of the accumulator word and placed into the corresponding **r**D word. The result in **r**D is also placed in the accumulator.

NOTE

This instruction produces a valid result only if the intermediate product can be represented in the lower 32 bits.

Other registers altered: ACC

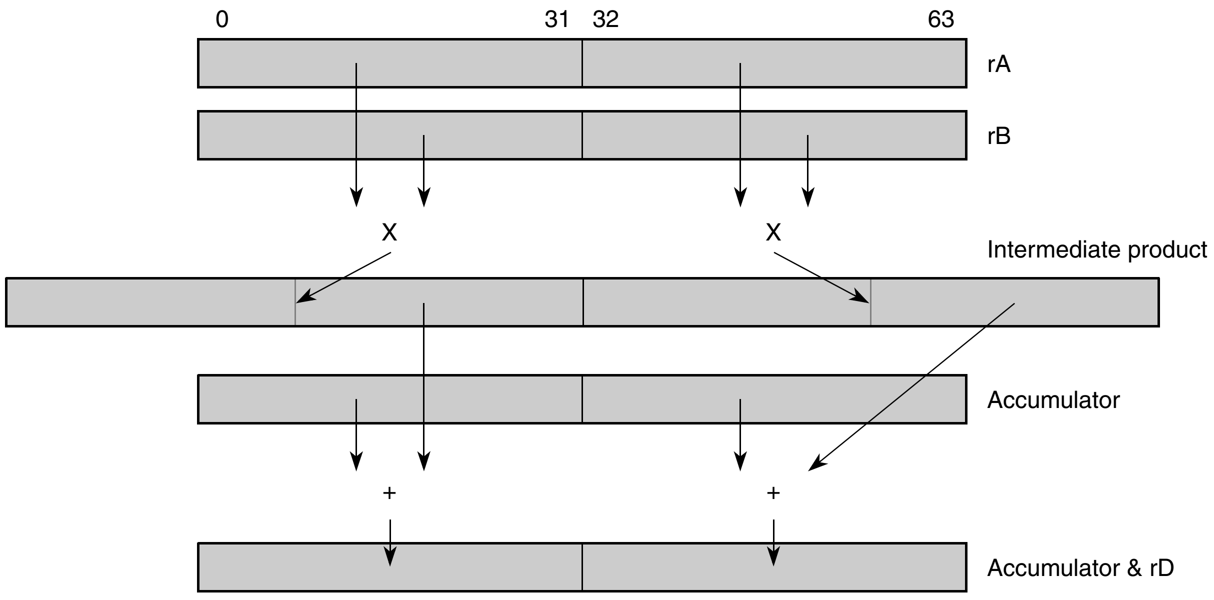


Figure 6-69. evmwlsmiaaw

evmwlsmianw evmwlsmianw

Vector Multiply Word Low Signed, Modulo, Integer and Accumulate Negative in Words

evmwlsmianw **r**D,**r**A,**r**B (M=1, F=0, S=1)

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

```
temp1$_{0:63}$ = rA$_{0:31}$ *si rB$_{0:31}$
temp2$_{0:63}$ = rA$_{32:63}$ *si rB$_{32:63}$
rD$_{0:31}$ = ACC$_{0:31}$ - temp1$_{32:63}$
rD$_{32:63}$ = ACC$_{32:63}$ - temp2$_{32:63}$
ACC$_{0:63}$ = rD$_{0:63}$
```

For each word element in the accumulator the following operations are performed in the order shown:

Each signed integer word element in **r**A is multiplied by the corresponding signed integer word element in **r**B. The low 32 bits of the 64-bit intermediate product are subtracted from the contents of the accumulator word and placed into the corresponding **r**D word. The result in **r**D is also placed in the accumulator.

NOTE

This instruction produces a valid result only if the intermediate product can be represented in the lower 32 bits.

Other registers altered: ACC

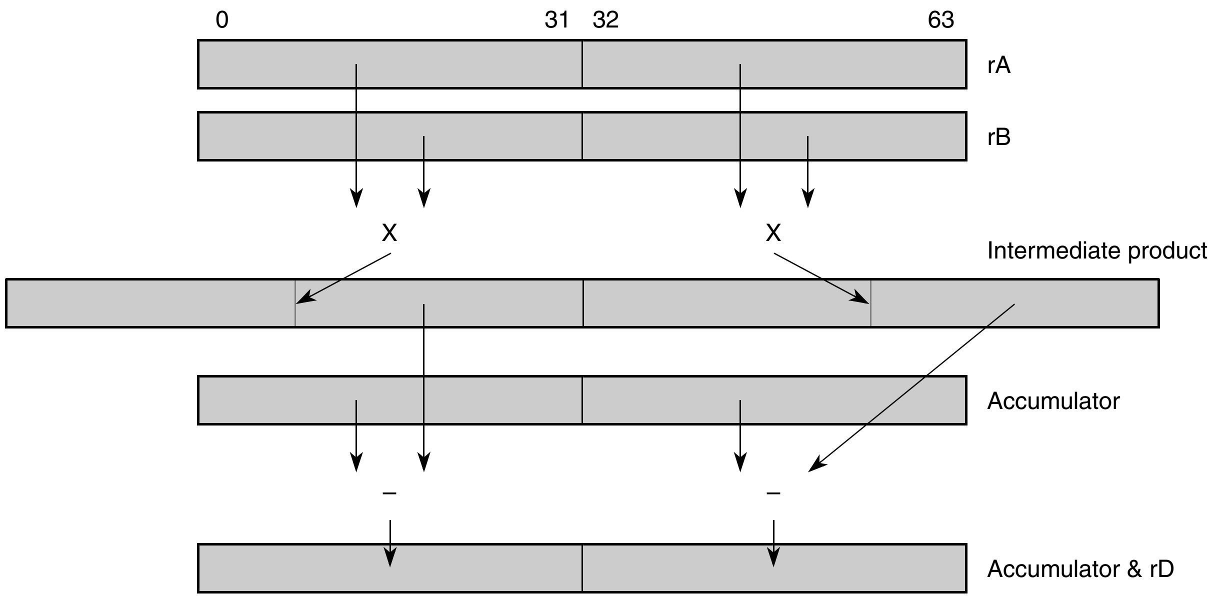


Figure 6-70. evmwlsmianw

evmwlssiaaw evmwlssiaaw

Vector Multiply Word Low Signed, Saturate, Integer and Accumulate in Words

evmwlssiaaw **r**D,**r**A,**r**B (M=0, F=0, S=1)

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

```
temp1$_{0:63}$ = rA$_{0:31}$ *si rB$_{0:31}$
temp2$_{0:63}$ = rA$_{32:63}$ *si rB$_{32:63}$
temp3$_{0:32}$ = {ACC$_{0}$,ACC$_{0:31}$} + {temp1$_{32}$,temp1$_{32:63}$}
temp4$_{0:32}$ = {ACC$_{32}$,ACC$_{32:63}$} + {temp2$_{32}$,temp2$_{32:63}$}
ovh = temp3$_{0}$ $\oplus$ temp3$_{1}$
ovl = temp4$_{0}$ $\oplus$ temp4$_{1}$
rD$_{0:31}$ = SATURATE_ACC(ovh, temp3$_{0}$, 0x80000000, 0x7FFFFFFF, temp3$_{1:32}$)
rD$_{32:63}$ = SATURATE_ACC(ovl, temp4$_{0}$, 0x80000000, 0x7FFFFFFF, temp4$_{1:32}$)
ACC$_{0:31}$ = rD$_{0:31}$
ACC$_{32:63}$ = rD$_{32:63}$
SPEFSCR$_{OVH}$ = ovh
SPEFSCR$_{OV}$ = ovl
SPEFSCR$_{SOVH}$ = SPEFSCR$_{SOVH}$ | ovh
SPEFSCR$_{SOV}$ = SPEFSCR$_{SOV}$ | ovl
```

For each word element in the accumulator the following operations are performed in the order shown:

Each signed integer word element in **r**A is multiplied by the corresponding signed integer word element in **r**B.

The low 32 bits of the 64-bit intermediate product are added to the contents of the accumulator word to form an intermediate sum. If the intermediate sum has overflowed, the appropriate saturation value (`0x7FFFFFFF` if positive overflow or `0x80000000` if negative overflow) is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word.

If there is an overflow from the addition, the overflow information is recorded in the SPEFSCR overflow and summary overflow bits.

NOTE

This instruction produces a valid result only if the intermediate product can be represented in the lower 32 bits.

Other registers altered: SPEFSCR, ACC

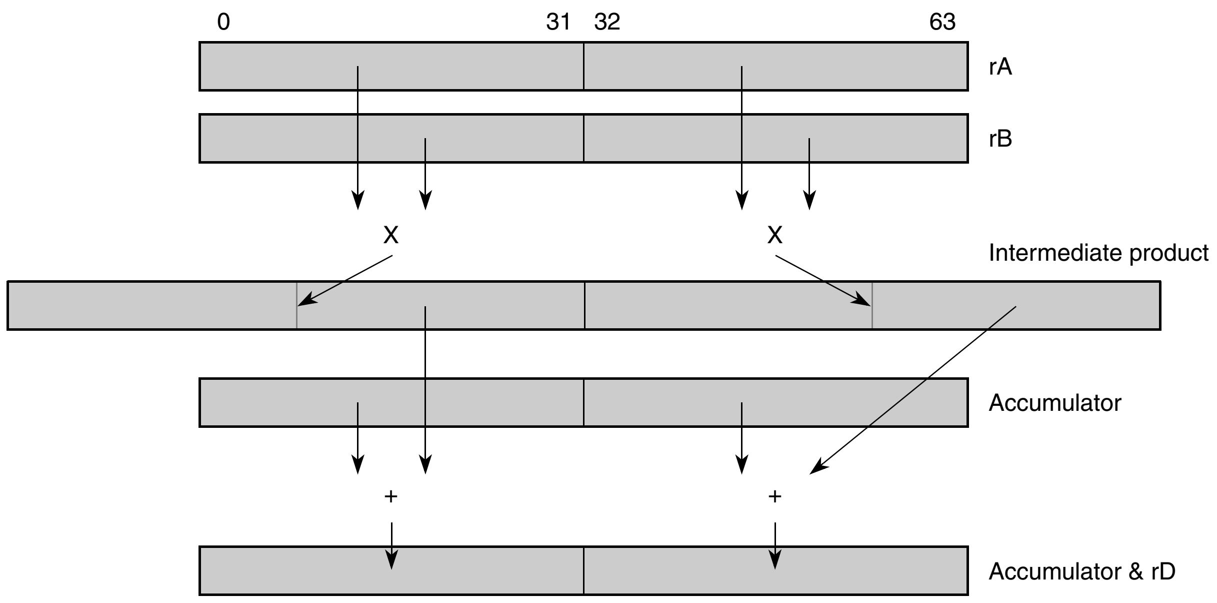


Figure 6-71. evmwlssiaaw

evmwlssianw evmwlssianw

Vector Multiply Word Low Signed, Saturate, Integer and Accumulate Negative in Words

evmwlssianw **r**D,**r**A,**r**B (M=0, F=0, S=1)

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

```
temp1$_{0:63}$ = rA$_{0:31}$ *si rB$_{0:31}$
temp2$_{0:63}$ = rA$_{32:63}$ *si rB$_{32:63}$
temp3$_{0:32}$ = {ACC$_{0}$,ACC$_{0:31}$} - {temp1$_{32}$,temp1$_{32:63}$}
temp4$_{0:32}$ = {ACC$_{32}$,ACC$_{32:63}$} - {temp2$_{32}$,temp2$_{32:63}$}
ovh = temp3$_{0}$ $\oplus$ temp3$_{1}$
ovl = temp4$_{0}$ $\oplus$ temp4$_{1}$
rD$_{0:31}$ = SATURATE_ACC(ovh, temp3$_{0}$, 0x80000000, 0x7FFFFFFF, temp3$_{1:32}$)
rD$_{32:63}$ = SATURATE_ACC(ovl, temp4$_{0}$, 0x80000000, 0x7FFFFFFF, temp4$_{1:32}$)
ACC$_{0:31}$ = rD$_{0:31}$
ACC$_{32:63}$ = rD$_{32:63}$
SPEFSCR$_{OVH}$ = ovh
SPEFSCR$_{OV}$ = ovl
SPEFSCR$_{SOVH}$ = SPEFSCR$_{SOVH}$ | ovh
SPEFSCR$_{SOV}$ = SPEFSCR$_{SOV}$ | ovl
```

For each word element in the accumulator the following operations are performed in the order shown:

Each signed integer word element in **r**A is multiplied by the corresponding signed integer word element in **r**B.

The low 32 bits of the 64-bit intermediate product are subtracted from the contents of the accumulator word to form an intermediate difference. If the intermediate difference has overflowed, the appropriate saturation value (`0x7FFFFFFF` if positive overflow or `0x80000000` if negative overflow) is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word.

If there is an overflow from the difference, the overflow information is recorded in the SPEFSCR overflow and summary overflow bits.

NOTE

This instruction produces a valid result only if the intermediate product can be represented in the lower 32 bits.

Other registers altered: SPEFSCR, ACC

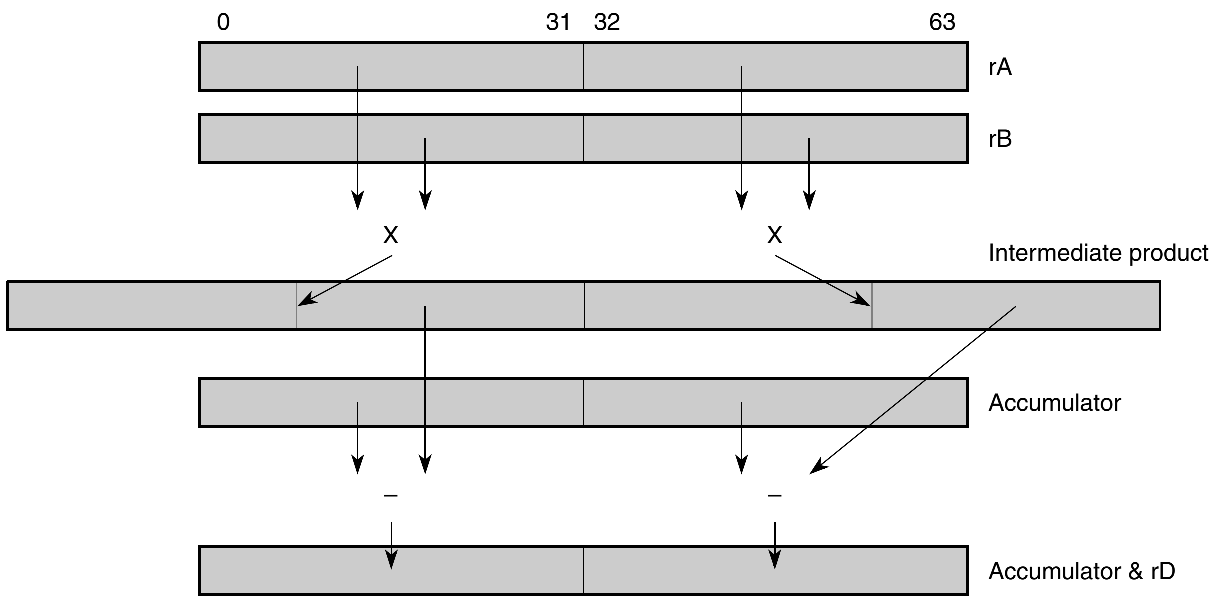


Figure 6-72. evmwlssianw

evmwlumi evmwlumi

Vector Multiply Word Low Unsigned, Modulo, Integer

evmwlumi **r**D,**r**A,**r**B (M=1, F=0, S=0,A=0)

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

```
temp1$_{0:63}$ = rA$_{0:31}$ *ui rB$_{0:31}$
temp2$_{0:63}$ = rA$_{32:63}$ *ui rB$_{32:63}$
rD$_{0:31}$ = temp1$_{32:63}$
rD$_{32:63}$ = temp2$_{32:63}$
```

Each unsigned integer word element in **r**A is multiplied by the corresponding unsigned integer word element in **r**B. The lower 32 bits of the two 64-bit unsigned integer products are placed into the two word elements of **r**D.

NOTE

The low-order 32 bits of the product are independent of whether the word elements in **r**A and **r**B are treated as signed or unsigned 32-bit integers.

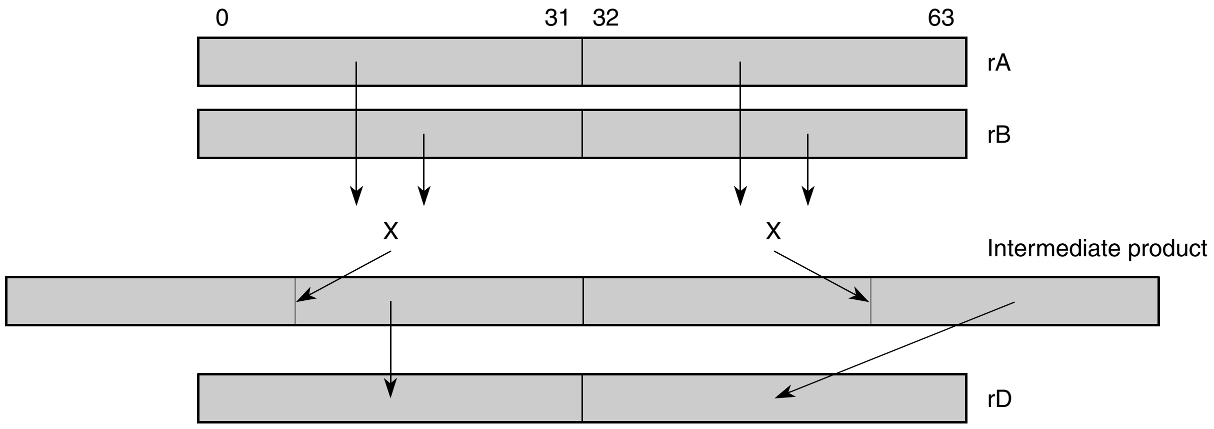


Figure 6-73. evmwlumi

evmwlumia evmwlumia

Vector Multiply Word Low Unsigned, Modulo, Integer, to Accumulator

evmwlumia **r**D,**r**A,**r**B (M=1, F=0, S=0,A=1)

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

```
temp1$_{0:63}$ = rA$_{0:31}$ *ui rB$_{0:31}$
temp2$_{0:63}$ = rA$_{32:63}$ *ui rB$_{32:63}$
rD$_{0:31}$ = temp1$_{32:63}$
rD$_{32:63}$ = temp2$_{32:63}$
ACC$_{0:63}$ = rD$_{0:63}$
```

Each unsigned integer word element in **r**A is multiplied by the corresponding unsigned integer word element in **r**B. The lower 32 bits of the two 64-bit unsigned integer products are placed into the two word elements of **r**D. The result in **r**D is also placed in the accumulator.

NOTE

The low-order 32 bits of the product are independent of whether the word elements in **r**A and **r**B are treated as signed or unsigned 32-bit integers.

Other registers altered: ACC

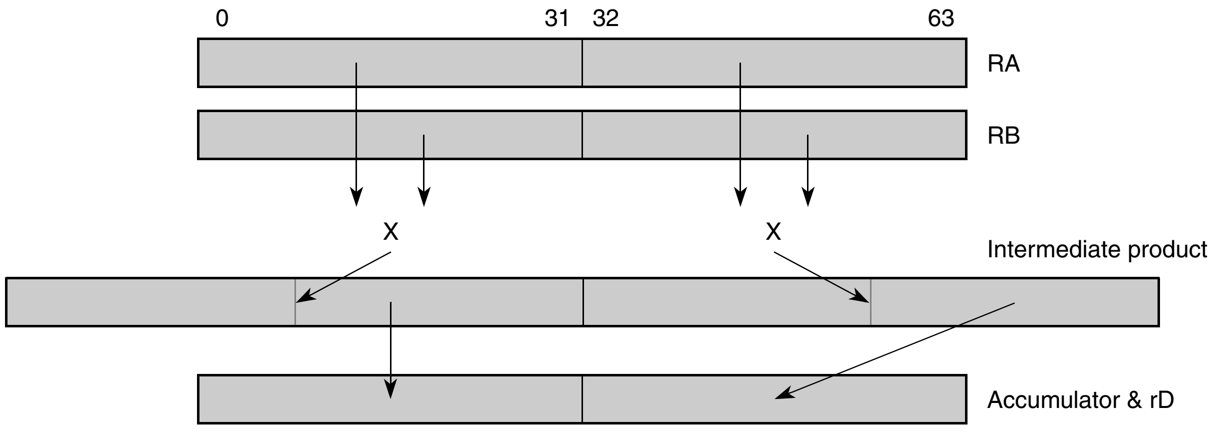


Figure 6-74. evmwlumia

evmwlumiaaw evmwlumiaaw

Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate in Words

evmwlumiaaw **r**D,**r**A,**r**B (M=1, F=0, S=0)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	101 0100 1000

```
$temp1_{0:63}$ = $rA_{0:31}$ *ui $rB_{0:31}$
$temp2_{0:63}$ = $rA_{32:63}$ *ui $rB_{32:63}$
$rD_{0:31}$ = $ACC_{0:31}$ + $temp1_{32:63}$
$rD_{32:63}$ = $ACC_{32:63}$ + $temp2_{32:63}$
$ACC_{0:63}$ = $rD_{0:63}$
```

For each word element in the accumulator the following operations are performed in the order shown:

Each unsigned integer word element in **r**A is multiplied by the corresponding unsigned integer word element in **r**B.

The low 32 bits of the 64-bit intermediate product are added to the contents of the accumulator word and placed into the corresponding **r**D word. The result in **r**D is also placed in the accumulator.

NOTE

This instruction produces a valid result only if the intermediate product can be represented in the lower 32 bits.

Other registers altered: ACC

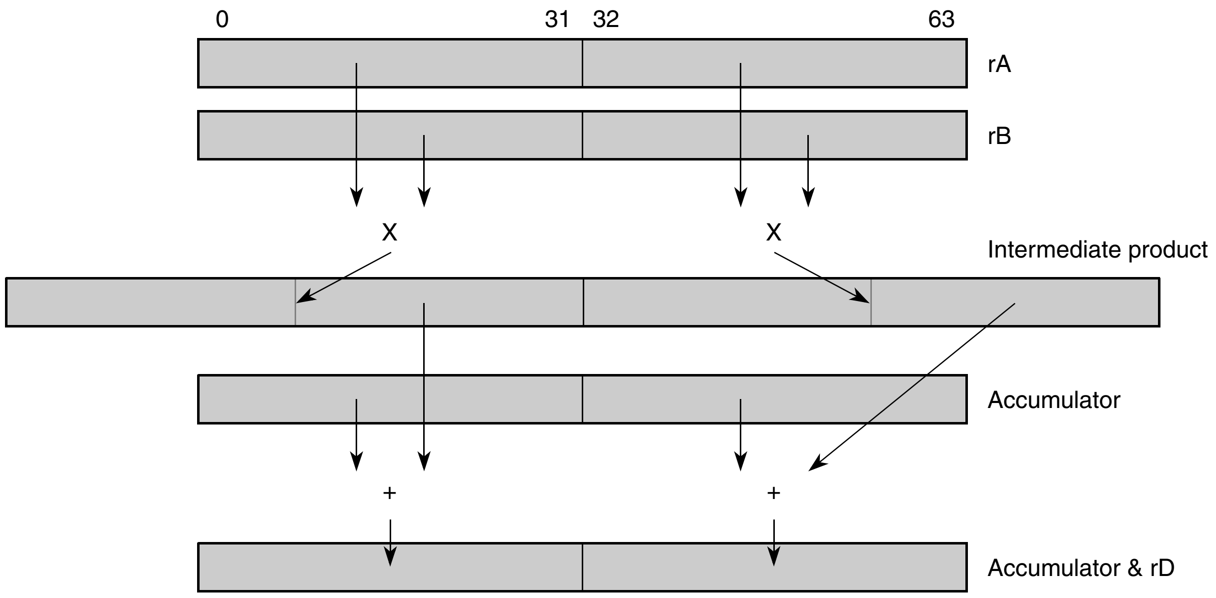


Figure 6-75. evmwlumiaaw

evmwlumianw evmwlumianw

Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate Negative in Words

evmwlumianw **r**D,**r**A,**r**B (M=1, F=0, S=0)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	101 1100 1000

```
$temp1_{0:63}$ = $rA_{0:31}$ *ui $rB_{0:31}$
$temp2_{0:63}$ = $rA_{32:63}$ *ui $rB_{32:63}$
$rD_{0:31}$ = $ACC_{0:31}$ - $temp1_{32:63}$
$rD_{32:63}$ = $ACC_{32:63}$ - $temp2_{32:63}$
$ACC_{0:63}$ = $rD_{0:63}$
```

For each word element in the accumulator the following operations are performed in the order shown:

Each unsigned integer word element in **r**A is multiplied by the corresponding unsigned integer word element in **r**B.

The low 32 bits of the 64-bit intermediate product are subtracted from the contents of the accumulator word and placed into the corresponding **r**D word. The result in **r**D is also placed in the accumulator.

NOTE

This instruction produces a valid result only if the intermediate product can be represented in the lower 32 bits.

Other registers altered: ACC

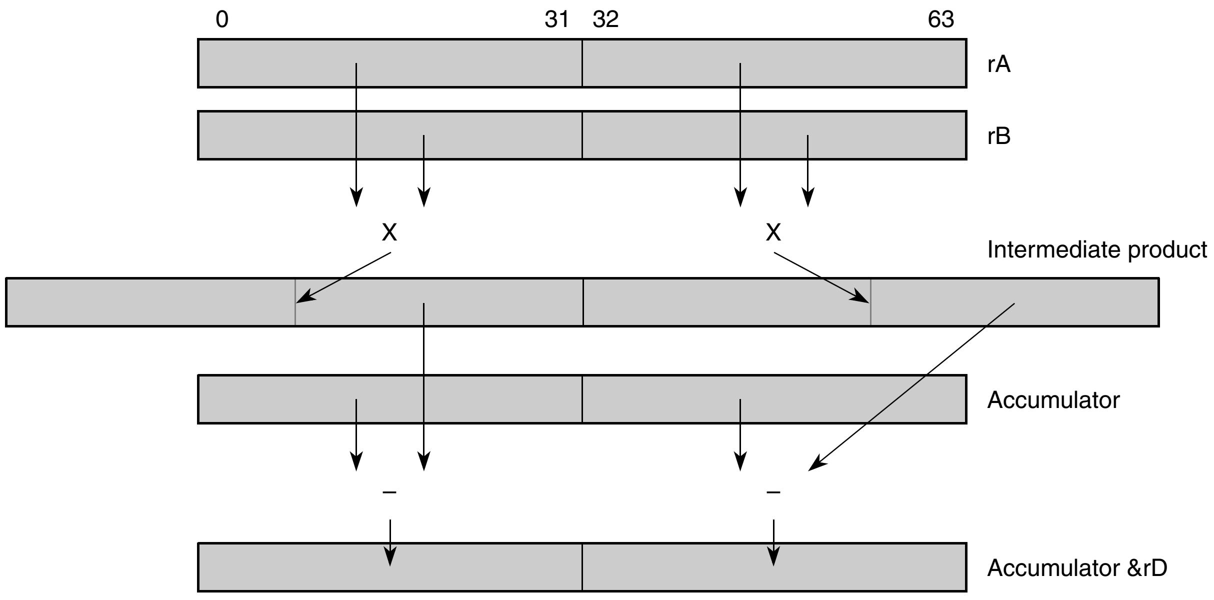


Figure 6-76. evmwlumianw

evmwlusiaaw

Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate in Words

evmwlusiaaw **r**D,**r**A,**r**B (M=0, F=0, S=0)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	101 0100 0000

```
temp1$_{0:63}$ = rA$_{0:31}$ *ui rB$_{0:31}$
temp2$_{0:63}$ = rA$_{32:63}$ *ui rB$_{32:63}$
temp3$_{0:32}$ = ACC$_{0:31}$ + temp1$_{32:63}$
temp4$_{0:32}$ = ACC$_{32:63}$ + temp2$_{32:63}$
ovh = temp3$_{0}$
ovl = temp4$_{0}$
rD$_{0:31}$ = SATURATE_ACC(ovh, 0xFFFFFFFF, temp3$_{1:32}$)
rD$_{32:63}$ = SATURATE_ACC(ovl, 0xFFFFFFFF, temp4$_{1:32}$)
ACC$_{0:31}$ = rD$_{0:31}$
ACC$_{32:63}$ = rD$_{32:63}$
SPEFSCR$_{OVH}$ = ovh
SPEFSCR$_{OV}$ = ovl
SPEFSCR$_{SOVH}$ = SPEFSCR$_{SOVH}$ | ovh
SPEFSCR$_{SOV}$ = SPEFSCR$_{SOV}$ | ovl
```

For each word element in the accumulator the following operations are performed in the order shown:

Each unsigned integer word element in **r**A is multiplied by the corresponding unsigned integer word element in **r**B.

The low 32 bits of the 64-bit intermediate product are added to the contents of the accumulator word to form a 33-bit intermediate sum. If the intermediate sum has overflowed, `0xFFFFFFFF` is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word.

If there is an overflow from the addition, the overflow information is recorded in the SPEFSCR overflow and summary overflow bits.

NOTE

This instruction produces a valid result only if the intermediate product can be represented in the lower 32 bits.

Other registers altered: SPEFSCR, ACC

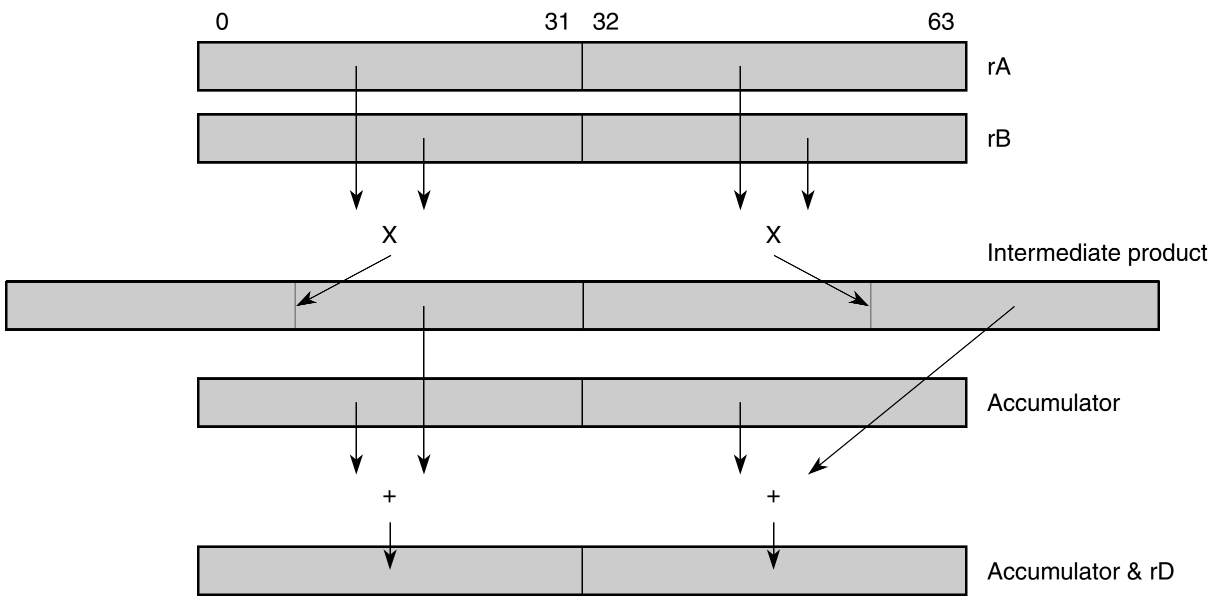


Figure 6-77. evmwlusiaaw

evmwlusianw evmwlusianw

Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate Negative in Words

evmwlusianw **r**D,**r**A,**r**B (M=0, F=0, S=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 1100 0000

```
temp1_{0:63} = rA_{0:31} *ui rB_{0:31}
temp2_{0:63} = rA_{32:63} *ui rB_{32:63}
temp3_{0:32} = ACC_{0:31} - temp1_{32:63}
temp4_{0:32} = ACC_{32:63} - temp2_{32:63}
ovh = temp3_0
ovl = temp4_0
rD_{0:31} = SATURATE_ACC(ovh, 0x00000000, temp3_{1:32})
rD_{32:63} = SATURATE_ACC(ovl, 0x00000000, temp4_{1:32})
ACC_{0:31} = rD_{0:31}
ACC_{32:63} = rD_{32:63}
SPEFSCR_{OVH} = ovh
SPEFSCR_{OV} = ovl
SPEFSCR_{SOVH} = SPEFSCR_{SOVH} | ovh
SPEFSCR_{SOV} = SPEFSCR_{SOV} | ovl
```

For each word element in the accumulator the following operations are performed in the order shown:

Each unsigned integer word element in **r**A is multiplied by the corresponding unsigned integer word element in **r**B.

The low 32 bits of the 64-bit intermediate product are subtracted from the contents of the accumulator word to form a 33-bit intermediate difference. If the intermediate difference has underflowed, `0x00000000` is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word.

If there is an underflow from the difference, the underflow information is recorded in the SPEFSCR overflow and summary overflow bits.

NOTE

This instruction produces a valid result only if the intermediate product can be represented in the lower 32 bits.

Other registers altered: SPEFSCR, ACC

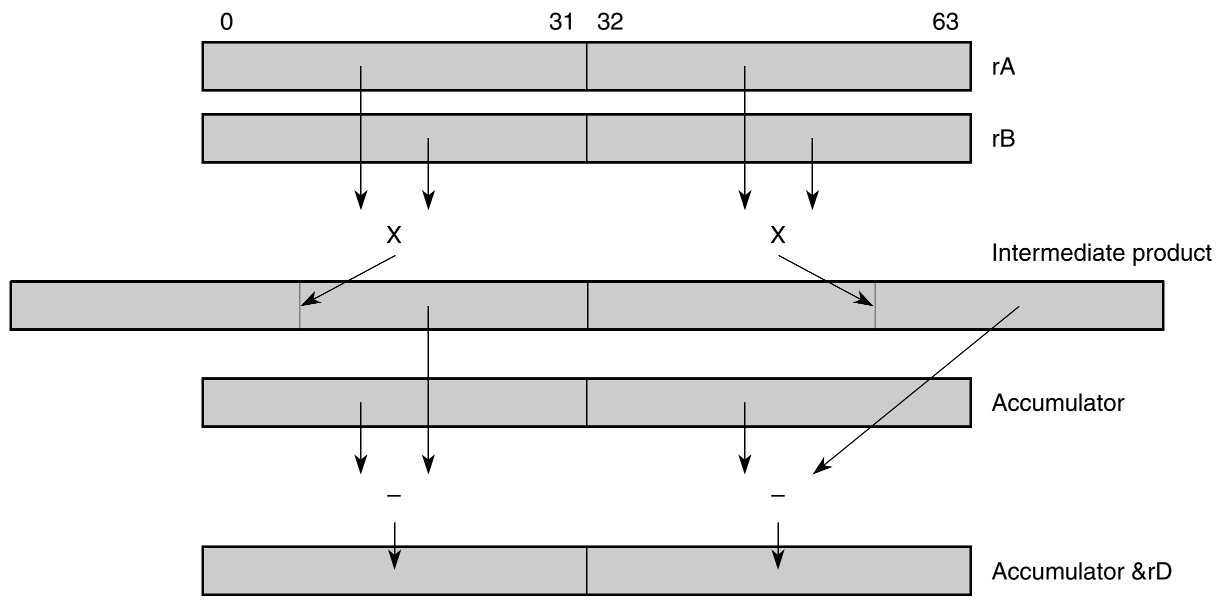


Figure 6-78. evmwlusianw

evmwsmf evmwsmf

Vector Multiply Word Signed, Modulo, Fractional

evmwsmf **r**D,**r**A,**r**B (M=1, F=1, S=1, A=0)

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

```
$temp1_{0:64}$ = ($rA_{32:63}$ * $rB_{32:63}$) || 0
$rD_{0:63}$ = $temp1_{1:64}$
```

The low signed fractional word element in **r**A is multiplied by the corresponding low signed fractional word element in **r**B. Bits 1:63 of the 64-bit signed fractional product are padded on the right with a ‘0’, and this result is placed in **r**D.

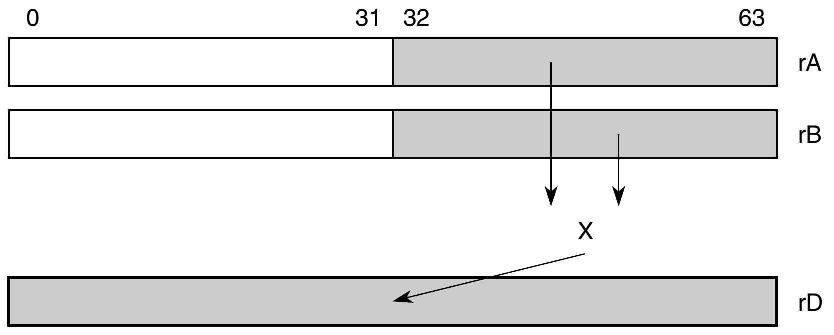


Figure 6-79. evmwsmf

evmwsmfa evmwsmfa

Vector Multiply Word Signed, Modulo, Fractional, to Accumulator

evmwsmfa **r**D,**r**A,**r**B (M=1, F=1, S=1, A=1)

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

```
$temp1_{0:64}$ = ($rA_{32:63}$ * $rB_{32:63}$) || 0
$ACC_{0:63}$ = $rD_{0:63}$ = $temp1_{1:64}$
```

The low signed fractional word element in **r**A is multiplied by the corresponding low signed fractional word element in **r**B. Bits 1:63 of the 64-bit signed fractional product are padded on the right with a ‘0’, and this result is placed in **r**D. The result in **r**D is also placed in the accumulator.

Other registers altered: ACC

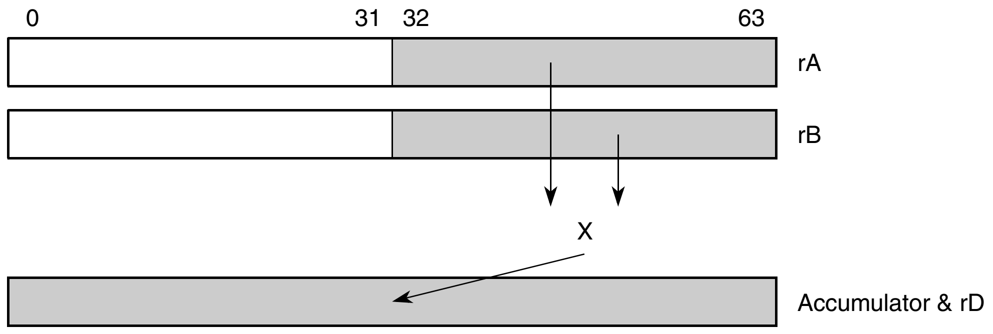


Figure 6-80. evmwsmfa

evmwsmfaa evmwsmfaa

Vector Multiply Word Signed, Modulo, Fractional and Accumulate

evmwsmfaa **r**D,**r**A,**r**B (M=1, F=1, S=1)

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

```
$temp1_{0:64}$ = ($rA_{32:63}$ * $rB_{32:63}$) || 0
$temp2_{0:64}$ = $ACC_{0:63}$ + $temp1_{1:64}$
$ACC_{0:63}$ = $rD_{0:63}$ = $temp2_{1:64}$
```

The low signed fractional word element in **r**A is multiplied by the corresponding low signed fractional word element in **r**B. Bits 1:63 of the 64-bit signed fractional product are padded on the right with a ‘0’, and this result is added to the contents of the 64-bit accumulator to form a 65-bit intermediate sum. The lower 64 bits of the intermediate sum is placed back into the accumulator and also written into **r**D.

Other registers altered: ACC

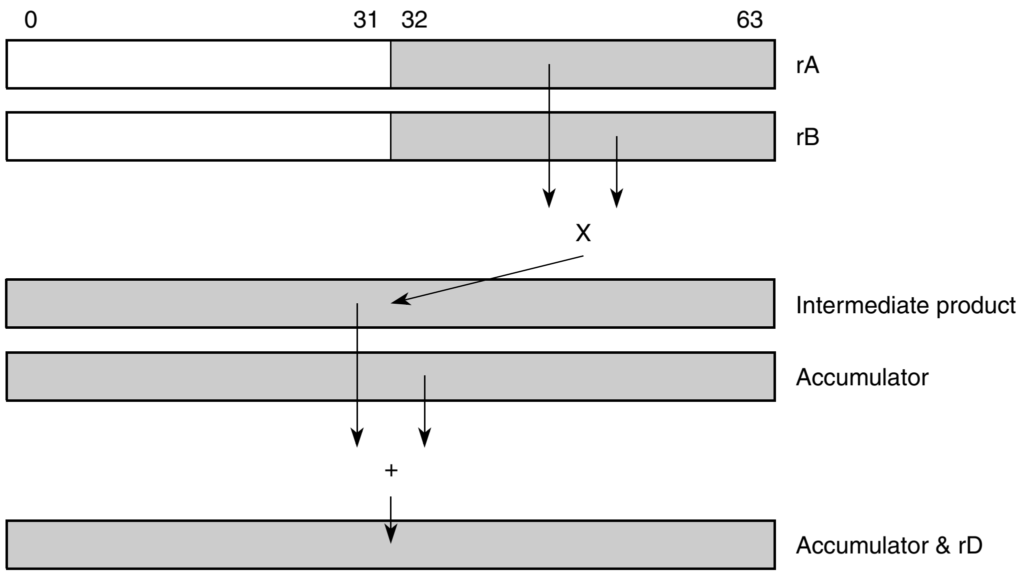


Figure 6-81. evmwsmfaa

evmwsmfan evmwsmfan

Vector Multiply Word Signed, Modulo, Fractional and Accumulate Negative

evmwsmfan **r**D,**r**A,**r**B (M=1, F=1, S=1)

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

```
temp1$_{0:64}$ = (rA$_{32:63}$ * rB$_{32:63}$) || 0
temp2$_{0:64}$ = ACC$_{0:63}$ - temp1$_{1:64}$
ACC$_{0:63}$ = rD$_{0:63}$ = temp2$_{1:64}$
```

The low signed fractional word element in **r**A is multiplied by the corresponding low signed fractional word element in **r**B. Bits 1:63 of the 64-bit signed fractional product are padded on the right with a ‘0’, and this result is subtracted from the contents of the 64-bit accumulator to form a 65-bit intermediate difference. The lower 64 bits of the intermediate difference is placed back into the accumulator and also written into **r**D.

Other registers altered: ACC

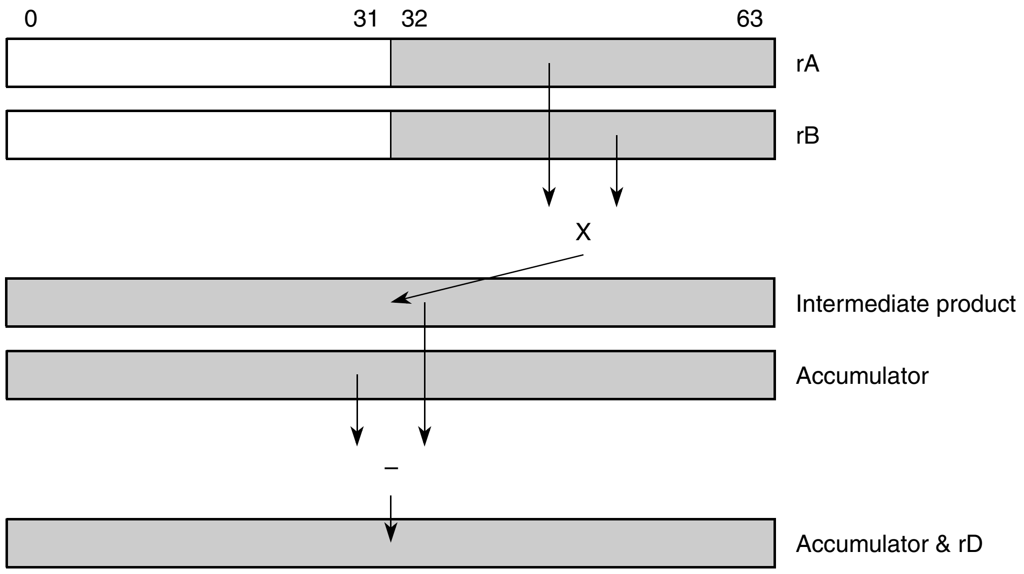


Figure 6-82. evmwsmfan

evmwsmi evmwsmi

Vector Multiply Word Signed, Modulo, Integer

evmwsmi **r**D,**r**A,**r**B (M=1, F=0, S=1, A=0)

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

```
temp$_{0:63}$ = rA$_{32:63}$ *si rB$_{32:63}$
ACC$_{0:63}$ = rD$_{0:63}$ = temp$_{0:63}$
```

The low signed integer word element in **r**A is multiplied by the corresponding low signed integer word element in **r**B. The 64-bit signed integer product is placed in **r**D.

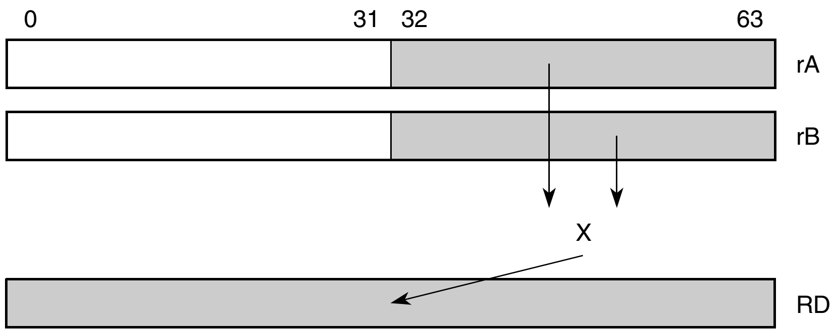


Figure 6-83. evmwsmi

evmwsmia evmwsmia

Vector Multiply Word Signed, Modulo, Integer, to Accumulator

evmwsmia **r**D,**r**A,**r**B (M=1, F=0, S=1, A=1)

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

```
temp0:63 = rA32:63 *si rB32:63
ACC0:63 = rD0:63 = temp0:63
```

The low signed integer word element in **r**A is multiplied by the corresponding low signed integer word element in **r**B. The 64-bit signed integer product is placed in **r**D. The result in **r**D is also placed in the accumulator.

Other registers altered: ACC

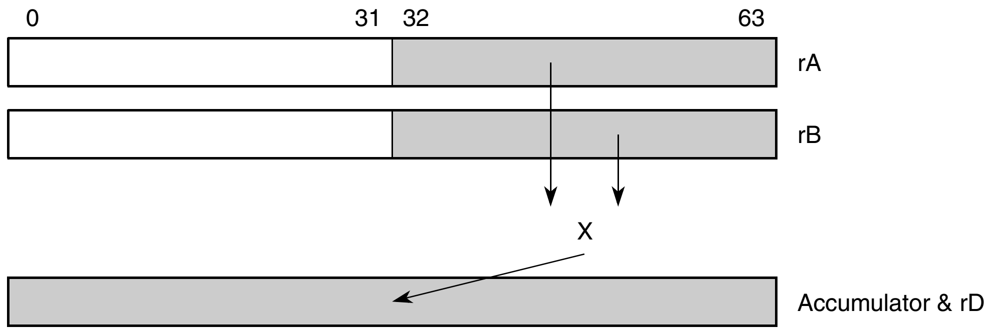


Figure 6-84. evmwsmia

evmwsmiaa

Vector Multiply Word Signed, Modulo, Integer and Accumulate

evmwsmiaa **r**D,**r**A,**r**B (M=1, F=0, S=1)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	101 0101 1001

```
temp1_{0:63} = rA_{32:63} *si rB_{32:63}
temp2_{0:64} = ACC_{0:63} + temp1_{0:63}
ACC_{0:63} = rD_{0:63} = temp2_{1:64}
```

The low signed integer word element in **r**A is multiplied by the corresponding low signed integer word element in **r**B. The intermediate product is added to the contents of the 64-bit accumulator to form a 65-bit intermediate sum. The lower 64 bits of the intermediate sum is placed back into the accumulator and also written into **r**D.

Other registers altered: ACC

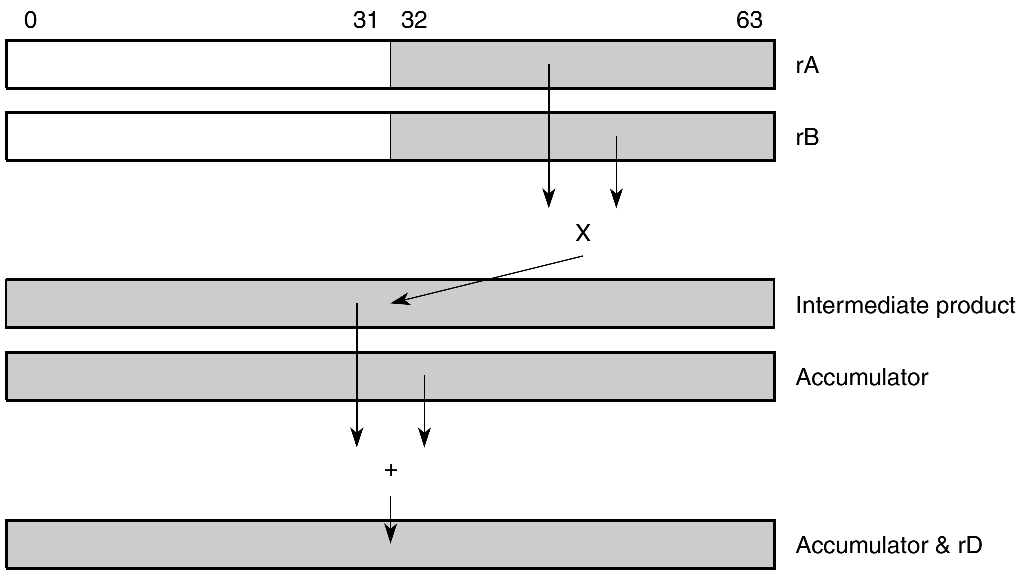


Figure 6-85. evmwsmiaa

evmwsmian evmwsmian

Vector Multiply Word Signed, Modulo, Integer and Accumulate Negative

evmwsmian **r**D,**r**A,**r**B (M=1, F=0, S=1)

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

```
temp1$_{0:63}$ = rA$_{32:63}$ *si rB$_{32:63}$
temp2$_{0:64}$ = ACC$_{0:63}$ - temp1$_{0:63}$
ACC$_{0:63}$ = rD$_{0:63}$ = temp2$_{1:64}$
```

The low signed integer word element in **r**A is multiplied by the corresponding low signed integer word element in **r**B. The intermediate product is subtracted from the contents of the 64-bit accumulator to form a 65-bit intermediate difference. The lower 64 bits of the intermediate difference is placed back into the accumulator and also written into **r**D.

Other registers altered: ACC

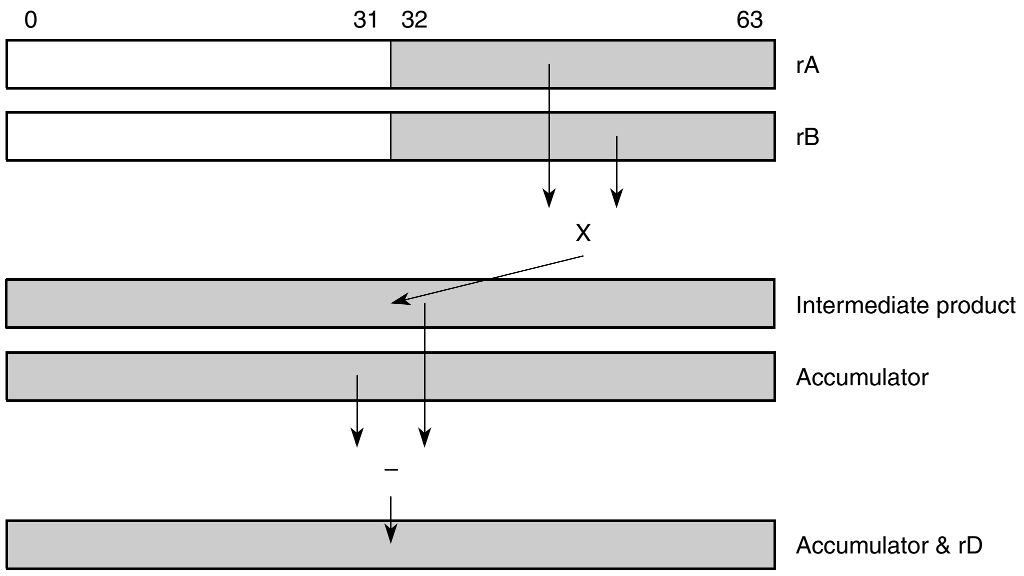


Figure 6-86. evmwsmian

evmwssf evmwssf

Vector Multiply Word Signed, Saturate, Fractional

evmwssf **r**D,**r**A,**r**B (M=0, F=1, S=1, A=0)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	100 0101 0011

```
temp_{0:64} = (rA_{32:63} * rB_{32:63}) || 0
movl = temp_0 ⊕ temp_1
rD_{0:63} = SATURATE(movh, 0x7FFFFFFFFFFFFFFF, temp_{1:64})
SPEFSCR_{OVH} = 0
SPEFSCR_{OV} = movl
SPEFSCR_{SOV} = SPEFSCR_{SOV} | movl
```

The low signed fractional word element in **r**A is multiplied by the corresponding low signed fractional word element in **r**B. The 64-bit signed fractional product is placed in **r**D. If the inputs are –1.0 and –1.0 the result is saturated to the most positive signed fraction (`0x7FFFFFFFFFFFFFFF`). If saturation occurs the overflow and summary overflow bits are recorded.

Other registers altered: SPEFSCR

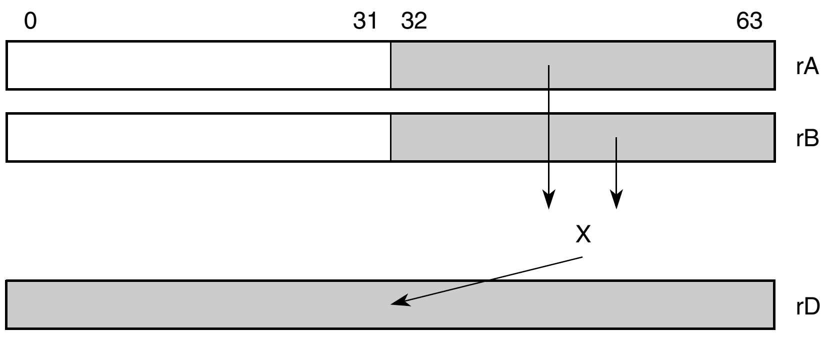


Figure 6-87. evmwssf

evmwssfa evmwssfa

Vector Multiply Word Signed, Saturate, Fractional, to Accumulator

evmwssfa **r**D,**r**A,**r**B (M=0, F=1, S=1, A=1)

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

```
temp$_{0:64}$ = (rA$_{32:63}$ * rB$_{32:63}$) || 0
movl = temp$_0$ ⊕ temp$_1$
ACC$_{0:63}$ = rD$_{0:63}$ = SATURATE(movh, 0x7FFFFFFFFFFFFFFF, temp$_{1:64}$)
SPEFSCR$_{OVH}$ = 0
SPEFSCR$_{OV}$ = movl
SPEFSCR$_{SOV}$ = SPEFSCR$_{SOV}$ | movl
```

The low signed fractional word element in **r**A is multiplied by the corresponding low signed fractional word element in **r**B. The 64-bit signed fractional product is placed in **r**D. If the inputs are –1.0 and –1.0 the result is saturated to the most positive signed fraction (0x7FFFFFFFFFFFFFFF). If saturation occurs the overflow and summary overflow bits are recorded. The result in **r**D is also placed in the accumulator.

Other registers altered: SPEFSCR, ACC

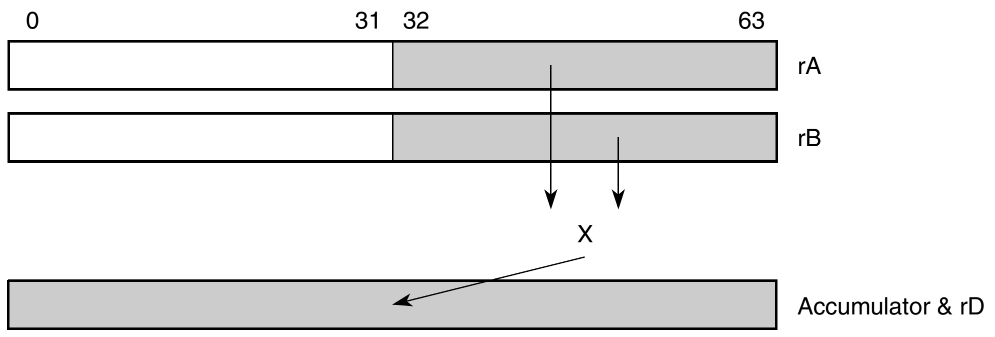


Figure 6-88. evmwssfa

evmwssfaa evmwssfaa

Vector Multiply Word Signed, Saturate, Fractional and Accumulate

evmwssfaa **r**D,**r**A,**r**B (M=0, F=1, S=1)

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

```
temp1$_{0:64}$ = (rA$_{32:63}$ * rB$_{32:63}$) || 0
mov = temp1$_0$ $\oplus$ temp1$_1$
temp2$_{0:63}$ = SATURATE(mov, 0x7FFFFFFFFFFFFFFF, temp1$_{1:64}$)
temp3$_{0:64}$ = {ACC$_0$,ACC$_{0:63}$} + {temp2$_0$,temp2$_{0:63}$}
ov = temp3$_0$ $\oplus$ temp3$_1$
rD$_{0:63}$ = SATURATE_ACC(ov, temp3$_0$, 0x8000000000000000, 0x7FFFFFFFFFFFFFFF, temp3$_{1:64}$)
ACC$_{0:63}$ = rD$_{0:63}$

SPEFSCR$_{OV}$ = mov | ov
SPEFSCR$_{OVH}$ = 0
SPEFSCR$_{SOV}$ = SPEFSCR$_{SOV}$ | mov | ov
```

The low signed fractional word element in **r**A is multiplied by the corresponding low signed fractional word element in **r**B. If the inputs are –1.0 and –1.0 the product is saturated to the most positive signed fraction (`0x7FFFFFFFFFFFFFFF`). The 64-bit intermediate product is shifted left by one bit (to eliminate the redundant sign bit) and padded on the right with a ‘0’, and this value is then added to the contents of the 64-bit accumulator to form an intermediate sum. If the intermediate sum has overflowed, the appropriate saturation value (`0x7FFFFFFFFFFFFFFF` if positive overflow or `0x8000000000000000` if negative overflow) is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 64 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word. The overflow and summary overflow bits are recorded to indicate occurrence of saturation on either the multiply or the addition.

Other registers altered: SPEFSCR, ACC

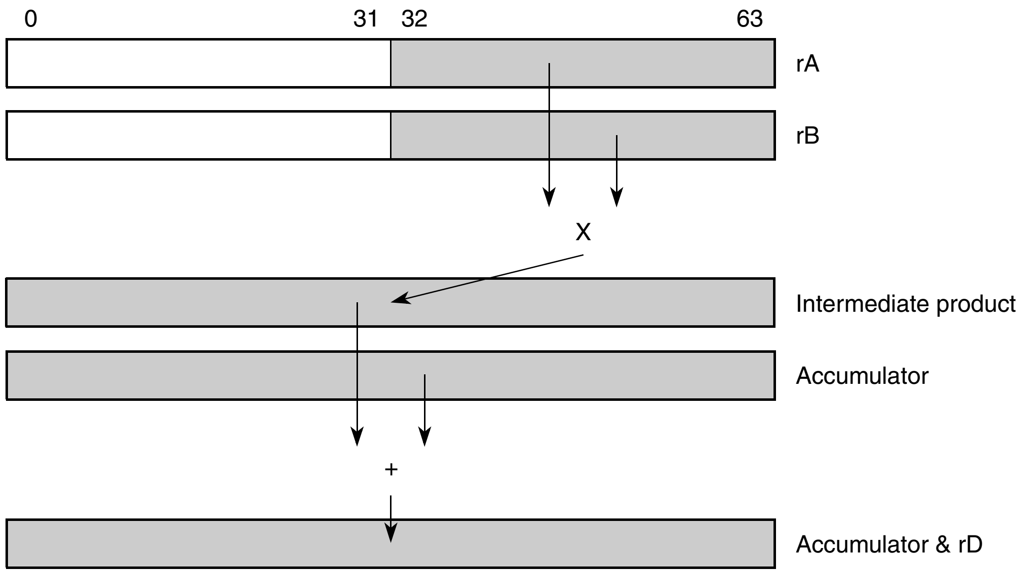


Figure 6-89. evmwssfaa

evmwssfan evmwssfan

Vector Multiply Word Signed, Saturate, Fractional and Accumulate Negative

evmwssfan **r**D,**r**A,**r**B (M=0, F=1, S=1)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	101 1101 0011

```
$temp1_{0:64}$ = ($rA_{32:63}$ * $rB_{32:63}$) || 0
mov = $temp1_0 \oplus temp1_1$
$temp2_{0:63}$ = SATURATE(mov, 0x7FFFFFFFFFFFFFFF, $temp1_{1:64}$)
$temp3_{0:64}$ = {$ACC_0$,$ACC_{0:63}$} - {$temp2_0$,$temp2_{0:63}$}
ov = $temp3_0 \oplus temp3_1$
$rD_{0:63}$ = SATURATE_ACC(ov, $temp3_0$, 0x8000000000000000, 0x7FFFFFFFFFFFFFFF, $temp3_{1:64}$)
$ACC_{0:63}$ = $rD_{0:63}$

$SPEFSCR_{OV}$ = mov | ov
$SPEFSCR_{OVH}$ = 0
$SPEFSCR_{SOV}$ = $SPEFSCR_{SOV}$ | mov | ov
```

The low signed fractional word element in **r**A is multiplied by the corresponding low signed fractional word element in **r**B. If the inputs are –1.0 and –1.0 the product is saturated to the most positive signed fraction (`0x7FFFFFFFFFFFFFFF`). The 64-bit intermediate product is shifted left by one bit (to eliminate the redundant sign bit) and padded on the right with a ‘0’, and this value is then subtracted from the contents of the 64-bit accumulator to form an intermediate sum. If the intermediate difference has overflowed, the appropriate saturation value (`0x7FFFFFFFFFFFFFFF` if positive overflow or `0x8000000000000000` if negative overflow) is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 64 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word. The overflow and summary overflow bits are recorded to indicate occurrence of saturation either the multiply or the subtraction.

Other registers altered: SPEFSCR, ACC

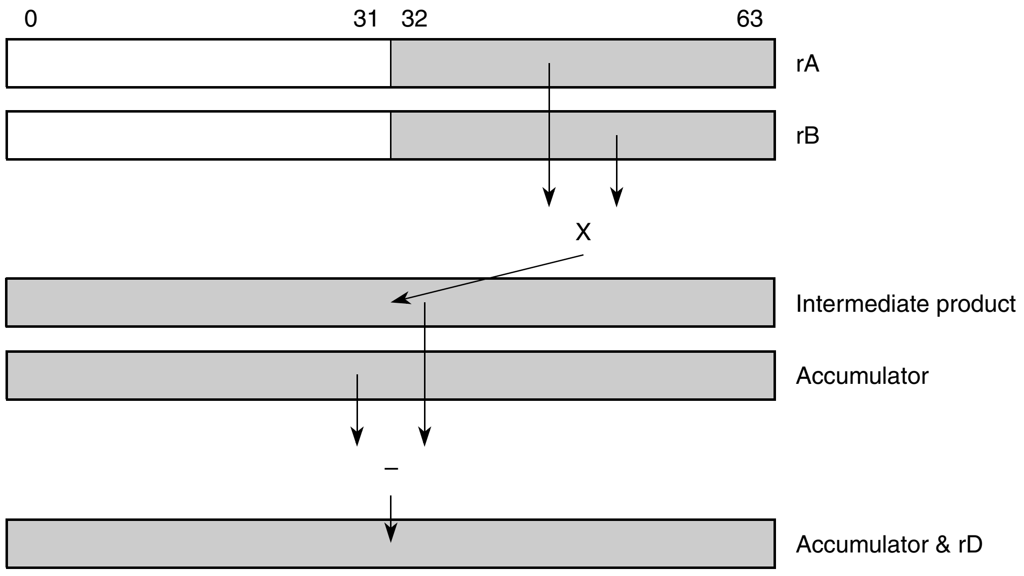


Figure 6-90. evmwssfan

evmwumi evmwumi

Vector Multiply Word Unsigned, Modulo, Integer

evmwumi **r**D,**r**A,**r**B (M=1, F=0, S=0, A=0)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	100 0101 1000

```
temp$_{0:63}$ = rA$_{32:63}$ *ui rB$_{32:63}$
ACC$_{0:63}$ = rD$_{0:63}$ = temp$_{0:63}$
```

The low unsigned integer word element in **r**A is multiplied by the corresponding low unsigned integer word element in **r**B. The 64-bit unsigned integer product is placed in **r**D.

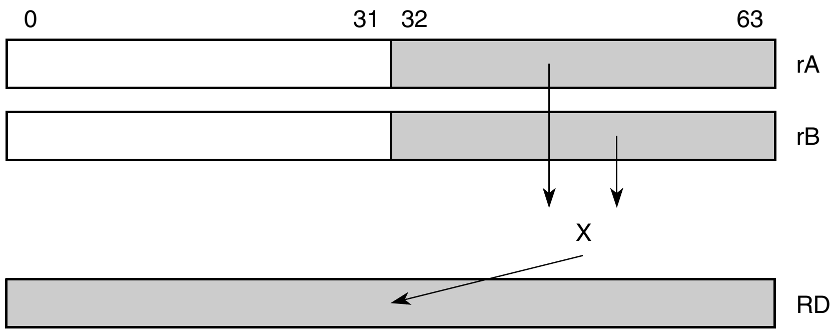


Figure 6-91. evmwumi

evmwumia evmwumia

Vector Multiply Word Unsigned, Modulo, Integer, to Accumulator

evmwumia **r**D,**r**A,**r**B (M=1, F=0, S=0, A=1)

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

```
temp$_{0:63}$ = rA$_{32:63}$ *ui rB$_{32:63}$
ACC$_{0:63}$ = rD$_{0:63}$ = temp$_{0:63}$
```

The low unsigned integer word element in **r**A is multiplied by the corresponding low unsigned integer word element in **r**B. The 64-bit unsigned integer product is placed in **r**D. The result in **r**D is also placed in the accumulator.

Other registers altered: ACC

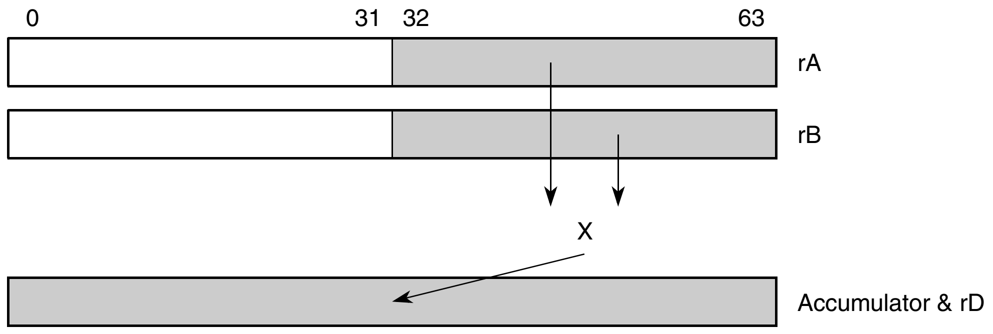


Figure 6-92. evmwumia

evmwumiaa evmwumiaa

Vector Multiply Word Unsigned, Modulo, Integer and Accumulate

evmwumiaa **r**D,**r**A,**r**B (M=1, F=0, S=0)

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

```
temp1_{0:63} = rA_{32:63} *ui rB_{32:63}
temp2_{0:64} = ACC_{0:63} + temp1_{0:63}
ACC_{0:63} = rD_{0:63} = temp2_{1:64}
```

The low unsigned integer word element in **r**A is multiplied by the corresponding low unsigned integer word element in **r**B. The intermediate product is added to the contents of the 64-bit accumulator to form a 65-bit intermediate sum. The lower 64 bits of the intermediate sum is placed back into the accumulator and also written into **r**D.

Other registers altered: ACC

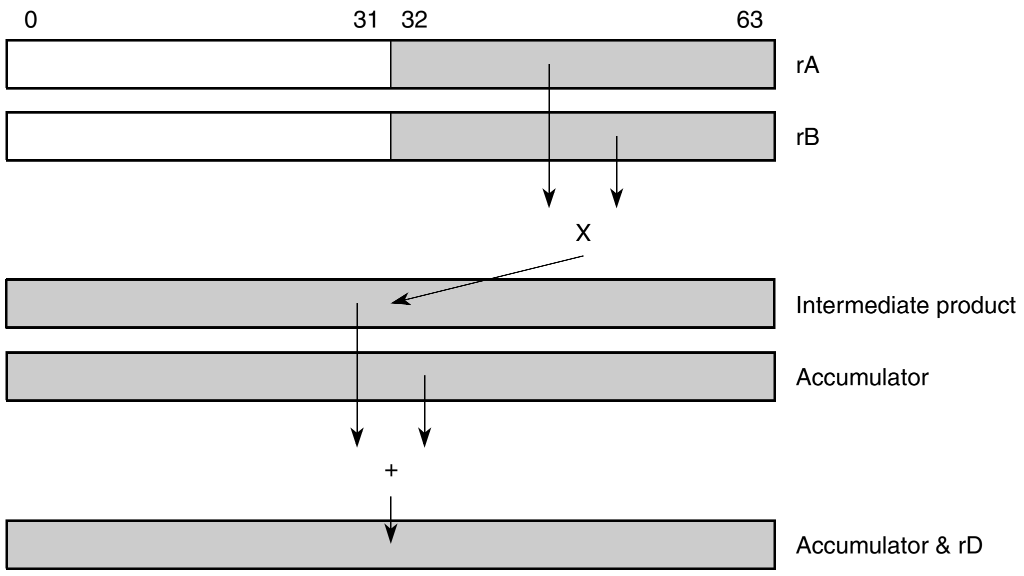


Figure 6-93. evmwumiaa

evmwumian evmwumian

Vector Multiply Word Unsigned, Modulo, Integer and Accumulate Negative

evmwumian **r**D,**r**A,**r**B (M=1, F=0, S=0)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	101 1101 1000

```
temp1_{0:63} = rA_{32:63} *ui rB_{32:63}
temp2_{0:64} = ACC_{0:63} - temp1_{0:63}
ACC_{0:63} = rD_{0:63} = temp2_{1:64}
```

The low unsigned integer word element in **r**A is multiplied by the corresponding low unsigned integer word element in **r**B. The intermediate product is subtracted from the contents of the 64-bit accumulator to form a 65-bit intermediate difference. The lower 64 bits of the intermediate difference is placed back into the accumulator and also written into **r**D.

Other registers altered: ACC

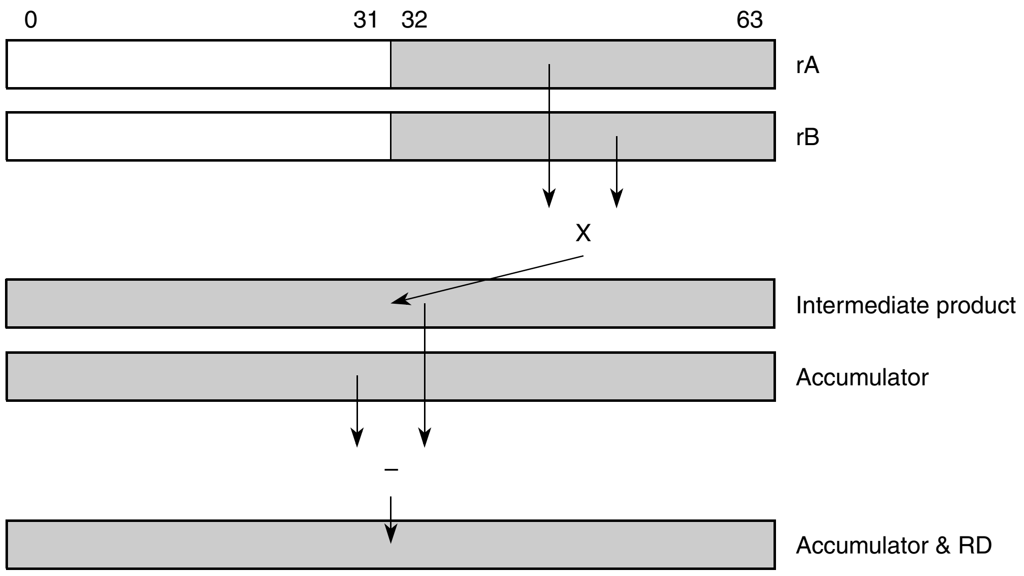


Figure 6-94. evmwumian

6.4.3 Add/subtract word to accumulator instructions

The following instructions perform addition and subtraction, with and without accumulates, using signed or unsigned integer or fractional operands, with optional saturation.

evaddsmiaaw evaddsmiaaw

Vector Add Signed, Modulo, Integer to Accumulator Word

evaddsmiaaw **r**D,**r**A (M=1, S=1)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	0000 0	100 1100 1001

```
rD0:31 = ACC0:31 + rA0:31
rD32:63 = ACC32:63 + rA32:63
ACC0:63 = rD0:63
```

Each word element in **r**A is added to the corresponding word element in the accumulator and placed into the corresponding **r**D word. The result in **r**D is also placed in the accumulator.

Other registers altered: ACC

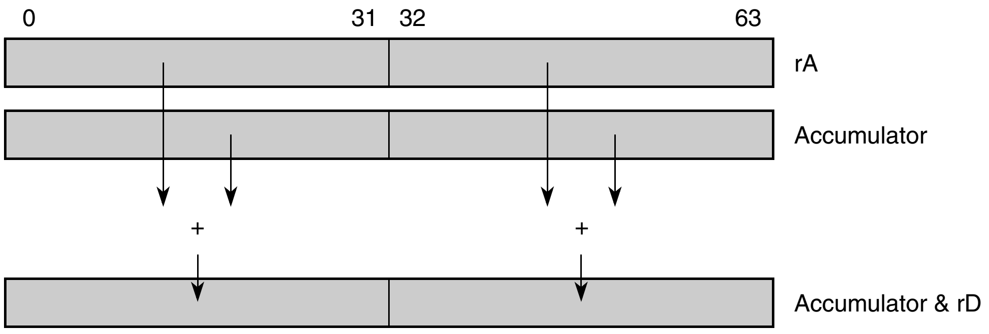


Figure 6-95. evaddsmiaaw

evaddssiaaw evaddssiaaw

Vector Add Signed, Saturate, Integer to Accumulator Word

evaddssiaaw **r**D,**r**A (M=0, S=1)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	0000 0	100 1100 0001

```
temp1_{0:32} = EXTS(ACC_{0:31}) + EXTS(rA_{0:31})
temp2_{0:32} = EXTS(ACC_{32:63}) + EXTS(rA_{32:63})
ovh = temp1_0 ⊕ temp1_1
ovl = temp2_0 ⊕ temp2_1
rD_{0:31} = SATURATE_ACC(ovh, temp1_0, 0x80000000, 0x7FFFFFFF, temp1_{1:32})
rD_{32:63} = SATURATE_ACC(ovl, temp2_0, 0x80000000, 0x7FFFFFFF, temp2_{1:32})
ACC_{0:31} = rD_{0:31}
ACC_{32:63} = rD_{32:63}
SPEFSCR_{OVH} = ovh
SPEFSCR_{OV} = ovl
SPEFSCR_{SOVH} = SPEFSCR_{SOVH} | ovh
SPEFSCR_{SOV} = SPEFSCR_{SOV} | ovl
```

Each word element in **r**A is added to the corresponding word element in the accumulator to form 33-bit intermediate sum. If the intermediate sum has overflowed, the appropriate saturation value (`0x7FFFFFFF` if positive overflow or `0x80000000` if negative overflow) is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word.

If there is an overflow from the addition, the overflow information is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR, ACC

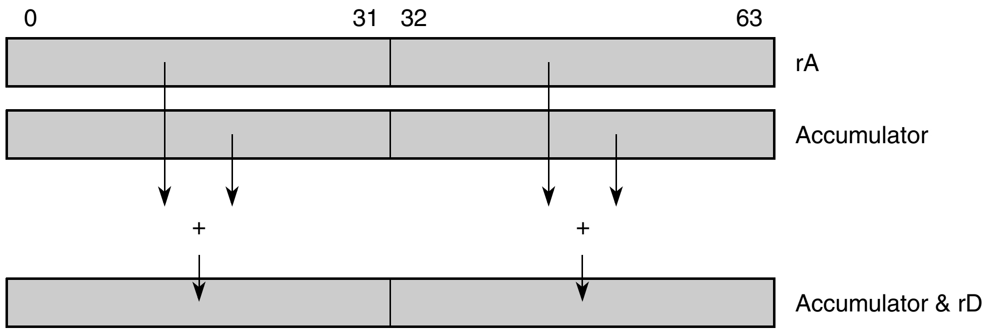


Figure 6-96. evaddssiaaw

evaddumiaaw evaddumiaaw

Vector Add Unsigned, Modulo, Integer to Accumulator Word

evaddumiaaw **r**D,**r**A (M=1, S=0)

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

$rD_{0:31} = ACC_{0:31} + rA_{0:31}$
$rD_{32:63} = ACC_{32:63} + rA_{32:63}$
$ACC_{0:63} = rD_{0:63}$

Each word element in **r**A is added to the corresponding word element in the accumulator and placed into the corresponding **r**D word. The result in **r**D is also placed in the accumulator.

Other registers altered: ACC

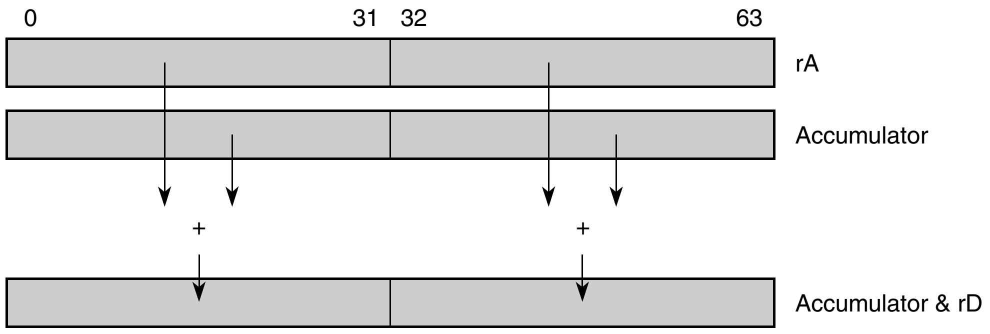


Figure 6-97. evaddumiaaw

evaddusiaaw

Vector Add Unsigned, Saturate, Integer to Accumulator Word

evaddusiaaw **r**D,**r**A (M=0, S=0)

0 — 5	6 — 10	11 — 15	16 — 20	21 — 31
4	RD	RA	0000 0	100 1100 0000

```
temp1$_{0:32}$ = EXTZ(ACC$_{0:31}$) + EXTZ(rA$_{0:31}$)
temp2$_{0:32}$ = EXTZ(ACC$_{32:63}$) + EXTZ(rA$_{32:63}$)
ovh = temp1$_0$
ovl = temp2$_0$
rD$_{0:31}$ = SATURATE(ovh, 0xFFFFFFFF, temp1$_{1:32}$)
rD$_{32:63}$ = SATURATE(ovl, 0xFFFFFFFF, temp2$_{1:32}$)

ACC$_{0:31}$ = rD$_{0:31}$
ACC$_{32:63}$ = rD$_{32:63}$
SPEFSCR$_{OVH}$ = ovh
SPEFSCR$_{OV}$ = ovl
SPEFSCR$_{SOVH}$ = SPEFSCR$_{SOVH}$ | ovh
SPEFSCR$_{SOV}$ = SPEFSCR$_{SOV}$ | ovl
```

Each word element in **r**A is added to the corresponding word element in the accumulator to form 33-bit intermediate sum. If the intermediate sum has overflowed, `0xFFFFFFFF` is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate sum are placed into the accumulator word and the corresponding **r**D word.

If there is an overflow from the addition, the overflow information is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR, ACC

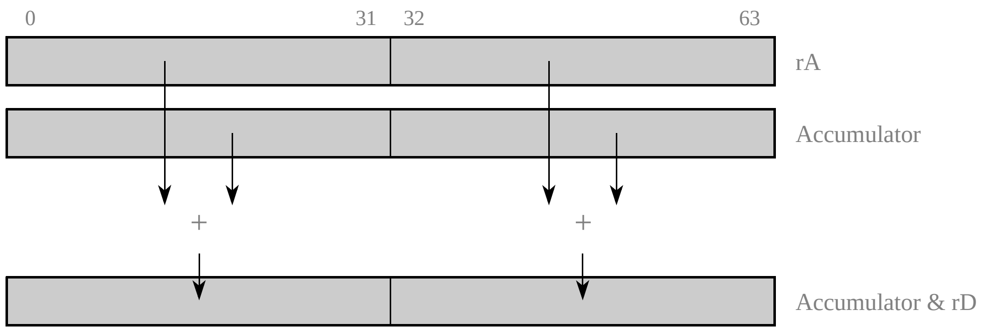


Figure 6-98. evaddusiaaw

evsubfsmiaaw evsubfsmiaaw

Vector Subtract Signed, Modulo, Integer to Accumulator Word

evsubfsmiaaw **r**D,**r**A (M=1, S=1)

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

```
rD0:31 = ACC0:31 - rA0:31
rD32:63 = ACC32:63 - rA32:63
ACC0:63 = rD0:63
```

Each word element in **r**A is subtracted from the corresponding word element in the accumulator and placed into the corresponding **r**D word. The result in **r**D is also placed in the accumulator.

Other registers altered: ACC

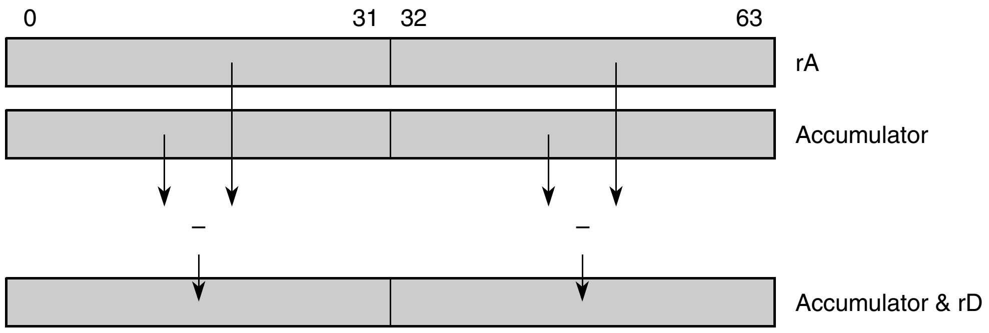


Figure 6-99. evsubfsmiaaw

evsubfssiaaw

Vector Subtract Signed, Saturate, Integer to Accumulator Word

evsubfssiaaw **r**D,**r**A (M=0, S=1)

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

```
temp1$_{0:32}$ = EXTS(ACC$_{0:31}$) - EXTS(rA$_{0:31}$)
temp2$_{0:32}$ = EXTS(ACC$_{32:63}$) - EXTS(rA$_{32:63}$)
ovh = temp1$_0$ ⊕ temp1$_1$
ovl = temp2$_0$ ⊕ temp2$_1$
rD$_{0:31}$ = SATURATE_ACC(ovh, temp1$_0$, 0x80000000, 0x7FFFFFFF, temp1$_{1:32}$)
rD$_{32:63}$ = SATURATE_ACC(ovl, temp2$_0$, 0x80000000, 0x7FFFFFFF, temp2$_{1:32}$)
ACC$_{0:31}$ = rD$_{0:31}$
ACC$_{32:63}$ = rD$_{32:63}$
SPEFSCR$_{OVH}$ = ovh
SPEFSCR$_{OV}$ = ovl
SPEFSCR$_{SOVH}$ = SPEFSCR$_{SOVH}$ | ovh
SPEFSCR$_{SOV}$ = SPEFSCR$_{SOV}$ | ovl
```

Each word element in **r**A is subtracted from the corresponding word element in the accumulator to form 33-bit intermediate difference. If the intermediate difference has overflowed, the appropriate saturation value (`0x7FFFFFFF` if positive overflow or `0x80000000` if negative overflow) is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word.

If there is an overflow from the subtraction, the overflow information is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR, ACC

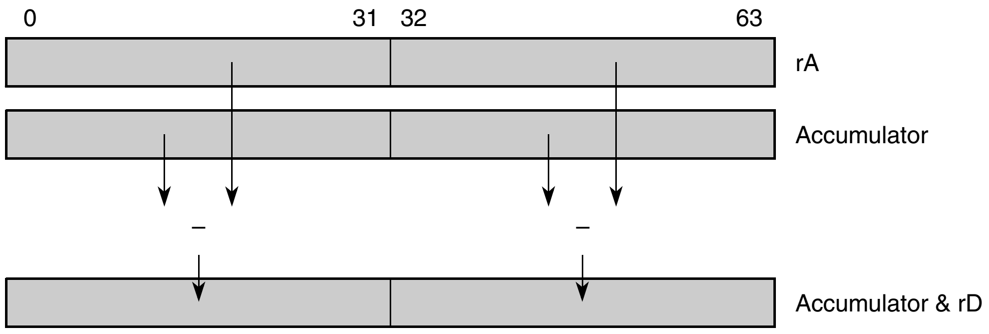


Figure 6-100. evsubfssiaaw

evsubfumiaaw evsubfumiaaw

Vector Subtract Unsigned, Modulo, Integer to Accumulator Word

evsubfumiaaw **r**D,**r**A (M=1, S=0)

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

```
rD0:31 = ACC0:31 - rA0:31
rD32:63 = ACC32:63 - rA32:63
ACC0:63 = rD0:63
```

Each word element in **r**A is subtracted from the corresponding word element in the accumulator and placed into the corresponding **r**D word. The result in **r**D is also placed in the accumulator.

Other registers altered: ACC

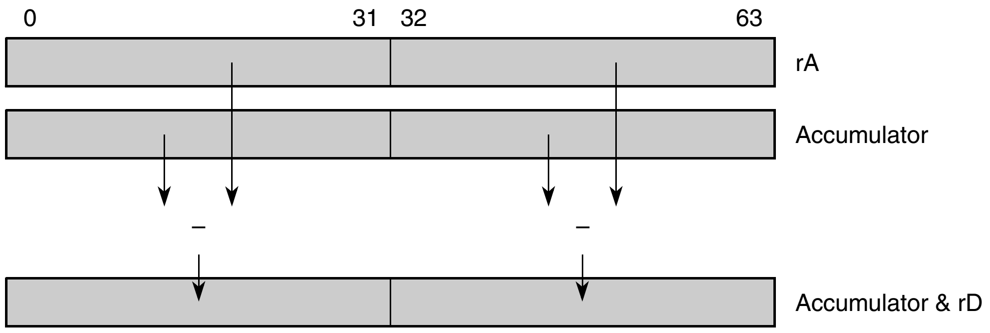


Figure 6-101. evsubfumiaaw

Vector Subtract Unsigned, Saturate, Integer to Accumulator Word

evsubfusiaaw **r**D,**r**A (M=0, S=0)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	0000 0	100 1100 0010

```
$temp1_{0:32}$ = EXTZ($ACC_{0:31}$) - EXTZ($rA_{0:31}$)
$temp2_{0:32}$ = EXTZ($ACC_{32:63}$) - EXTZ($rA_{32:63}$)
ovh = $temp1_{0}$
ovl = $temp2_{0}$
$rD_{0:31}$ = SATURATE(ovh, 0x00000000, $temp1_{1:32}$)
$rD_{32:63}$ = SATURATE(ovl, 0x00000000, $temp2_{1:32}$)
$ACC_{0:31}$ = $rD_{0:31}$
$ACC_{32:63}$ = $rD_{32:63}$
$SPEFSCR_{OVH}$ = ovh
$SPEFSCR_{OV}$ = ovl
$SPEFSCR_{SOVH}$ = $SPEFSCR_{SOVH}$ | ovh
$SPEFSCR_{SOV}$ = $SPEFSCR_{SOV}$ | ovl
```

Each word element in **r**A is subtracted from the corresponding word element in the accumulator to form 33-bit intermediate difference. If the intermediate difference has underflowed, `0x00000000` is placed into the accumulator word and the corresponding **r**D word. Otherwise, the low 32 bits of the intermediate difference are placed into the accumulator word and the corresponding **r**D word.

If there is an underflow from the subtraction, the underflow information is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR, ACC

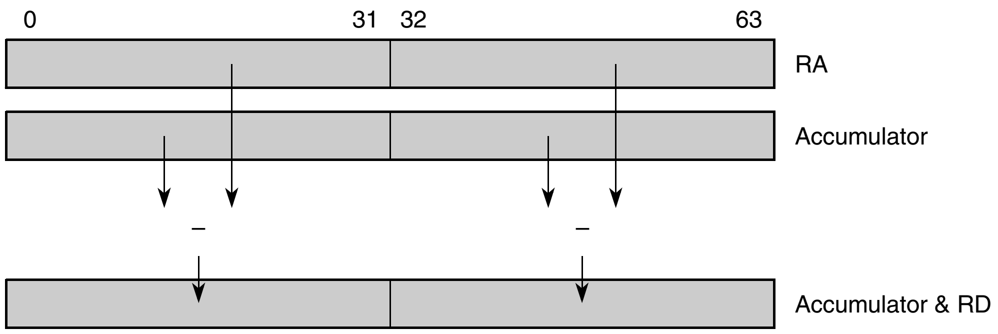


Figure 6-102. evsubfusiaaw

6.4.4 Initializing and reading the accumulator

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

```
evxor RD, RD, RD                  // Zero the contents of RD, not necessary if
                                      // a zero is available in some register.
```

```
      evmwumiaa RD, RD, RD                    // Multiply 0 with 0, add the 0 result to
                                        // accumulator and store back the value in acc and RD
```

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

evmra evmra

Move Register to Accumulator

evmra **r**D,**r**A

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

```
$RD_{0:63}$ = $acc_{0:63}$ = $RA_{0:63}$
```

The contents of **r**A are written into the accumulator and copied into **r**D. This is the method for initializing the accumulator.

6.5 SPE vector load/store instructions

SPE Vector load and store instructions are provided with a variety of options. The mnemonics are formed as follows:

ev{l,st}<X><Y>[Z]x

- X specifies the size of the load
- Y specifies the size of data packed into the value being loaded. Thus **evldhx** specified a load that brings in a double-word composed of four half words.
- Z specifies the operation to be performed such as unpack or splat.

All load and store instructions are specified as indexed forms. A specification of a 0 in the **r**A field of the instruction results in the non-indexed form of the instruction. For all loads and stores, only the lower 32 bits of registers **r**A and **r**B are used and the effective address is 32 bits.

PowerISA 2.06 load instructions are implemented such that the upper half of all registers are left unchanged for a load.

evldd

Vector Load Double into Double

evldd **r**D,**d**(**r**A)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	UIMM[1]	0 1 1 0 0 0 0 0 0 0 1

[1] **d** = UIMM<<3

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*8)
RD = MEM(EA,8)
```

Figure 6-103 shows how bytes are loaded into **r**D as determined by the endian mode.

Byte addr	0	1	2	3	4	5	6	7
Memory	a	b	c	d	e	f	g	h
GPR in big endian	a	b	c	d	e	f	g	h
GPR in little endian	h	g	f	e	d	c	b	a

Figure 6-103. evldd results in big- and little-endian modes

evlddx evlddx

Vector Load Double into Double Indexed

evlddx **r**D,**r**A,**r**B

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

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
RD = MEM(EA,8)
```

Figure 6-104 shows how bytes are loaded into **r**D as determined by the endian mode.

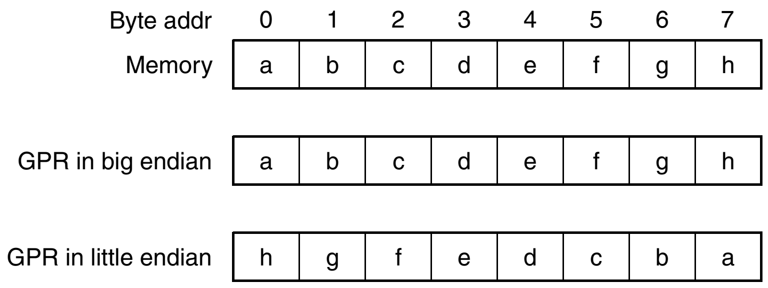

Byte addr 0 1 2 3 4 5 6 7
Memory a b c d e f g h
GPR in big endian a b c d e f g h
GPR in little endian h g f e d c b a

Figure 6-104. evlddx results in big- and little-endian modes

evldw <span>evldw</span>

Vector Load Double into Words

evldw **r**D,**d**(**r**A)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	UIMM[1]	0 1 1 0 0 0 0 0 0 1 1

[1] **d** = UIMM<<3

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*8)
RD$_{0:31}$ = MEM(EA,4)
RD$_{32:63}$ = MEM(EA+4,4)
```

Figure 6-105 shows how bytes are loaded into **r**D as determined by the endian mode.

Byte addr	0	1	2	3	4	5	6	7
Memory	a	b	c	d	e	f	g	h
GPR in big endian	a	b	c	d	e	f	g	h
GPR in little endian	d	c	b	a	h	g	f	e

Figure 6-105. evldw results in big- and little-endian modes

evldwx evldwx

Vector Load Double into Words Indexed

evldwx **r**D,**r**A,**r**B

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	011 0000 0010

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
$RD_{0:31}$ = MEM(EA,4)
$RD_{32:63}$ = MEM(EA+4,4)
```

Figure 6-106 shows how bytes are loaded into **r**D as determined by the endian mode.

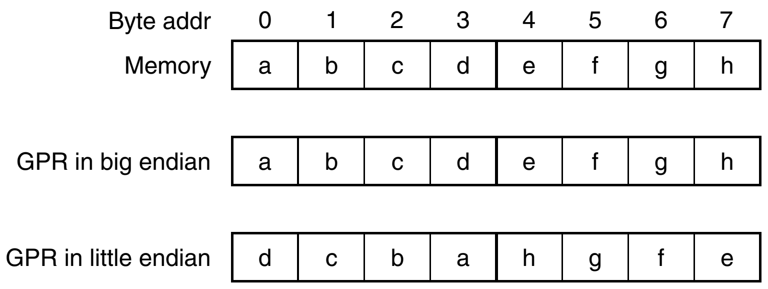


Figure 6-106. evldwx results in big- and little-endian modes

evldh

Vector Load Double into Halfwords

evldh **r**D,**d**(**r**A)

0 … 5	6 … 10	11 … 15	16 … 20	21 … 31
4	RD	RA	UIMM[1]	011 0000 0101

[1] **d** = UIMM<<3

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*8)
$RD_{0:15}$ = MEM(EA,2)
$RD_{16:31}$ = MEM(EA+2,2)
$RD_{32:47}$ = MEM(EA+4,2)
$RD_{48:63}$ = MEM(EA+6,2)
```

Figure 6-107 shows how bytes are loaded into **r**D as determined by the endian mode.

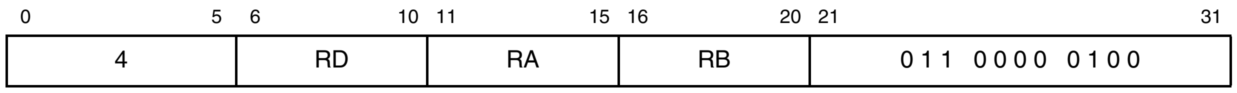
Byte addr	0	1	2	3	4	5	6	7
Memory	a	b	c	d	e	f	g	h
GPR in big endian	a	b	c	d	e	f	g	h
GPR in little endian	b	a	d	c	f	e	h	g

Figure 6-107. evldh results in big- and little-endian modes

evldhx

Vector Load Double into Halfwords Indexed

evldhx **r**D,**r**A,**r**B



```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
RD$_{0:15}$ = MEM(EA,2)
RD$_{16:31}$ = MEM(EA+2,2)
RD$_{32:47}$ = MEM(EA+4,2)
RD$_{48:63}$ = MEM(EA+6,2)
```

Figure 6-108 shows how bytes are loaded into **r**D as determined by the endian mode.

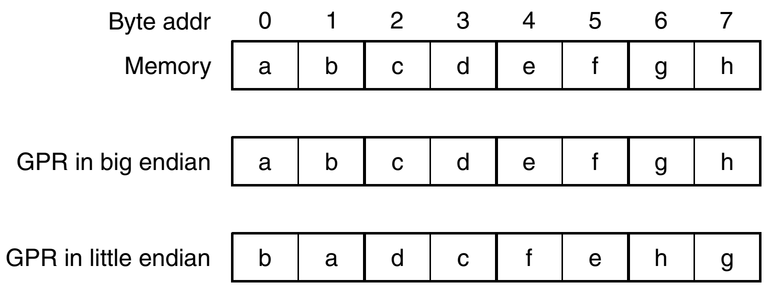


Figure 6-108. evldhx results in big- and little-endian modes

evlwhe evlwhe

Vector Load Word into Half words Even

evlwhe **r**D,**d**(**r**A)

0 — 5	6 — 10	11 — 15	16 — 20	21 — 31
4	RD	RA	UIMM[1]	0 1 1 0 0 0 1 0 0 0 1

[1] **d** = UIMM<<2

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*4)
RD$_{0:15}$ = MEM(EA,2)
RD$_{16:31}$ = 0x0000
RD$_{32:47}$ = MEM(EA+2,2)
RD$_{48:63}$ = 0x0000
```

Figure 6-109 shows how bytes are loaded into **r**D as determined by the endian mode.

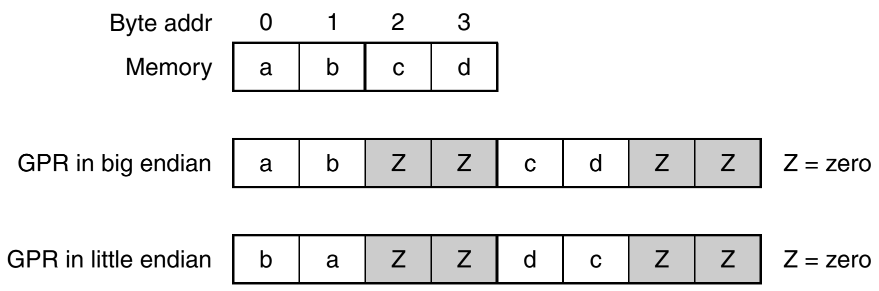


Figure 6-109. evlwhe results in big- and little-endian modes

evlwhex

Vector Load Word into Halfwords Even Indexed

evlwhex **r**D,**r**A,**r**B

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

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
RD$_{0:15}$ = MEM(EA,2)
RD$_{16:31}$ = 0x0000
RD$_{32:47}$ = MEM(EA+2,2)
RD$_{48:63}$ = 0x0000
```

Figure 6-110 shows how bytes are loaded into **r**D as determined by the endian mode.

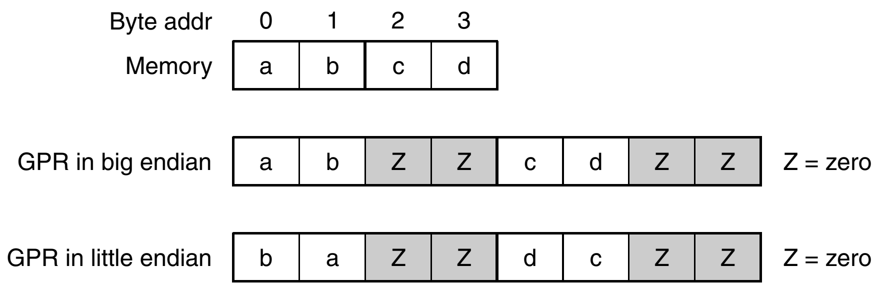


Figure 6-110. evlwhex results in big- and little-endian modes

evlwhou

Vector Load Word into Halfwords Odd Unsigned (zero-extended)

evlwhou **r**D,**d**(**r**A)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	UIMM[1]	0 1 1 0 0 0 1 0 1 0 1

[1] **d** = UIMM<<2

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*4)
RD0:15 = 0x0000
RD16:31 = MEM(EA,2)
RD32:47 = 0x0000
RD48:63 = MEM(EA+2,2)
```

Figure 6-111 shows how bytes are loaded into **r**D as determined by the endian mode.

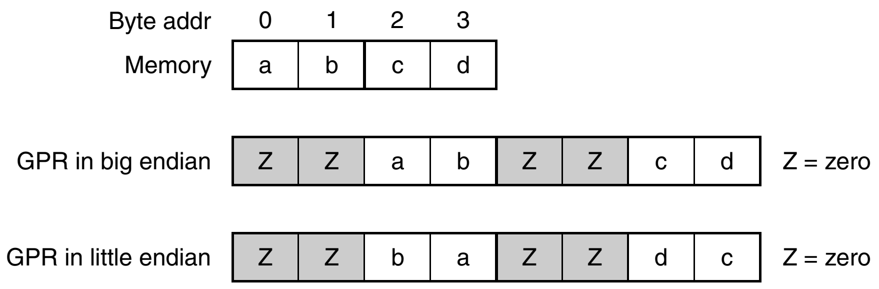


Figure 6-111. evlwhou results in big- and little-endian modes

evlwhoux

Vector Load Word into Halfwords Odd Unsigned Indexed (zero-extended)

evlwhoux **r**D,**r**A,**r**B

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

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
RD$_{0:15}$ = 0x0000
RD$_{16:31}$ = MEM(EA,2)
RD$_{32:47}$ = 0x0000
RD$_{48:63}$ = MEM(EA+2,2)
```

Figure 6-112 shows how bytes are loaded into **r**D as determined by the endian mode.

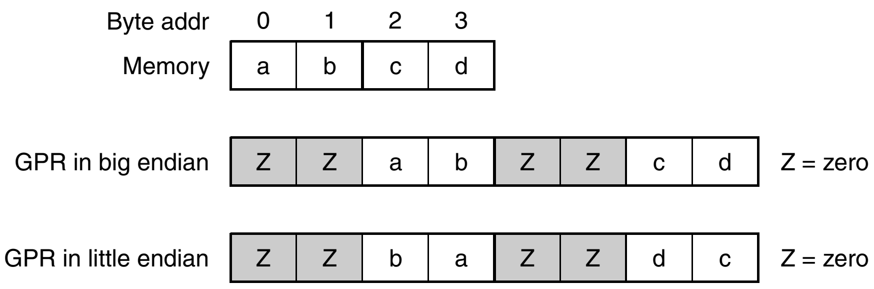


Figure 6-112. evlwhoux results in big- and little-endian modes

Vector Load Word into Halfwords Odd Signed (with sign extension)

evlwhos **r**D,**d**(**r**A)

0–5	6–10	11–15	16–20	21–31
4	RD	RA	UIMM[1]	0 1 1 0 0 0 1 0 1 1 1

[1] **d** = UIMM<<2

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*4)
RD0:31 = EXTS(MEM(EA,2))
RD32:63 = EXTS(MEM(EA+2,2))
```

Figure 6-113 shows how bytes are loaded into **r**D as determined by the endian mode.

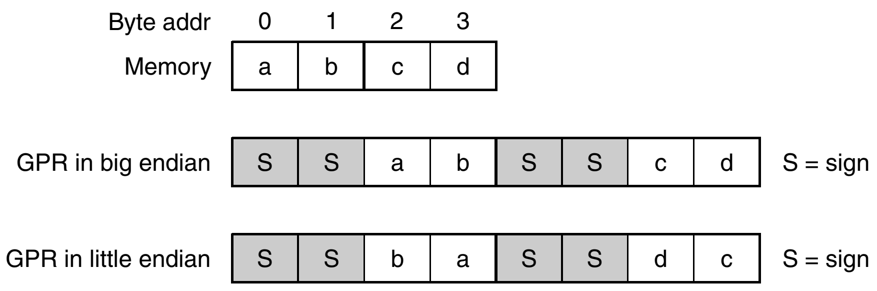


Figure 6-113. evlwhos results in big- and little-endian modes

In the big-endian memory, the msb of a and c are sign-extended. In the little-endian memory, the msb of b and d are sign-extended.

evlwhosx evlwhosx

Vector Load Word into Halfwords Odd Signed Indexed (with sign extension)

evlwhosx **r**D,**r**A,**r**B

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

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
RD$_{0:31}$ = EXTS(MEM(EA,2))
RD$_{32:63}$ = EXTS(MEM(EA+2,2))
```

Figure 6-114 shows how bytes are loaded into **r**D as determined by the endian mode.

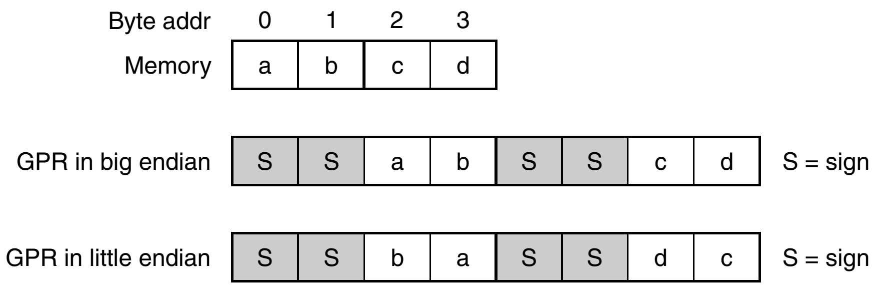


Figure 6-114. evlwhosx results in big- and little-endian modes

In the big-endian memory, the msbs of a and c are sign-extended. In the little-endian memory, the msbs of b and d are sign-extended.

evlwwsplat

Vector Load Word into Word and Splat

evlwwsplat **r**D,**d**(**r**A)

0 – 5	6 – 10	11 – 15	16 – 20	21 – 31
4	RD	RA	UIMM[1]	0 1 1 0 0 0 1 1 0 0 1

[1] **d** = UIMM<<2

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*4)
RD0:31 = MEM(EA,4)
RD32:63 = MEM(EA,4)
```

Figure 6-115 shows how bytes are loaded into **r**D as determined by the endian mode.

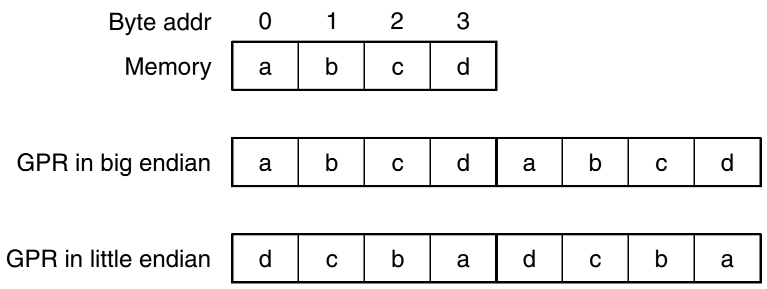


Figure 6-115. evlwwsplat results in big- and little-endian modes

evlwwsplatx evlwwsplatx

Vector Load Word into Word and Splat Indexed

evlwwsplatx rD,rA,rB

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	011 0001 1000

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
RD$_{0:31}$ = MEM(EA,4)
RD$_{32:63}$ = MEM(EA,4)
```

Figure 6-116 shows how bytes are loaded into **r**D as determined by the endian mode.

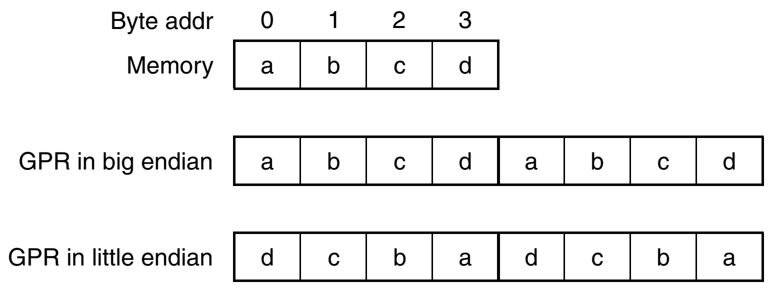


Figure 6-116. evlwwsplatx results in big- and little-endian modes

evlwhsplat

Vector Load Word into Halfwords and Splat

evlwhsplat **r**D,**d**(**r**A)

0 … 5	6 … 10	11 … 15	16 … 20	21 … 31
4	RD	RA	UIMM[1]	0 1 1 0 0 0 1 1 1 0 1

[1] **d** = UIMM<<2

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*4)
RD$_{0:15}$ = MEM(EA,2)
RD$_{16:31}$ = MEM(EA,2)
RD$_{32:47}$ = MEM(EA+2,2)
RD$_{48:63}$ = MEM(EA+2,2)
```

Figure 6-117 shows how bytes are loaded into **r**D as determined by the endian mode.

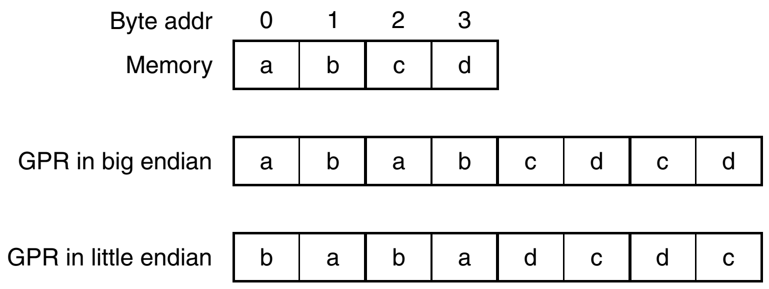


Figure 6-117. evlwhsplat results in big- and little-endian modes

evlwhsplatx

Vector Load Word into Halfwords and Splat Indexed

evlwhsplatx **r**D,**r**A,**r**B

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	011 0001 1100

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
$RD_{0:15}$ = MEM(EA,2)
$RD_{16:31}$ = MEM(EA,2)
$RD_{32:47}$ = MEM(EA+2,2)
$RD_{48:63}$ = MEM(EA+2,2)
```

Figure 6-118 shows how bytes are loaded into **r**D as determined by the endian mode.

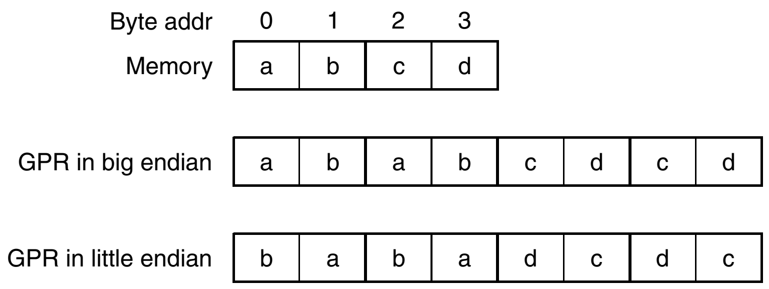


Figure 6-118. evlwhsplatx results in big- and little-endian modes

evlhhesplat

Vector Load Halfword into Halfword Even and Splat

evlhhesplat **r**D,**d**(**r**A)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	UIMM[1]	0 1 1 0 0 0 0 1 0 0 1

[1] **d** = UIMM<<1

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*2)
$RD_{0:15}$ = MEM(EA,2)
$RD_{16:31}$ = 0x0000
$RD_{32:47}$ = MEM(EA,2)
$RD_{48:63}$ = 0x0000
```

Figure 6-119 shows how bytes are loaded into **r**D as determined by the endian mode.

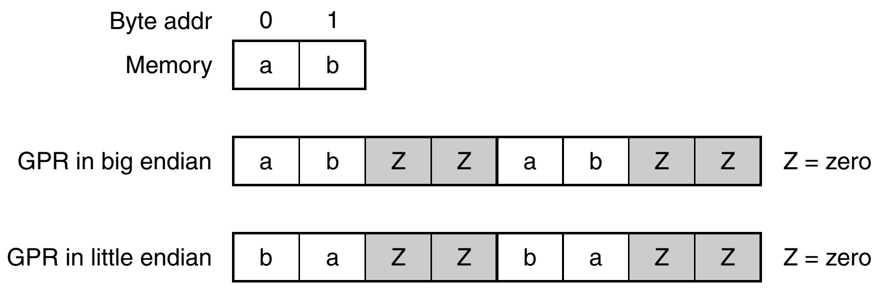


Figure 6-119. evlhhesplat results in big- and little-endian modes

evlhhesplatx evlhhesplatx

Vector Load Halfword into Halfword Even and Splat Indexed

evlhhesplatx rD,rA,rB

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	011 0000 1000

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
RD$_{0:15}$ = MEM(EA,2)
RD$_{16:31}$ = 0x0000
RD$_{32:47}$ = MEM(EA,2)
RD$_{48:63}$ = 0x0000
```

Figure 6-120 shows how bytes are loaded into **r**D as determined by the endian mode.

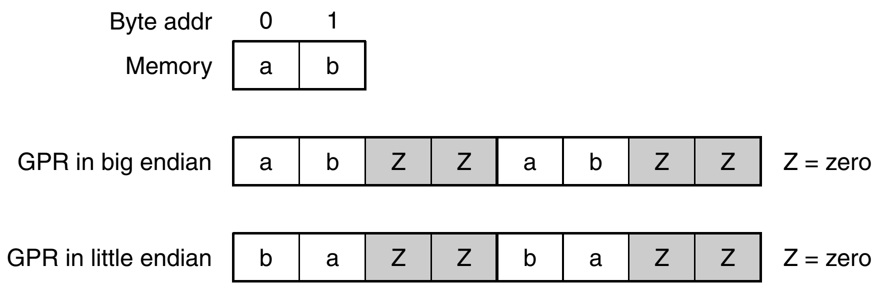


Figure 6-120. evlhhesplatx results in big- and little-endian modes

evlhhousplat evlhhousplat

Vector Load Halfword into Halfword Odd Unsigned and Splat

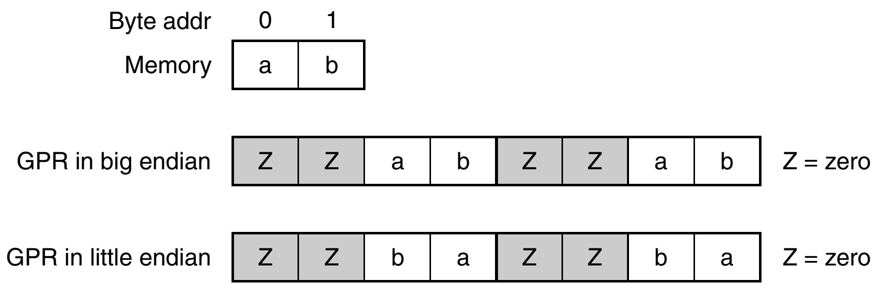
evlhhousplat **r**D,**d**(**r**A)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	UIMM[1]	0 1 1 0 0 0 0 1 1 0 1

[1] **d** = UIMM<<1

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*2)
RD$_{0:15}$ = 0x0000
RD$_{16:31}$ = MEM(EA,2)
RD$_{32:47}$ = 0x0000
RD$_{48:63}$ = MEM(EA,2)
```

Figure 6-121 shows how bytes are loaded into **r**D as determined by the endian mode.



Byte addr 0 1
Memory a b
GPR in big endian Z Z a b Z Z a b Z = zero
GPR in little endian Z Z b a Z Z b a Z = zero

Figure 6-121. evlhhousplat results in big- and little-endian modes

evlhhousplatx evlhhousplatx

Vector Load Halfword into Halfword Odd Unsigned and Splat Indexed

evlhhousplatx **r**D,**r**A,**r**B

0–5	6–10	11–15	16–20	21–31
4	RD	RA	RB	011 0000 1100

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
RD0:15 = 0x0000
RD16:31 = MEM(EA,2)
RD32:47 = 0x0000
RD48:63 = MEM(EA,2)
```

Figure 6-122 shows how bytes are loaded into **r**D as determined by the endian mode.

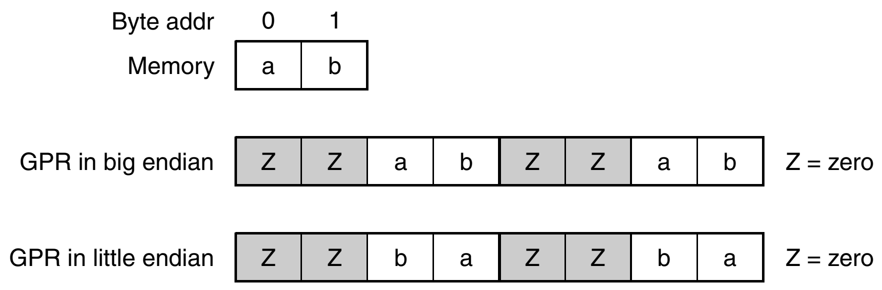


Figure 6-122. evlhhousplatx results in big- and little-endian modes

evlhhossplat evlhhossplat

Vector Load Halfword into Halfword Odd Signed and Splat

evlhhossplat **r**D,**d**(**r**A)

0 5	6 10	11 15	16 20	21 31
4	RD	RA	UIMM[1]	0 1 1 0 0 0 0 1 1 1 1

[1] **d** = UIMM<<1

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*2)
RD$_{0:31}$ = EXTS(MEM(EA,2))
RD$_{32:63}$ = EXTS(MEM(EA,2))
```

Figure 6-123 shows how bytes are loaded into **r**D as determined by the endian mode.

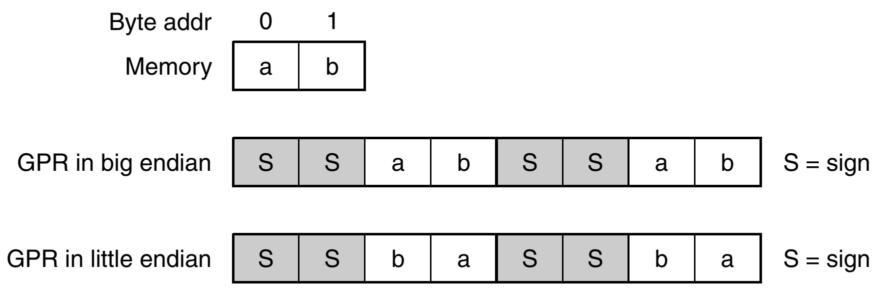


Figure 6-123. evlhhossplat results in big- and little-endian modes

In big-endian memory, the msb of a is sign-extended. In the little-endian memory, the msb of b is sign-extended.

evlhhossplatx evlhhossplatx

Vector Load Halfword into Halfword Odd Signed and Splat Indexed

evlhhossplatx **r**D,**r**A,**r**B

0 5	6 10	11 15	16 20	21 31
4	RD	RA	RB	011 0000 1110

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
RD$_{0:31}$ = EXTS(MEM(EA,2))
RD$_{32:63}$ = EXTS(MEM(EA,2))
```

Figure 6-124 shows how bytes are loaded into **r**D as determined by the endian mode.

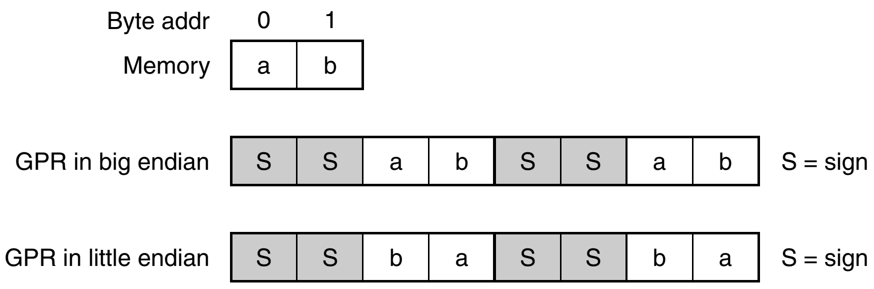


Figure 6-124. evlhhossplatx results in big- and little-endian modes

In big-endian memory, the msb of a is sign-extended. In the little-endian memory, the msb of b is sign-extended.

evstdd evstdd

Vector Store Double of Double

evstdd **rS,d(rA)**

0–5	6–10	11–15	16–20	21–31
4	RS	RA	UIMM[1]	0 1 1 0 0 1 0 0 0 0 1

[1] **d** = UIMM<<3

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*8)
MEM(EA,8) = RS0:63
```

Figure 6-125 shows how bytes are stored in memory as determined by the endian mode.

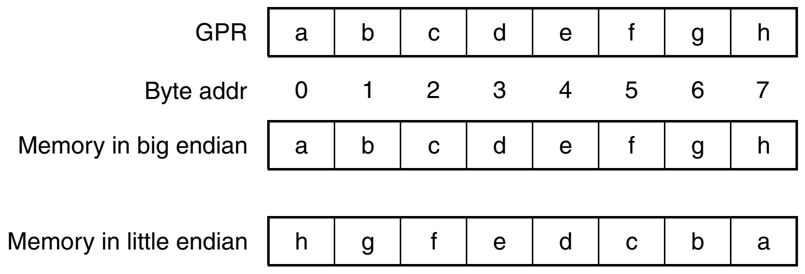


Figure 6-125. evstdd results in big- and little-endian modes

evstddx evstddx

Vector Store Double of Double Indexed

evstddx **r**S,**r**A,**r**B

0–5	6–10	11–15	16–20	21–31
4	RS	RA	RB	011 0010 0000

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
MEM(EA,8) = $RS_{0:63}$
```

Figure 6-126 shows how bytes are stored in memory as determined by the endian mode.

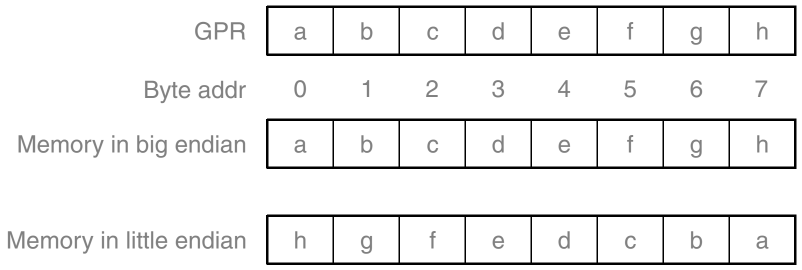


Figure 6-126. evstddx results in big- and little-endian modes

evstdw

Vector Store Double of Two Words

evstdw **rS,d(rA)**

0 — 5	6 — 10	11 — 15	16 — 20	21 — 31
4	RS	RA	UIMM[1]	011 0010 0011

[1] **d** = UIMM<<3

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*8)
MEM(EA,4) = RS$_{0:31}$
MEM(EA+4,4) = RS$_{32:63}$
```

Figure 6-127 shows how bytes are stored in memory as determined by the endian mode.

GPR	a	b	c	d	e	f	g	h
Byte addr	0	1	2	3	4	5	6	7
Memory in big endian	a	b	c	d	e	f	g	h
Memory in little endian	d	c	b	a	h	g	f	e

Figure 6-127. evstdw results in big- and little-endian modes

evstdwx evstdwx

Vector Store Double of Two Words Indexed

evstdwx **r**S,**r**A,**r**B

0–5	6–10	11–15	16–20	21–31
4	RS	RA	RB	011 0010 0010

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
MEM(EA,4) = $RS_{0:31}$
MEM(EA+4,4) = $RS_{32:63}$
```

Figure 6-128 shows how bytes are stored in memory as determined by the endian mode.

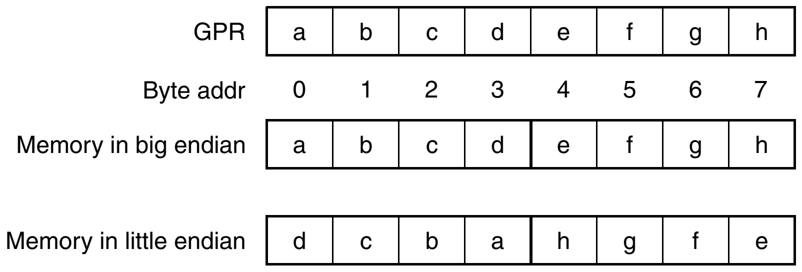


Figure 6-128. evstdwx results in big- and little-endian modes

evstdh

evstdh

Vector Store Double of Four Halfwords

evstdh **rS,d(rA)**

0 5	6 10	11 15	16 20	21 31
4	RS	RA	UIMM[1]	0 1 1 0 0 1 0 0 1 0 1

[1] **d** = UIMM<<3

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*8)
MEM(EA,2) = RS$_{0:15}$
MEM(EA+2,2) = RS$_{16:31}$
MEM(EA+4,2) = RS$_{32:47}$
MEM(EA+6,2) = RS$_{48:63}$
```

Figure 6-129 shows how bytes are stored in memory as determined by the endian mode.

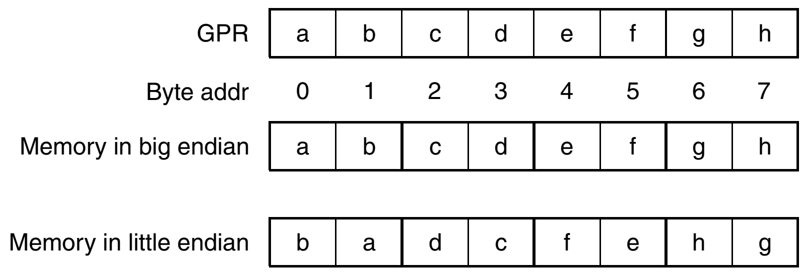


Figure 6-129. evstdh results in big- and little-endian modes

evstdhx   evstdhx

Vector Store Double of Four Halfwords Indexed

evstdhx **r**S,**r**A,**r**B

0–5	6–10	11–15	16–20	21–31
4	RS	RA	RB	011 0010 0100

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
MEM(EA,2) = $RS_{0:15}$
MEM(EA+2,2) = $RS_{16:31}$
MEM(EA+4,2) = $RS_{32:47}$
MEM(EA+6,2) = $RS_{48:63}$
```

Figure 6-130 shows how bytes are stored in memory as determined by the endian mode.

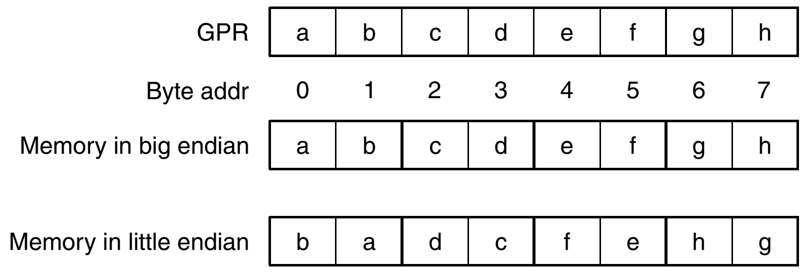


Figure 6-130. evstdhx results in big- and little-endian modes

evstwwe evstwwe

Vector Store Word of Word from Even

evstwwe **rS,d(rA)**

0–5	6–10	11–15	16–20	21–31
4	RS	RA	UIMM[1]	0 1 1 0 0 1 1 1 0 0 1

[1] **d** = UIMM<<2

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*4)
MEM(EA,4) = RS0:31
```

Figure 6-131 shows how bytes are stored in memory as determined by the endian mode.

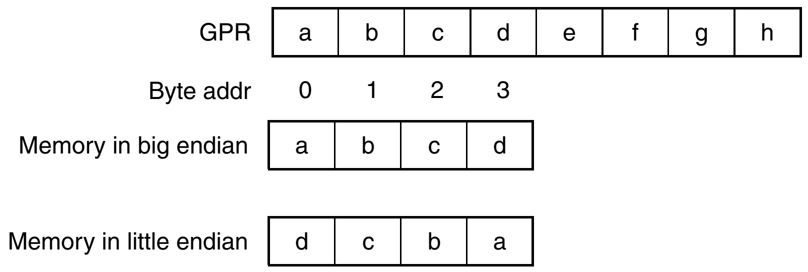


Figure 6-131. evstwwe results in big- and little-endian modes

evstwwex evstwwex

Vector Store Word of Word from Even Indexed

evstwwex **r**S,**r**A,**r**B

0–5	6–10	11–15	16–20	21–31
4	RS	RA	RB	0 1 1 0 0 1 1 1 0 0 0

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
MEM(EA,4) = RS$_{0:31}$
```

Figure 6-132 shows how bytes are stored in memory as determined by the endian mode.

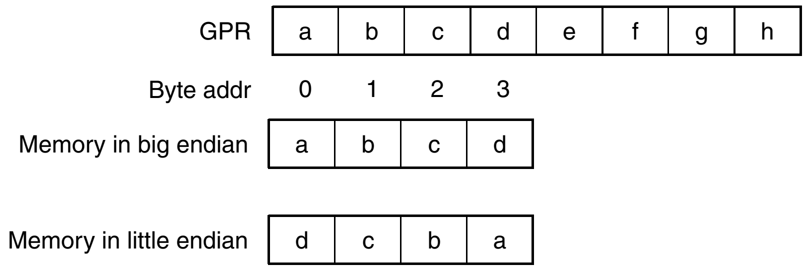


Figure 6-132. evstwwex results in big- and little-endian modes

evstwwo evstwwo

Vector Store Word of Word from Odd

evstwwo **rS,d(rA)**

0–5	6–10	11–15	16–20	21–31
4	RS	RA	UIMM[1]	0 1 1 0 0 1 1 1 1 0 1

[1] **d** = UIMM<<2

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*4)
MEM(EA,4) = rS$_{32:63}$
```

Figure 6-133 shows how bytes are stored in memory as determined by the endian mode.

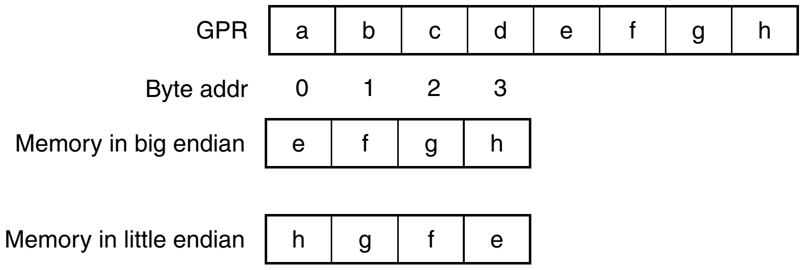


Figure 6-133. evstwwo results in big- and little-endian modes

Vector Store Word of Word from Odd Indexed

evstwwox **rS,rA,rB**

0 … 5	6 … 10	11 … 15	16 … 20	21 … 31
4	RS	RA	RB	0 1 1 0 0 1 1 1 1 0 0

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
MEM(EA,4) = $rS_{32:63}$
```

Figure 6-134 shows how bytes are stored in memory as determined by the endian mode.

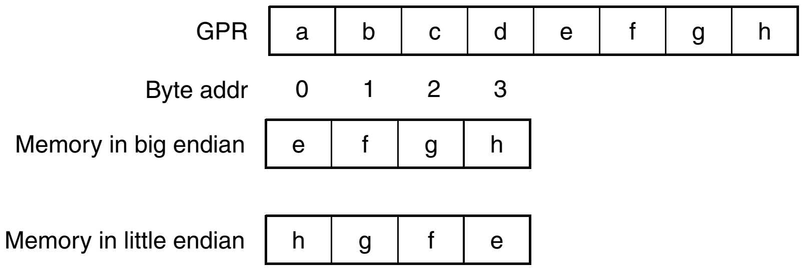


Figure 6-134. evstwwox results in big- and little-endian modes

evstwhe

Vector Store Word of Two Halfwords from Even

evstwhe **rS,d(rA)**

0–5	6–10	11–15	16–20	21–31
4	RS	RA	UIMM[1]	011 0011 0001

[1] **d** = UIMM<<2

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*4)
MEM(EA,2) = RS$_{0:15}$
MEM(EA+2,2) = RS$_{32:47}$
```

Figure 6-135 shows how bytes are stored in memory as determined by the endian mode.

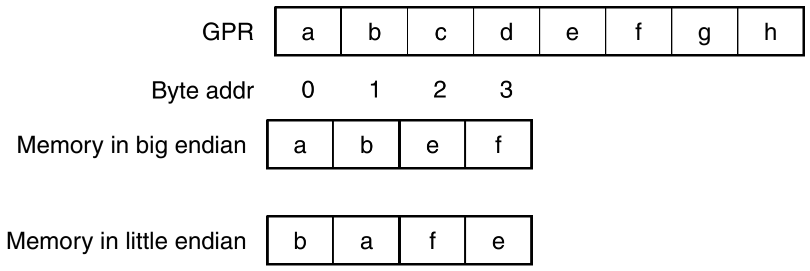
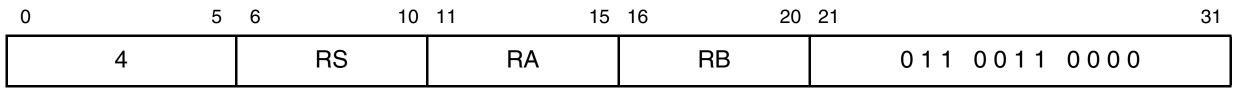


Figure 6-135. evstwhe results in big- and little-endian modes

evstwhex evstwhex

Vector Store Word of Two Halfwords from Even Indexed

evstwhex **r**S,**r**A,**r**B



```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
MEM(EA,2) = RS$_{0:15}$
MEM(EA+2,2) = RS$_{32:47}$
```

Figure 6-136 shows how bytes are stored in memory as determined by the endian mode.

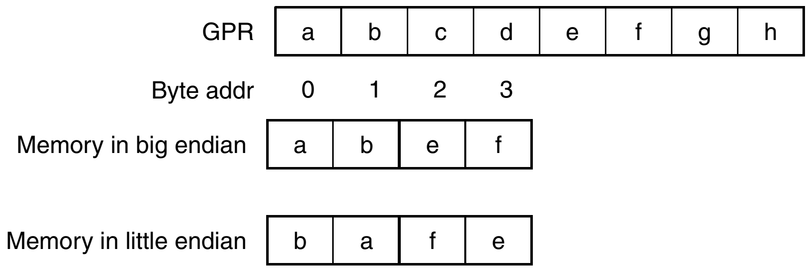


Figure 6-136. evstwhex results in big- and little-endian modes

evstwho

Vector Store Word of Two Halfwords from Odd

evstwho **rS,d(rA)**

0 – 5	6 – 10	11 – 15	16 – 20	21 – 31
4	RS	RA	UIMM[1]	0 1 1 0 0 1 1 0 1 0 1

[1] **d** = UIMM<<2

```
if (rA == 0) then b = 0
else b = (rA)
EA = b + EXTZ(UIMM*4)
MEM(EA,2) = $RS_{16:31}$
MEM(EA+2,2) = $RS_{48:63}$
```

Figure 6-137 shows how bytes are stored in memory as determined by the endian mode.

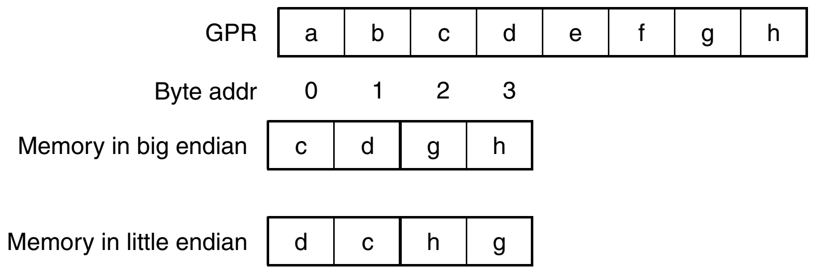
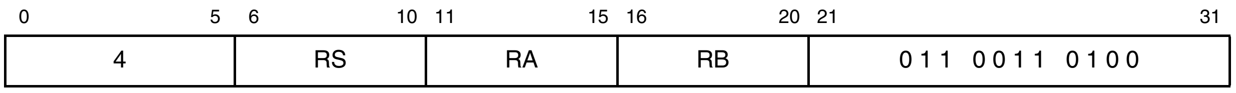


Figure 6-137. evstwho results in big- and little-endian modes

evstwhox evstwhox

Vector Store Word of Two Halfwords from Odd Indexed

evstwhox **rS,rA,rB**



```
if (rA == 0) then b = 0
else b = (rA)
EA = b + (rB)
MEM(EA,2) = RS$_{16:31}$
MEM(EA+2,2) = RS$_{48:63}$
```

Figure 6-138 shows how bytes are stored in memory as determined by the endian mode.

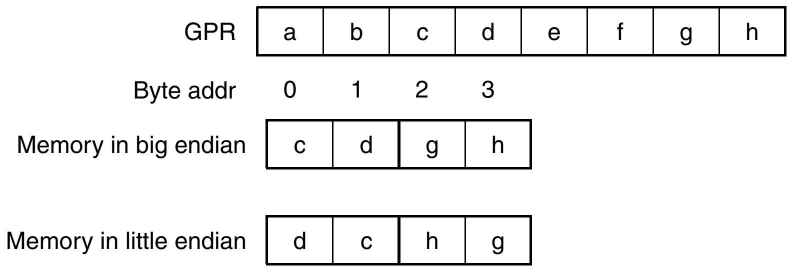


Figure 6-138. evstwhox results in big- and little-endian modes

6.6 SPE instruction timing

Instruction timing in number of processor clock cycles for SPE instructions are shown in Table 6-4, Table 6-5, and Table 6-6. Pipelined instructions are shown with cycles of total latency and throughput cycles. Divide instructions are not pipelined and block other instructions from executing during divide execution.

6.6.1 SPE integer simple instructions timing

Instruction timing for SPE integer simple instructions is shown in Table 6-4. The table is sorted by opcode. These instructions are issued as a pair of operations.

Table 6-4. Timing for integer simple instructions

Instruction	Latency	Throughput	Comments
brinc	1	1	—
evabs	1	1	—
evaddiw	1	1	—

Table 6-4. Timing for integer simple instructions (continued)

Instruction	Latency	Throughput	Comments
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	—
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	—

Table 6-4. Timing for integer simple instructions (continued)

Instruction	Latency	Throughput	Comments
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	—

6.6.2 SPE load and store instruction timing

Instruction timing for SPE load and store instructions is shown in Table 6-4. The table is sorted by opcode. Actual timing will depend on alignment; the table indicates timing for aligned operands.

Table 6-5. 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	—

Table 6-5. SPE load and store instruction timing (continued)

Instruction	Latency	Throughput	Comments
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	—
evstwhe	3	1	—
evstwhex	3	1	—
evstwho	3	1	—
evstwhox	3	1	—
evstwwe	3	1	—
evstwwex	3	1	—
evstwwo	3	1	—
evstwwox	3	1	—

6.6.3 SPE complex integer instruction timing

Instruction timing for SPE complex integer instructions is shown in Table 6-6. The table is sorted by opcode. For the divide instructions, the number of stall cycles is (latency) for following instructions.

Table 6-6. 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	4	1	—
evmhegsmfan	4	1	—
evmhegsmiaa	4	1	—
evmhegsmian	4	1	—
evmhegumiaa	4	1	—
evmhegumian	4	1	—
evmhesmf	4	1	—
evmhesmfa	4	1	—
evmhesmfaaw	4	1	—
evmhesmfanw	4	1	—
evmhesmi	4	1	—
evmhesmia	4	1	—
evmhesmiaaw	4	1	—
evmhesmianw	4	1	—
evmhessf	4	1	—
evmhessfa	4	1	—
evmhessfaaw	4	1	—
evmhessfanw	4	1	—
evmhessiaaw	4	1	—
evmhessianw	4	1	—
evmheumi	4	1	—

Table 6-6. SPE complex integer instruction timing (continued)

Instruction	Latency	Throughput	Comments
evmheumia	4	1	—
evmheumiaaw	4	1	—
evmheumianw	4	1	—
evmheusiaaw	4	1	—
evmheusianw	4	1	—
evmhogsmfaa	4	1	—
evmhogsmfan	4	1	—
evmhogsmiaa	4	1	—
evmhogsmian	4	1	—
evmhogumiaa	4	1	—
evmhogumian	4	1	—
evmhosmf	4	1	—
evmhosmfa	4	1	—
evmhosmfaaw	4	1	—
evmhosmfanw	4	1	—
evmhosmi	4	1	—
evmhosmia	4	1	—
evmhosmiaaw	4	1	—
evmhosmianw	4	1	—
evmhossf	4	1	—
evmhossfa	4	1	—
evmhossfaaw	4	1	—
evmhossfanw	4	1	—
evmhossiaaw	4	1	—
evmhossianw	4	1	—
evmhoumi	4	1	—
evmhoumia	4	1	—

Table 6-6. SPE complex integer instruction timing (continued)

Instruction	Latency	Throughput	Comments
evmhoumiaaw	4	1	—
evmhoumianw	4	1	—
evmhousiaaw	4	1	—
evmhousianw	4	1	—
evmra	4	1	—
evmwhsmf	4	1	—
evmwhsmfa	4	1	—
evmwhsmi	4	1	—
evmwhsmia	4	1	—
evmwhssf	4	1	—
evmwhssfa	4	1	—
evmwhumi	4	1	—
evmwhumia	4	1	—
evmwlsmiaaw	4	1	—
evmwlsmianw	4	1	—
evmwlssiaaw	4	1	—
evmwlssianw	4	1	—
evmwlumi	4	1	—
evmwlumia	4	1	—
evmwlumiaaw	4	1	—
evmwlumianw	4	1	—
evmwlusiaaw	4	1	—
evmwlusianw	4	1	—
evmwsmf	4	1	—
evmwsmfa	4	1	—
evmwsmfaa	4	1	—
evmwsmfan	4	1	—

Table 6-6. SPE complex integer instruction timing (continued)

Instruction	Latency	Throughput	Comments
evmwsmi	4	1	—
evmwsmia	4	1	—
evmwsmiaa	4	1	—
evmwsmian	4	1	—
evmwssf	4	1	—
evmwssfa	4	1	—
evmwssfaa	4	1	—
evmwssfan	4	1	—
evmwumi	4	1	—
evmwumia	4	1	—
evmwumiaa	4	1	—
evmwumian	4	1	—
evsubfsmiaaw	1	1	—
evsubfssiaaw	1	1	—
evsubfumiaaw	1	1	—
evsubfusiaaw	1	1	—

[1] Timing is data dependent

6.7 Instruction forms and opcodes

Table 6-7 gives the division of the opcode space for the new SPE instructions.

Table 6-7. Opcode space division

Opcode bits		Instruction Class
0–5	21–25	
4	0100*	SPE APU integer simple instructions
4	01010	EFPU floating-point instructions
4	01011	Embedded floating-point APU instructions
4	01100	SPE APU load/store instructions
4	01101	SPE APU reserved for future use

Table 6-7. Opcode space division (continued)

Opcode bits		Instruction Class
0–5	21–25	
4	0111*	SPE APU reserved for future use
4	10***	SPE APU integer complex instructions
4	11***	SPE APU integer complex instructions: reserved for future use

6.7.1 SPE vector integer simple instructions

For instructions that have signed and unsigned forms, bit 31 is 1 for the signed form and 0 for the unsigned form. For instructions that have immediate forms, bit 30 is 1 for immediate forms. All instructions have the destination register specified in the bits 6–10, which differs from Power Architecture ISA/Book E where some instructions have the destination in bits 11–15.

Table 6-8. Opcodes for integer simple instructions

Instruction	Opcode					Comments
	0–5	6–10	11–15	16–20	21–31	
brinc	4	rD	RA	rB	010 0000 1111	—
evabs	4	RD	RA	00000	010 0000 1000	—
evaddiw	4	RD	UIMM	RB	010 0000 0010	—
evaddw	4	RD	RA	RB	010 0000 0000	—
evand	4	RD	RA	RB	010 0001 0001	RD = RA & RB
evandc	4	RD	RA	RB	010 0001 0010	RD = RA & (~RB)
evcmpeq	4	crfD 00	RA	RB	010 0011 0100	—
evcmpgts	4	crfD 00	RA	RB	010 0011 0001	—
evcmpgtu	4	crfD 00	RA	RB	010 0011 0000	—
evcmplts	4	crfD 00	RA	RB	010 0011 0011	—
evcmpltu	4	crfD 00	RA	RB	010 0011 0010	—
evcntlsw	4	RD	RA	00000	010 0000 1110	—
evcntlzw	4	RD	RA	00000	010 0000 1101	—
eveqv	4	RD	RA	RB	010 0001 1001	RD = ~(RA XOR RB)
evextsb	4	RD	RA	00000	010 0000 1010	—
evextsh	4	RD	RA	00000	010 0000 1011	—

Table 6-8. Opcodes for integer simple instructions (continued)

Instruction	Opcode					Comments
	0–5	6–10	11–15	16–20	21–31	
evmergehi	4	RD	RA	RB	010 0010 1100	—
evmergehilo	4	RD	RA	RB	010 0010 1110	—
evmergelo	4	RD	RA	RB	010 0010 1101	—
evmergelohi	4	RD	RA	RB	010 0010 1111	—
evnand	4	RD	RA	RB	010 0001 1110	RD = ~(RA & RB)
evneg	4	RD	RA	00000	010 0000 1001	—
evnor	4	RD	RA	RB	010 0001 1000	RD = ~(RA | RB)
evor	4	RD	RA	RB	010 0001 0111	RD = RA | RB
evorc	4	RD	RA	RB	010 0001 1011	RD = RA | (~RB)
evrlw	4	RD	RA	RB	010 0010 1000	—
evrlwi	4	RD	RA	UIMM	010 0010 1010	—
evrndw	4	RD	RA	00000	010 0000 1100	—
evsel	4	RD	RA	RB	010 0111 1crfS	crfS is a 3-bit field
evslw	4	RD	RA	RB	010 0010 0100	—
evslwi	4	RD	RA	UIMM	010 0010 0110	—
evsplatfi	4	RD	SIMM	00000	010 0010 1011	—
evsplati	4	RD	SIMM	00000	010 0010 1001	—
evsrwis	4	RD	RA	UIMM	010 0010 0011	—
evsrwiu	4	RD	RA	UIMM	010 0010 0010	—
evsrws	4	RD	RA	RB	010 0010 0001	—
evsrwu	4	RD	RA	RB	010 0010 0000	—
evsubfw	4	RD	RA	RB	010 0000 0100	—
evsubifw	4	RD	UIMM	RB	010 0000 0110	—
evxor	4	RD	RA	RB	010 0001 0110	RD = RA XOR RB

6.7.2 Opcodes for SPE load and store instructions

Load instructions have a ‘0’ in bit 26 whereas all stores have a ‘1’ in bit 26. Bits 27 and 28 indicate the size of the data access to memory. Bit 31 indicates whether the index is immediate or the contents of a register. All store instructions have the source of the data register specified in bits 6:10 (RS).

Table 6-9. SPE load and store instruction opcodes

Instruction	Opcode bits					Comments
	0–5	6–10	11–15	16–20	21–31	
evldd	4	RD	RA	UIMM	011 0000 0001	—
evlddx	4	RD	RA	RB	011 0000 0000	—
evldh	4	RD	RA	UIMM	011 0000 0101	—
evldhx	4	RD	RA	RB	011 0000 0100	—
evldw	4	RD	RA	UIMM	011 0000 0011	—
evldwx	4	RD	RA	RB	011 0000 0010	—
evlhhesplat	4	RD	RA	UIMM	011 0000 1001	—
evlhhesplatx	4	RD	RA	RB	011 0000 1000	—
evlhhossplat	4	RD	RA	UIMM	011 0000 1111	—
evlhhossplatx	4	RD	RA	RB	011 0000 1110	—
evlhhousplat	4	RD	RA	UIMM	011 0000 1101	—
evlhhousplatx	4	RD	RA	RB	011 0000 1100	—
evlwhe	4	RD	RA	UIMM	011 0001 0001	—
evlwhex	4	RD	RA	RB	011 0001 0000	—
evlwhos	4	RD	RA	UIMM	011 0001 0111	—
evlwhosx	4	RD	RA	RB	011 0001 0110	—
evlwhou	4	RD	RA	UIMM	011 0001 0101	—
evlwhoux	4	RD	RA	RB	011 0001 0100	—
evlwhsplat	4	RD	RA	UIMM	011 0001 1101	—
evlwhsplatx	4	RD	RA	RB	011 0001 1100	—
evlwwsplat	4	RD	RA	UIMM	011 0001 1001	—
evlwwsplatx	4	RD	RA	RB	011 0001 1000	—

Table 6-9. SPE load and store instruction opcodes (continued)

Instruction	Opcode bits					Comments
	0–5	6–10	11–15	16–20	21–31	
evstdd	4	RS	RA	UIMM	011 0010 0001	—
evstddx	4	RS	RA	RB	011 0010 0000	—
evstdh	4	RS	RA	UIMM	011 0010 0101	—
evstdhx	4	RS	RA	RB	011 0010 0100	—
evstdw	4	RS	RA	UIMM	011 0010 0011	—
evstdwx	4	RS	RA	RB	011 0010 0010	—
evstwhe	4	RS	RA	UIMM	011 0011 0001	—
evstwhex	4	RS	RA	RB	011 0011 0000	—
evstwho	4	RS	RA	UIMM	011 0011 0101	—
evstwhox	4	RS	RA	RB	011 0011 0100	—
evstwwe	4	RS	RA	UIMM	011 0011 1001	—
evstwwex	4	RS	RA	RB	011 0011 1000	—
evstwwo	4	RS	RA	UIMM	011 0011 1101	—
evstwwox	4	RS	RA	RB	011 0011 1100	—

6.7.3 Opcodes for SPE complex integer instructions

Table 6-10. Opcodes for complex integer instructions, sorted by mnemonic

Instruction	Opcode bits				
	0–5	6–10	11–15	16–20	21–31
evaddsmiaaw	4	RD	RA	00000	100 1100 1001
evaddssiaaw	4	RD	RA	00000	100 1100 0001
evaddumiaaw	4	RD	RA	00000	100 1100 1000
evaddusiaaw	4	RD	RA	00000	100 1100 0000
evdivws	4	RD	RA	RB	100 1100 0110
evdivwu	4	RD	RA	RB	100 1100 0111
evmhegsmfaa	4	RD	RA	RB	101 0010 1011

Table 6-10. Opcodes for complex integer instructions, sorted by mnemonic (continued)

Instruction	Opcode bits				
	0–5	6–10	11–15	16–20	21–31
evmhegsmfan	4	RD	RA	RB	101 1010 1011
evmhegsmiaa	4	RD	RA	RB	101 0010 1001
evmhegsmian	4	RD	RA	RB	101 1010 1001
evmhegumiaa	4	RD	RA	RB	101 0010 1000
evmhegumian	4	RD	RA	RB	101 1010 1000
evmhesmf	4	RD	RA	RB	100 0000 1011
evmhesmfa	4	RD	RA	RB	100 0010 1011
evmhesmfaaw	4	RD	RA	RB	101 0000 1011
evmhesmfanw	4	RD	RA	RB	101 1000 1011
evmhesmi	4	RD	RA	RB	100 0000 1001
evmhesmia	4	RD	RA	RB	100 0010 1001
evmhesmiaaw	4	RD	RA	RB	101 0000 1001
evmhesmianw	4	RD	RA	RB	101 1000 1001
evmhessf	4	RD	RA	RB	100 0000 0011
evmhessfa	4	RD	RA	RB	100 0010 0011
evmhessfaaw	4	RD	RA	RB	101 0000 0011
evmhessfanw	4	RD	RA	RB	101 1000 0011
evmhessiaaw	4	RD	RA	RB	101 0000 0001
evmhessianw	4	RD	RA	RB	101 1000 0001
evmheumi	4	RD	RA	RB	100 0000 1000
evmheumia	4	RD	RA	RB	100 0010 1000
evmheumiaaw	4	RD	RA	RB	101 0000 1000
evmheumianw	4	RD	RA	RB	101 1000 1000
evmheusiaaw	4	RD	RA	RB	101 0000 0000
evmheusianw	4	RD	RA	RB	101 1000 0000
evmhogsmfaa	4	RD	RA	RB	101 0010 1111

Table 6-10. Opcodes for complex integer instructions, sorted by mnemonic (continued)

Instruction	Opcode bits				
	0–5	6–10	11–15	16–20	21–31
evmhogsmfan	4	RD	RA	RB	101 1010 1111
evmhogsmiaa	4	RD	RA	RB	101 0010 1101
evmhogsmian	4	RD	RA	RB	101 1010 1101
evmhogumiaa	4	RD	RA	RB	101 0010 1100
evmhogumian	4	RD	RA	RB	101 1010 1100
evmhosmf	4	RD	RA	RB	100 0000 1111
evmhosmfa	4	RD	RA	RB	100 0010 1111
evmhosmfaaw	4	RD	RA	RB	101 0000 1111
evmhosmfanw	4	RD	RA	RB	101 1000 1111
evmhosmi	4	RD	RA	RB	100 0000 1101
evmhosmia	4	RD	RA	RB	100 0010 1101
evmhosmiaaw	4	RD	RA	RB	101 0000 1101
evmhosmianw	4	RD	RA	RB	101 1000 1101
evmhossf	4	RD	RA	RB	100 0000 0111
evmhossfa	4	RD	RA	RB	100 0010 0111
evmhossfaaw	4	RD	RA	RB	101 0000 0111
evmhossfanw	4	RD	RA	RB	101 1000 0111
evmhossiaaw	4	RD	RA	RB	101 0000 0101
evmhossianw	4	RD	RA	RB	101 1000 0101
evmhoumi	4	RD	RA	RB	100 0000 1100
evmhoumia	4	RD	RA	RB	100 0010 1100
evmhoumiaaw	4	RD	RA	RB	101 0000 1100
evmhoumianw	4	RD	RA	RB	101 1000 1100
evmhousiaaw	4	RD	RA	RB	101 0000 0100
evmhousianw	4	RD	RA	RB	101 1000 0100
evmra	4	RD	RA	00000	100 1100 0100

Table 6-10. Opcodes for complex integer instructions, sorted by mnemonic (continued)

Instruction	Opcode bits				
	0–5	6–10	11–15	16–20	21–31
evmwhsmf	4	RD	RA	RB	100 0100 1111
evmwhsmfa	4	RD	RA	RB	100 0110 1111
evmwhsmi	4	RD	RA	RB	100 0100 1101
evmwhsmia	4	RD	RA	RB	100 0110 1101
evmwhssf	4	RD	RA	RB	100 0100 0111
evmwhssfa	4	RD	RA	RB	100 0110 0111
evmwhumi	4	RD	RA	RB	100 0100 1100
evmwhumia	4	RD	RA	RB	100 0110 1100
evmwlsmiaaw	4	RD	RA	RB	101 0100 1001
evmwlsmianw	4	RD	RA	RB	101 1100 1001
evmwlssiaaw	4	RD	RA	RB	101 0100 0001
evmwlssianw	4	RD	RA	RB	101 1100 0001
evmwlumi	4	RD	RA	RB	100 0100 1000
evmwlumia	4	RD	RA	RB	100 0110 1000
evmwlumiaaw	4	RD	RA	RB	101 0100 1000
evmwlumianw	4	RD	RA	RB	101 1100 1000
evmwlusiaaw	4	RD	RA	RB	101 0100 0000
evmwlusianw	4	RD	RA	RB	101 1100 0000
evmwsmf	4	RD	RA	RB	100 0101 1011
evmwsmfa	4	RD	RA	RB	100 0111 1011
evmwsmfaa	4	RD	RA	RB	101 0101 1011
evmwsmfan	4	RD	RA	RB	101 1101 1011
evmwsmi	4	RD	RA	RB	100 0101 1001
evmwsmia	4	RD	RA	RB	100 0111 1001
evmwsmiaa	4	RD	RA	RB	101 0101 1001
evmwsmian	4	RD	RA	RB	101 1101 1001

Table 6-10. Opcodes for complex integer instructions, sorted by mnemonic (continued)

Instruction	Opcode bits				
	0–5	6–10	11–15	16–20	21–31
evmwssf	4	RD	RA	RB	100 0101 0011
evmwssfa	4	RD	RA	RB	100 0111 0011
evmwssfaa	4	RD	RA	RB	101 0101 0011
evmwssfan	4	RD	RA	RB	101 1101 0011
evmwumi	4	RD	RA	RB	100 0101 1000
evmwumia	4	RD	RA	RB	100 0111 1000
evmwumiaa	4	RD	RA	RB	101 0101 1000
evmwumian	4	RD	RA	RB	101 1101 1000
evsubfsmiaaw	4	RD	RA	00000	100 1100 1011
evsubfssiaaw	4	RD	RA	00000	100 1100 0011
evsubfumiaaw	4	RD	RA	00000	100 1100 1010
evsubfusiaaw	4	RD	RA	00000	100 1100 0010

Table 6-11. Opcodes for complex integer instructions, sorted by opcode

Instruction	Opcode bits				
	0–5	6–10	11–15	16–20	21–31
evmhessf	4	RD	RA	RB	100 0000 0011
evmhossf	4	RD	RA	RB	100 0000 0111
evmheumi	4	RD	RA	RB	100 0000 1000
evmhesmi	4	RD	RA	RB	100 0000 1001
evmhesmf	4	RD	RA	RB	100 0000 1011
evmhoumi	4	RD	RA	RB	100 0000 1100
evmhosmi	4	RD	RA	RB	100 0000 1101
evmhosmf	4	RD	RA	RB	100 0000 1111
evmhessfa	4	RD	RA	RB	100 0010 0011
evmhossfa	4	RD	RA	RB	100 0010 0111

Table 6-11. Opcodes for complex integer instructions, sorted by opcode (continued)

Instruction	Opcode bits				
	0–5	6–10	11–15	16–20	21–31
evmheumia	4	RD	RA	RB	100 0010 1000
evmhesmia	4	RD	RA	RB	100 0010 1001
evmhesmfa	4	RD	RA	RB	100 0010 1011
evmhoumia	4	RD	RA	RB	100 0010 1100
evmhosmia	4	RD	RA	RB	100 0010 1101
evmhosmfa	4	RD	RA	RB	100 0010 1111
evmwhssf	4	RD	RA	RB	100 0100 0111
evmwlumi	4	RD	RA	RB	100 0100 1000
evmwhumi	4	RD	RA	RB	100 0100 1100
evmwhsmi	4	RD	RA	RB	100 0100 1101
evmwhsmf	4	RD	RA	RB	100 0100 1111
evmwssf	4	RD	RA	RB	100 0101 0011
evmwumi	4	RD	RA	RB	100 0101 1000
evmwsmi	4	RD	RA	RB	100 0101 1001
evmwsmf	4	RD	RA	RB	100 0101 1011
evmwhssfa	4	RD	RA	RB	100 0110 0111
evmwlumia	4	RD	RA	RB	100 0110 1000
evmwhumia	4	RD	RA	RB	100 0110 1100
evmwhsmia	4	RD	RA	RB	100 0110 1101
evmwhsmfa	4	RD	RA	RB	100 0110 1111
evmwssfa	4	RD	RA	RB	100 0111 0011
evmwumia	4	RD	RA	RB	100 0111 1000
evmwsmia	4	RD	RA	RB	100 0111 1001
evmwsmfa	4	RD	RA	RB	100 0111 1011
evaddusiaaw	4	RD	RA	00000	100 1100 0000
evaddssiaaw	4	RD	RA	00000	100 1100 0001

Table 6-11. Opcodes for complex integer instructions, sorted by opcode (continued)

Instruction	Opcode bits				
	0–5	6–10	11–15	16–20	21–31
evsubfusiaaw	4	RD	RA	00000	100 1100 0010
evsubfssiaaw	4	RD	RA	00000	100 1100 0011
evmra	4	RD	RA	00000	100 1100 0100
evdivws	4	RD	RA	RB	100 1100 0110
evdivwu	4	RD	RA	RB	100 1100 0111
evaddumiaaw	4	RD	RA	00000	100 1100 1000
evaddsmiaaw	4	RD	RA	00000	100 1100 1001
evsubfumiaaw	4	RD	RA	00000	100 1100 1010
evsubfsmiaaw	4	RD	RA	00000	100 1100 1011
evmheusiaaw	4	RD	RA	RB	101 0000 0000
evmhessiaaw	4	RD	RA	RB	101 0000 0001
evmhessfaaw	4	RD	RA	RB	101 0000 0011
evmhousiaaw	4	RD	RA	RB	101 0000 0100
evmhossiaaw	4	RD	RA	RB	101 0000 0101
evmhossfaaw	4	RD	RA	RB	101 0000 0111
evmheumiaaw	4	RD	RA	RB	101 0000 1000
evmhesmiaaw	4	RD	RA	RB	101 0000 1001
evmhesmfaaw	4	RD	RA	RB	101 0000 1011
evmhoumiaaw	4	RD	RA	RB	101 0000 1100
evmhosmiaaw	4	RD	RA	RB	101 0000 1101
evmhosmfaaw	4	RD	RA	RB	101 0000 1111
evmhegumiaa	4	RD	RA	RB	101 0010 1000
evmhegsmiaa	4	RD	RA	RB	101 0010 1001
evmhegsmfaa	4	RD	RA	RB	101 0010 1011
evmhogumiaa	4	RD	RA	RB	101 0010 1100
evmhogsmiaa	4	RD	RA	RB	101 0010 1101

Table 6-11. Opcodes for complex integer instructions, sorted by opcode (continued)

Instruction	Opcode bits				
	0–5	6–10	11–15	16–20	21–31
evmhogsmfaa	4	RD	RA	RB	101 0010 1111
evmwlusiaaw	4	RD	RA	RB	101 0100 0000
evmwlssiaaw	4	RD	RA	RB	101 0100 0001
evmwlumiaaw	4	RD	RA	RB	101 0100 1000
evmwlsmiaaw	4	RD	RA	RB	101 0100 1001
evmwssfaa	4	RD	RA	RB	101 0101 0011
evmwumiaa	4	RD	RA	RB	101 0101 1000
evmwsmiaa	4	RD	RA	RB	101 0101 1001
evmwsmfaa	4	RD	RA	RB	101 0101 1011
evmheusianw	4	RD	RA	RB	101 1000 0000
evmhessianw	4	RD	RA	RB	101 1000 0001
evmhessfanw	4	RD	RA	RB	101 1000 0011
evmhousianw	4	RD	RA	RB	101 1000 0100
evmhossianw	4	RD	RA	RB	101 1000 0101
evmhossfanw	4	RD	RA	RB	101 1000 0111
evmheumianw	4	RD	RA	RB	101 1000 1000
evmhesmianw	4	RD	RA	RB	101 1000 1001
evmhesmfanw	4	RD	RA	RB	101 1000 1011
evmhoumianw	4	RD	RA	RB	101 1000 1100
evmhosmianw	4	RD	RA	RB	101 1000 1101
evmhosmfanw	4	RD	RA	RB	101 1000 1111
evmhegumian	4	RD	RA	RB	101 1010 1000
evmhegsmian	4	RD	RA	RB	101 1010 1001
evmhegsmfan	4	RD	RA	RB	101 1010 1011
evmhogumian	4	RD	RA	RB	101 1010 1100
evmhogsmian	4	RD	RA	RB	101 1010 1101

Table 6-11. Opcodes for complex integer instructions, sorted by opcode (continued)

Instruction	Opcode bits				
	0–5	6–10	11–15	16–20	21–31
evmhogsmfan	4	RD	RA	RB	101 1010 1111
evmwlusianw	4	RD	RA	RB	101 1100 0000
evmwlssianw	4	RD	RA	RB	101 1100 0001
evmwlumianw	4	RD	RA	RB	101 1100 1000
evmwlsmianw	4	RD	RA	RB	101 1100 1001
evmwssfan	4	RD	RA	RB	101 1101 0011
evmwumian	4	RD	RA	RB	101 1101 1000
evmwsmian	4	RD	RA	RB	101 1101 1001
evmwsmfan	4	RD	RA	RB	101 1101 1011

Chapter 7
Interrupts and Exceptions

The *PowerISA 2.06* document defines the mechanisms by which the e200z759n3 core implements interrupts and exceptions. The document uses the terminology *Interrupt* as the action in which the processor saves its old context and begins execution at a pre-determined interrupt handler address. *Exceptions* are referred to as events that, when enabled, cause the processor to take an interrupt. This section uses the same terminology.

The Power Architecture exception mechanism allows the processor to change to supervisor state as a result of unusual conditions arising in the execution of instructions, and from external signals, bus errors, or various internal conditions. When interrupts occur, information about the state of the processor is saved to machine state save/restore registers (SRR0/SRR1, CSRR0/CSRR1, or DSRR0/DSRR1, MCSRR0/MCSRR1) and the processor begins execution at an address (interrupt vector) determined by the Interrupt Vector Prefix register (IVPR), and one of the Interrupt Vector Offset registers (IVOR). Processing of instructions within the interrupt handler begins in supervisor mode.

Multiple exception conditions can map to a single interrupt vector, and may be distinguished by examining registers associated with the interrupt. The Exception Syndrome register (ESR) is updated with information specific to the exception type when an interrupt occurs.

To prevent loss of state information, interrupt handlers must save the information stored in the machine state save/restore registers, soon after the interrupt has been taken. Four sets of these registers are implemented; SRR0 and SRR1 for non-critical interrupts, CSRR0 and CSRR1 for critical interrupts, DSRR0 and DSRR1 for debug interrupts (when the Debug APU is enabled), and MCSRR0 and MCSRR1 for machine check interrupts. Hardware supports nesting of critical interrupts within non-critical interrupts, machine check interrupts within both critical and non-critical interrupts, and debug interrupts within both critical, non-critical, and machine check interrupts. It is up to the interrupt handler to save necessary state information if interrupts of a given class are re-enabled within the handler.

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

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

Returning from an interrupt is performed by executing an **rfi**, **rfci**, **rfdi**, or **rfmci** instruction or **se_rfi**, **se_rfci**, **se_rfdi**, or **se_rfmci** VLE instruction to restore state information from the respective machine state save/restore register pair.

7.1 e200z759n3 interrupts

As specified by the *PowerISA 2.06* architecture, interrupts can be either precise or imprecise, synchronous or asynchronous, and critical or non-critical. Asynchronous exceptions are caused by events external to the processor's instruction execution; synchronous exceptions are directly caused by instructions or an event somehow synchronous to the program flow, such as a context switch. A precise interrupt architecturally guarantees that no instruction beyond the instruction causing the exception has (visibly) executed. Critical interrupts are provided with a separate save/restore register pair (CSRR0/CSRR1) to allow certain critical exceptions to be handled within a non-critical interrupt handler. Machine check interrupts are also provided with a separate save/restore register pair (MCSRR0/MCSRR1) to allow machine check exceptions to be handled within a non-critical or critical interrupt handler.

The types of interrupts handled are shown in Table 7-1. Refer to Chapter 7 of *Book E: Enhanced PowerPC*^{tm} *Architecture v0.99* for exact details of each interrupt type.

Table 7-1. Interrupt classifications

Interrupt types	Synchronous/asynchronous	Precise/imprecise	Critical/non-critical/ debug/ machine check
System Reset	Asynchronous, non-maskable	Imprecise	—
Machine Check	—	—	Machine Check
Non-Maskable Input interrupt	Asynchronous, non-maskable	Imprecise	Machine Check
Critical Input interrupt Watchdog Timer interrupt	Asynchronous, maskable	Imprecise	Critical
External Input Interrupt Fixed-Interval Timer interrupt Decrementer interrupt	Asynchronous, maskable	Imprecise	Non-critical
Performance Monitor interrupts	Synchronous/Asynchronous, maskable	Imprecise	Non-critical
Instruction-based Debug interrupts	Synchronous	Precise	Critical / Debug
Debug Interrupt (UDE) Debug Imprecise interrupt	Asynchronous	Imprecise	Critical / Debug
Data Storage / Alignment / TLB interrupts Instruction Storage / TLB interrupts	Synchronous	Precise	Non-critical

These classifications are discussed in greater detail in Section 7.7, Interrupt definitions. Interrupts implemented in e200z759n3 and the exception conditions that cause them are listed in Table 7-2.

Table 7-2. Exceptions and conditions

Interrupt type	Interrupt vector offset register	Causing conditions
System reset	none, vector to [p_rstbase[0:29]] || 2'b00	Reset by assertion of **p_reset_b**.
Critical Input	IVOR 0[1]	**p_critint_b** is asserted and MSR_{CE}=1.

Table 7-2. Exceptions and conditions (continued)

Interrupt type	Interrupt vector offset register	Causing conditions
Machine check	IVOR 1	• **p_mcp_b** transitions from negated to asserted • ISI, ITLB Error on first instruction fetch for an exception handler • Parity Error signaled on cache access • External bus error
Machine check (NMI)	IVOR 1	**p_nmi_b** transitions from negated to asserted.
Data Storage	IVOR 2	• Access control. • Byte ordering due to misaligned access across page boundary to pages with mismatched E bits • Cache locking exception
Instruction Storage	IVOR 3	• Access control. • Byte ordering due to misaligned instruction across page boundary to pages with mismatched VLE bits, or access to page with VLE set, and E indicating little-endian. • Misaligned Instruction fetch due to a change of flow to an odd halfword instruction boundary on a BookE (non-VLE) instruction page
External Input	IVOR 4[1]	**p_extint_b** is asserted and MSR_{EE}=1.
Alignment	IVOR 5	• **lmw**, **stmw** not word aligned • **lwarx** or **stwcx.** not word aligned, **lharx** or **sthcx.** not halfword aligned • **dcbz** with disabled cache, or to W or I storage • SPE ld and st instructions not properly aligned
Program	IVOR 6	Illegal, Privileged, Trap, AP enabled.
Floating-point unavailable	IVOR 7	Unused by e200z759n3.
System call	IVOR 8	Execution of the System Call (**sc, se_sc**) instruction
AP unavailable	IVOR 9	Unused by e200z759n3
Decrementer	IVOR 10	As specified in *Book E: Enhanced PowerPC^{tm} Architecture v0.99*, Ch. 8, pg. 190-191
Fixed Interval Timer	IVOR 11	As specified in *Book E: Enhanced PowerPC^{tm} Architecture v0.99*, Ch. 8, pg. 191-192
Watchdog Timer	IVOR 12	As specified in *Book E: Enhanced PowerPC^{tm} Architecture v0.99*, Ch. 8, pg. 192-194
Data TLB Error	IVOR 13	Data translation lookup did not match a valid entry in the TLB
Instruction TLB Error	IVOR 14	Instruction translation lookup did not match a valid entry in the TLB
Debug	IVOR 15	Trap, Instruction Address Compare, Data Address Compare, Instruction Complete, Branch Taken, Return from Interrupt, Interrupt Taken, Debug Counter, External Debug Event, Unconditional Debug Event
Reserved	IVOR 16-31	—

Table 7-2. Exceptions and conditions (continued)

Interrupt type	Interrupt vector offset register	Causing conditions
SPE/EFPU Unavailable Exception	IVOR 32	See Section 6.2.6.1, SPE APU Unavailable exception, and Section 5.2.5.1, EFPU unavailable exception
EFPU Data Exception	IVOR 33	See Section 5.2.5.2, Embedded floating-point data exception
EFPU Round Exception	IVOR 34	See Section 5.2.5.3, Embedded floating-point round exception
Performance Monitor	IVOR 35	Performance Monitor Enabled Condition or Event

1 Autovectored External and Critical Input interrupts use this IVOR. Vectored interrupts supply an interrupt vector offset directly.

7.2 Exception Syndrome Register (ESR)

The Exception Syndrome Register (ESR) provides a *syndrome* to differentiate between exceptions that can generate the same interrupt type. e200z759n3 adds some implementation specific bits to this register, as seen in Figure 7-1.

0				PIL	PPR	PTR	FP	ST	0	DLK	ILK	AP	PUO	BO	PIE	0								SPE	0	VLEMI	0			MIF	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

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

Figure 7-1. Exception Syndrome Register (ESR)

The ESR bits are defined in Table 7-3.

Table 7-3. ESR field descriptions

Bits	Name	Description	Associated interrupt type
0:3 (32:35)	—	Allocated[1]	—
4 (36)	PIL	Illegal Instruction exception (For e200z759n3, PIL used for all illegal/unimplemented instructions)	Program
5 (37)	PPR	Privileged Instruction exception	Program
6 (38)	PTR	Trap exception	Program

Table 7-3. ESR field descriptions (continued)

Bits	Name	Description	Associated interrupt type
7 (39)	FP	Floating-point operation	Alignment (not on Zen) Data Storage (not on Zen) Data TLB (not on Zen) Program
8 (40)	ST	Store operation	Alignment Data Storage Data TLB
9 (41)	—	Reserved[2]	—
10 (42)	DLK	Data Cache Locking	Data Storage
11 (43)	ILK	Instruction Cache Locking	Data Storage
12 (44)	AP	Auxiliary Processor operation (Not used by Zen)	Alignment (not on Zen) Data Storage (not on Zen) Data TLB (not on Zen) Program (not on Zen)
13 (45)	PUO	Unimplemented Operation exception (Not used by e200z759n3, PIL used for all illegal/unimplemented instructions)	Program
14 (46)	BO	Byte Ordering exception Mismatched Instruction Storage exception	Data Storage Instruction Storage
15 (47)	PIE	Program Imprecise exception (Reserved)	Currently unused by Zen
16:23 (48:55)	—	Reserved[2]	—
24 (56)	SPE	SPE/EFPU APU Operation	SPE/EFPU Unavailable EFPU Floating-point Data Exception EFPU Floating-point Round Exception Alignment Data Storage Data TLB
25 (57)	—	Allocated[1]	—

Table 7-3. ESR field descriptions (continued)

Bits	Name	Description	Associated interrupt type
26 (58)	VLEMI	VLE Mode Instruction	SPE/EFPU Unavailable EFPU Floating-point Data Exception EFPU Floating-point Round Exception Data Storage Data TLB Instruction Storage Alignment Program System Call
27:29 (59:61)	—	Allocated[1]	—
30 (62)	MIF	Misaligned Instruction Fetch	Instruction Storage Instruction TLB
31 (63)	—	Allocated[1]	—

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

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

7.3 Machine State Register (MSR)

The Machine State Register defines the state of the processor. The e200z759n3 MSR is shown in Figure 7-2.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0					UCLE	SPE	0						WE	CE	0	EE	PR	FP	ME	FE0	0	DE	FE1	0		IS	DS	0	PMM	RI	0

Read/ Write; Reset - 0x0

Figure 7-2. Machine State Register (MSR)

The MSR bits are defined in Table 7-4.

Table 7-4. MSR field descriptions

Bits	Name	Description
0:4 (32:36)	—	Reserved[1]
5 (37)	UCLE	User Cache Lock Enable 0 Execution of the cache locking instructions in user mode (MSR_{PR}=1) disabled; DSI exception taken instead, and ILK or DLK set in ESR. 1 Execution of the cache lock instructions in user mode enabled.

Table 7-4. MSR field descriptions (continued)

Bits	Name	Description
6 (38)	SPE	SPE/EFPU Available 0 Execution of SPE and EFPU APU vector instructions is disabled; SPE/EFPU Unavailable exception taken instead, and SPE bit is set in ESR. 1 Execution of SPE and EFPU APU vector instructions is enabled.
7:12 (39:44)	—	Reserved[1]
13 (45)	WE	Wait State (Power management) enable. This bit is defined as optional in the *PowerISA 2.06* architecture. 0 Power management is disabled. 1 Power management is enabled. The processor can enter a power-saving mode when additional conditions are present. The mode chosen is determined by the DOZE, NAP, and SLEEP bits in the HID0 register, described in Section 2.4.11, Hardware Implementation Dependent Register 0 (HID0).
14 (46)	CE	Critical Interrupt Enable 0 Critical Input and Watchdog Timer interrupts are disabled. 1 Critical Input and Watchdog Timer interrupts are enabled.
15 (47)	—	Reserved[1]
16 (48)	EE	External Interrupt Enable 0 External Input, Decrementer, and Fixed-Interval Timer interrupts are disabled. 1 External Input, Decrementer, and Fixed-Interval Timer interrupts are enabled.
17 (49)	PR	Problem State 0 The processor is in supervisor mode, can execute any instruction, and can access any resource (e.g. GPRs, SPRs, MSR, etc.). 1 The processor is in user mode, cannot execute any privileged instruction, and cannot access any privileged resource.
18 (50)	FP	Floating-Point Available 0 Floating point unit is unavailable. The processor cannot execute floating-point instructions, including floating-point loads, stores, and moves. 1 Floating-point unit is available. The processor can execute floating-point instructions. Note that for e200z759n3, the floating point unit is not supported in hardware, and an Illegal Instruction exception will be generated for attempted execution of *PowerISA 2.06* floating point instructions regardless of the setting of FP. FP is ignored, but cleared on exceptions.
19 (51)	ME	Machine Check Enable 0 Asynchronous Machine Check interrupts are disabled. 1 Asynchronous Machine Check interrupts are enabled.
20 (52)	FE0	Floating-point exception mode 0 (not used by Zen)
21 (53)	—	Reserved[1]
22 (54)	DE	Debug Interrupt Enable 0 Debug interrupts are disabled. 1 Debug interrupts are enabled.
23 (55)	FE1	Floating-point exception mode 1 (not used by Zen)

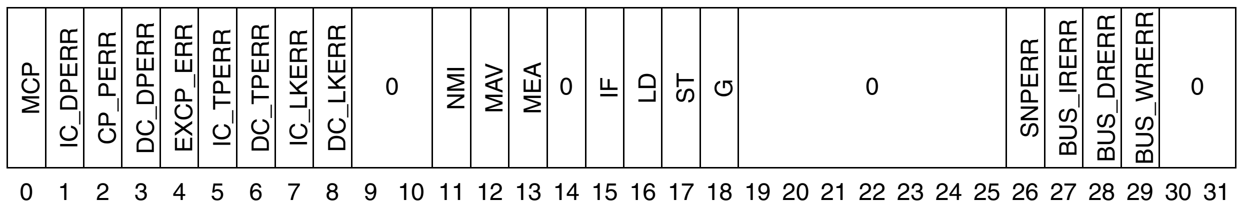
Table 7-4. MSR field descriptions (continued)

Bits	Name	Description
24 (56)	—	Reserved[1]
25 (57)	—	Preserved[1]
26 (58)	IS	Instruction Address Space 0 The processor directs all instruction fetches to address space 0 (TS=0 in the relevant TLB entry). 1 The processor directs all instruction fetches to address space 1 (TS=1 in the relevant TLB entry).
27 (59)	DS	Data Address Space 0 The processor directs all data storage accesses to address space 0 (TS=0 in the relevant TLB entry). 1 The processor directs all data storage accesses to address space 1 (TS=1 in the relevant TLB entry).
28 (60)	—	Reserved[1]
29 (61)	PMM	PMM Performance monitor mark bit. System software can set PMM when a marked process is running to enable statistics to be gathered only during the execution of the marked process. MSR_{PR} and MSR_{PMM} together define a state that the processor (supervisor or user) and the process (marked or unmarked) may be in at any time. If this state matches an individual state specified in the Performance Monitor registers PMLCa n, the state for which monitoring is enabled, counting is enabled.
30 (62)	RI	Recoverable Interrupt - This bit is provided for software use to detect nested exception conditions. This bit is cleared by hardware when a Machine Check interrupt is taken
31 (63)	—	Preserved[1]

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

7.3.1 Machine Check Syndrome Register (MCSR)

When the processor takes a machine check interrupt, it updates the Machine Check Syndrome register (MCSR) to differentiate between machine check conditions. The MCSR is shown in Figure 7-3.



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

Figure 7-3. Machine Check Syndrome Register (MCSR)

Table 7-5 describes MCSR fields. The MCSR indicates the source of a machine check condition. When an “Async Mchk” or “Error Report” syndrome bit in the MCSR is set, the core complex asserts **p_mcp_out** for system information.

All bits in the MCSR are implemented as “write ‘1’ to clear”. Software in the machine check handler is expected to clear the MCSR bits it has sampled prior to re-enabling MSR_{ME} to avoid a redundant machine check exception and to prepare for updated status bit information on the next machine check interrupt. Hardware will not clear a bit in the MCSR other than at reset. Software will typically sample MCSR early in the machine check handler, and will use the sampled value to clear those bits that were set at the time of sampling. Note that additional bits may become set during the handler after sampling if an asynchronous event occurs. By writing back only the originally sampled bits, another machine check can be generated to process the new conditions after the original handler re-enables MSR_{ME} either explicitly, or by restoring the MSR from MSRR1 at the return.

Note that any set bit in the MCSR other than status-type bits will cause a subsequent machine check interrupt once MSR_{ME}=1.

Table 7-5. MCSR field descriptions

Bit	Name	Description	Exception Type[1]	Recoverable
0 (32)	MCP	Machine check input pin	Async Mchk	Maybe
1 (33)	IC_DPERR	Instruction Cache data array parity error	Async Mchk	Precise
2 (34)	CP_PERR	Data Cache push parity error	Async Mchk	Unlikely
3 (35)	DC_DPERR	Data Cache data array parity error	Async Mchk	Maybe
4 (36)	EXCP_ERR	ISI, ITLB, or Bus Error on first instruction fetch for an exception handler	Async Mchk	Precise
5 (37)	IC_TPERR	Instruction Cache Tag parity error	Async Mchk	Precise
6 (38)	DC_TPERR	Data Cache Tag parity error	Async Mchk	Maybe
7 (39)	IC_LKERR	Instruction Cache Lock error Indicates a cache control operation or invalidation operation invalidated one or more locked lines in the ICache or encountered an uncorrectable lock error, or that an ICache miss with an uncorrectable lock error occurred. May also be set on locked line refill error.	Status	—
8 (40)	DC_LKERR	Data Cache Lock error Indicates a cache control operation or invalidation operation invalidated one or more locked lines in the DCache or encountered an uncorrectable lock error, or that an ICache miss with an uncorrectable lock error occurred. May also be set on locked line refill error.	Status	—

Table 7-5. MCSR field descriptions (continued)

Bit	Name	Description	Exception Type[1]	Recoverable
9:10 (41:42)	—	Reserved, should be cleared.		—
11 (43)	NMI	NMI input pin	NMI	—
12 (44)	MAV	MCAR Address Valid Indicates that the address contained in the MCAR was updated by hardware to correspond to the first detected Async Mchk error condition	Status	—
13 (45)	MEA	MCAR holds Effective Address If MAV=1,MEA=1 indicates that the MCAR contains an effective address and MEA=0 indicates that the MCAR contains a physical address	Status	—
14 (46)	—	Reserved, should be cleared.		—
15 (47)	IF	Instruction Fetch Error Report An error occurred during the attempt to fetch an instruction. This could be due to a parity error, or an external bus error. MCSRR0 contains the instruction address.	Error Report	Precise
16 (48)	LD	Load type instruction Error Report An error occurred during the attempt to execute the load type instruction located at the address stored in MCSRR0. This could be due to a parity error or an external bus error.	Error Report	Precise
17 (49)	ST	Store type instruction Error Report An error occurred during the attempt to execute the store type instruction located at the address stored in MCSRR0. This could be due to a parity error, or on certain external bus errors.	Error Report	Precise
18 (50)	G	Guarded instruction Error Report An error occurred during the attempt to execute the load or store type instruction located at the address stored in MCSRR0 and the access was guarded and encountered an error on the external bus.	Error Report	Precise
19:25 (51:57)	—	Reserved, should be cleared.		—
26 (58)	SNPERR	Snoop Lookup Error An error occurred during certain snoop operations. This is typically due to a data cache tag parity error, in which case DC_TPERR will also be set.	Async Mchk	Unlikely?
27 (59)	BUS_IRERR	Read bus error on Instruction fetch or linefill	Async Mchk	Precise if data used
28 (60)	BUS_DRERR	Read bus error on data load or linefill	Async Mchk	Precise if data used

Table 7-5. MCSR field descriptions (continued)

Bit	Name	Description	Exception Type[1]	Recoverable
29 (61)	BUS_WRERR	Write bus error on store or cache line push	Async Mchk	Unlikely
30:31 (62:63)	—	Reserved, should be cleared.	—	—

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

- "Error Report" indicates that this bit is only set for error report exceptions that cause machine check interrupts. These bits are only updated when the machine check interrupt is actually taken. Error report exceptions are not gated by MSR_{ME}. These are synchronous exceptions. These bits will remain set until cleared by software writing a "1" to the bit position(s) to be cleared.

- "Status" indicates that this bit is provides additional status information regarding the logging of a machine check exception. These bits will remain set until cleared by software writing a "1" to the bit position(s) to be cleared.

- "NMI" indicates that this bit is only set for the non-maskable interrupt type exception that causes a machine check interrupt. This bit is only updated when the machine check interrupt is actually taken. NMI exceptions are not gated by MSR_{ME}. This is an asynchronous exception. This bit will remain set until cleared by software writing a "1" to the bit position.

- "Async Mchk" indicates that this bit is set for an asynchronous machine check exception. These bits are set immediately upon detection of the error. Once any "Async Mchk" bit is set in the MCSR, a machine check interrupt will occur if MSR_{ME}=1. If MSR_{ME}=0, the machine check exception will remain pending. These bits will remain set until cleared by software writing a "1" to the bit position(s) to be cleared.

7.4 Interrupt Vector Prefix Registers (IVPR)

The Interrupt Vector Prefix Register is used during interrupt processing for determining the starting address of a software handler used to handle an interrupt. The value contained in the Vector Offset field of the IVOR selected for a particular interrupt type is concatenated with the Vector Base value held in the Interrupt Vector Prefix register (IVPR) to form an instruction address from which execution is to begin. The format of IVPR is shown in Figure 7-4.

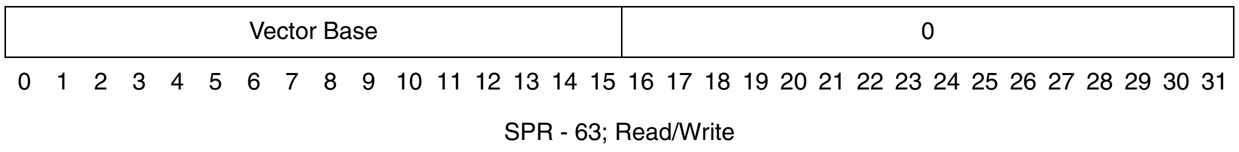


Figure 7-4. e200z759n3 Interrupt Vector Prefix Register (IVPR)

The IVPR fields are defined in Table 7-6.

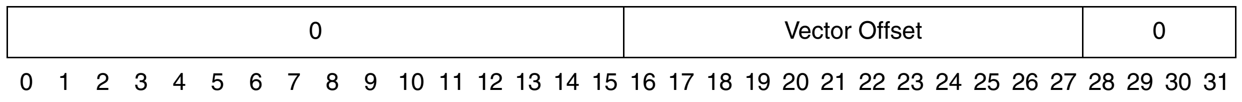
Table 7-6. IVPR field descriptions

Bits	Name	Description
0:15 (32:47)	Vec Base	Vector Base This field is used to define the base location of the vector table, aligned to a 64 KB boundary. This field provides the high-order 16 bits of the location of all interrupt handlers. The contents of the IVORxx register appropriate for the type of exception being processed are concatenated with the IVPR Vector Base to form the address of the handler in memory.
16:31 (48:63)	—	Reserved[1]

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

7.5 Interrupt Vector Offset Registers (IVORxx)

The Interrupt Vector Offset Registers are used during interrupt processing for determining the starting address of a software handler used to handle an interrupt. The value contained in the Vector Offset field of the IVOR selected for a particular interrupt type is concatenated with the value held in the Interrupt Vector Prefix register (IVPR) to form an instruction address from which execution is to begin. The format of a e200z759n3 IVOR is shown in Figure 7-5.



SPR - 400-415, 528-530; Read/Write

Figure 7-5. e200z759n3 Interrupt Vector Offset Register (IVOR)

The IVOR fields are defined in Table 7-7.

Table 7-7. IVOR field descriptions

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

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

7.6 Hardware Interrupt Vector Offset Values (p_voffset[0:15])

The **p_voffset[0:15]** input signals provide a hardware vector offset to be used when exception processing begins for an incoming interrupt request. These signals are sampled along with the **p_extint_b** and **p_critint_b** interrupt request inputs, and must be driven to a valid value when either of these signals is asserted unless the **p_avec_b** signal is also asserted. If **p_avec_b** is asserted, these inputs are not used. **p_voffset[0:11]** are used in forming the exception handler address, and **p_voffset[12:15]** are reserved and should be driven low.

7.7 Interrupt definitions

7.7.1 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 e200z759n3 detects the exception, if the exception is enabled by MSR_{CE}, e200z759n3 takes the Critical Input interrupt. The **p_critint_b** input is a level-sensitive signal expected to remain asserted until e200z759n3 acknowledges the interrupt. If **p_critint_b** is negated early, recognition of the interrupt request is not guaranteed. After e200z759n3 begins execution of the critical interrupt handler, the system can safely negate **p_critint_b**.

A Critical Input interrupt may be delayed by other higher priority exceptions or if MSR_{CE} is cleared when the exception occurs.

Table 7-8 lists register settings when a Critical Input interrupt is taken.

Table 7-8. 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 PMM 0 RI —
ESR	Unchanged		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	$IVPR_{0:15}$ || $IVOR0_{16:27}$ || 4b0000 (autovectored) $IVPR_{0:15}$ || **p_voffset[0:11]** || 4b0000 (non-autovectored)		

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

When the Debug APU is enabled, the MSR_{DE} bit is not automatically cleared by a Critical Input interrupt, but can be configured to be cleared via the HID0 register ($HID0_{CICLRDE}$). Refer to Section 2.4.11, Hardware Implementation Dependent Register 0 (HID0).

IVOR0 is the vector offset register used by autovectored Critical Input interrupts to determine the interrupt handler location. e200z759n3 also provides the capability to directly vector Critical Input interrupts to multiple handlers by allowing a Critical Input interrupt request to be accompanied by a vector offset. The

p_voffset[0:11] input signals are used in place of the value in IVOR0 to form the interrupt vector when a Critical Input interrupt request is not autovectored (**p_avec_b** negated when **p_critint_b** asserted).

7.7.2 Machine Check interrupt (IVOR1)

e200z759n3 implements the Machine Check exception as defined in the *Freescale EIS* Machine Check APU except for automatic clearing of the MSR_{DE} bit (see later paragraph). This behavior is different from the definition in *PowerISA 2.06*. e200z759n3 initiates a Machine Check interrupt if any of the machine check sources listed in Table 7-2 is detected.

As defined in *Freescale EIS* Machine Check APU, a machine check interrupt is taken for error report and NMI type machine check conditions even if MSR_{ME} is cleared, without the processor generating an internal checkstop condition. Processing of asynchronous type machine check sources (the sources reflected in the MCSR “async mchk” syndrome bits) is gated by MSR_{ME}.

The *Freescale EIS* Machine Check APU defines a separate set of save/restore registers (MCSRR0/1), a Machine Check Syndrome register (MCSR) to record the source(s) of machine checks, and a Machine Check Address register (MCAR) to hold an address associated with a machine check for certain classes of machine checks. Return from Machine Check instructions (**rfmci**, **se_rfmci**) are also provided to support returns using MCSRR0/1.

The MSR_{RI} status bit is provided for software use in determining if multiple nested machine check exceptions have occurred. Software may interrogate the $MCSRR1_{RI}$ bit to determine if a machine check occurred during the initial portion of a machine check handler prior to handler code, which sets MSR_{RI} to ‘1’ to indicate that the handler can now tolerate another machine check condition without losing state necessary for recovery.

The MSR_{DE} bit is not automatically cleared by a Machine Check exception, but can be configured to be cleared or left unchanged via the HID0 register ($HID0_{MCCLRDE}$). Refer to Section 2.4.11, Hardware Implementation Dependent Register 0 (HID0).

7.7.2.1 Machine check causes

Machine check causes are divided into different types:

- Error Report Machine Check conditions
- Non-Maskable Interrupt (NMI) machine check exceptions
- Asynchronous machine check exceptions

This division is intended to facilitate machine check handling in uni-processor, multiprocessor and multi-threaded systems. Although the initial implementation of the e200z759n3 does not implement multithreading, future versions are expected to, and the machine check model will remain compatible. In addition, the model is equally applicable to a single-threaded design.

7.7.2.1.1 Error report machine check exceptions

Error report machine check exceptions are directly associated with the current instruction execution stream, and are presented to the interrupt mechanism in a manner analogous to an Instruction storage or data storage interrupt. Since the execution stream cannot continue execution without suffering from corruption of architectural state, these exceptions are not masked by MSR_{ME}. Error report machine check exceptions are not necessarily recoverable if they occur during the initial portion of a machine check handler. The MSR_{RI} and $MCSRR1_{RI}$ bits are provided to assist software in determining recoverability.

For error report machine check exceptions, the MCSR (Machine Check Status Register) is updated only when the machine check interrupt is actually taken. The MCAR is not updated for error report machine check exceptions.

Error report machine check exceptions encountered by program execution can be flushed if an older exception exists or if an asynchronous interrupt or machine check is taken before the instruction that encountered the error becomes the oldest instruction in the machine. In this case the corresponding MCSR bit will not be set due to the flushed exception condition (although the corresponding bit may have already been set by a previous instruction's exception). Note that an async machine check condition may occur for the same error condition prior to the error report machine check, and the error report machine check may be discarded.

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

— $MCSR_{IF}$ will be set if the error occurred during an instruction fetch.

— $MCSR_{LD}$ will be set if the error occurred for a load instruction. If the error occurred for a guarded load and the error source was from the external bus, $MCSR_{G}$ will also be set.

— $MCSR_{ST}$ will be set if the error occurred in the data cache (parity) or MMU (DTLB Error or DSI) for a store type instruction (including **dcbz**), if an external termination error was received on a cache-inhibited guarded store or on a store conditional instruction, or if an unsuccessful flush with invalidation occurs on a store conditional instruction due to a tag or data parity error or external bus error. If an external termination error occurred on a cache-inhibited guarded store, or on a guarded store conditional, $MCSR_{G}$ will also be set.

Note that most (if not all) error report machine check exceptions will be accompanied by an associated asynchronous machine check exception on a single-threaded e200z759n3, although this will not generally be the case for a multi-threaded version.

Table 7-9. Error report machine check exceptions

Synchronous machine check source	Error type	MCSR updates	Precise[1]
Instruction Fetch	(ICache tag array parity error or data array parity error) & L1CSR1$_{ICEA}$='00'	IF	yes
	(ICache uncorrectable tag array parity error & L1CSR1$_{ICEA}$='01' & line potentially locked (locked or lock parity error) was invalidated	IF	yes
	cacheable miss & L1CSR1$_{ICEA}$='00' & any line with lock parity error	IF	yes
	cacheable miss & L1CSR1$_{ICEA}$='01' & and line with uncorrectable lock parity error was invalidated	IF	yes
	External termination error	IF	yes
Load instruction	(DCache tag array parity error or data array parity error) & L1CSR0$_{DCEA}$='00'	LD	yes
	(DCache uncorrectable tag array parity error or data array parity error) & L1CSR0$_{DCEA}$='01' & (line potentially locked (locked or lock parity error) was invalidated, or line potentially dirty (dirty or dirty parity error))	LD	yes
	cacheable miss & L1CSR0$_{DCEA}$='00' & any line with lock parity error, or dirty parity error on replacement line	LD	yes
	cacheable miss & L1CSR0$_{DCEA}$='01' & line with uncorrectable lock parity error was invalidated	LD	yes
	External termination error on load data	LD, [G][2]	yes
Load and reserve instruction	DCache tag array parity error & L1CSR0$_{DCEA}$='00'	LD	yes
	DCache hit and dirty parity error & L1CSR0$_{DCEA}$='00'	LD	yes
	(DCache uncorrectable tag array parity error or data array parity error) & L1CSR0$_{DCEA}$='01' & line potentially dirty (dirty or dirty parity error)	LD	yes
	DCache data push parity error[3]	LD	yes
	External termination error on dirty push[3]	LD	yes
	External termination error on load	LD, [G][2]	yes

Table 7-9. Error report machine check exceptions (continued)

Synchronous machine check source	Error type	MCSR updates	Precise[1]
Store instruction	DCache tag array parity error & L1CSR0$_{DCEA}$=‘00’	ST	yes
	DCache uncorrectable tag array parity error & L1CSR0$_{DCEA}$=‘01’ & (line potentially locked (locked or lock parity error) was invalidated, or line potentially dirty (dirty or dirty parity error))	ST	yes
	cacheable miss & L1CSR0$_{DCEA}$=’00’ & any line with lock parity error, or dirty parity error on replacement line	ST	yes
	cacheable miss & L1CSR0$_{DCEA}$=’01’ & line with uncorrectable lock parity error was invalidated	ST	yes
	External termination error on CI+G store[4]	ST, G	yes
Store conditional instruction	DCache tag array parity error & L1CSR0$_{DCEA}$=‘00’	ST	yes
	DCache hit and dirty parity error & L1CSR0$_{DCEA}$=‘00’	ST	yes
	DCache uncorrectable tag array parity error & L1CSR0$_{DCEA}$=‘01’ & line potentially dirty (dirty or dirty parity error)	ST	yes
	DCache data push parity error[5]	ST	yes
	External termination error on dirty push[5]	ST	yes
	External termination error on store conditional	ST, [G][6]	yes
dcbst instruction	DCache tag array parity error & miss & L1CSR0$_{DCEA}$=‘00’ & any line with error is potentially dirty (dirty or dirty parity error)	LD	yes
	DCache uncorrectable tag array parity error & cacheable miss & L1CSR0$_{DCEA}$=‘01’ & line potentially dirty (dirty or dirty parity error)	LD	yes
dcbf instruction	DCache tag array parity error & miss & L1CSR0$_{DCEA}$=‘00’ & (line potentially locked (locked or lock parity error) or line potentially dirty (dirty or dirty parity error))	LD	yes
	DCache uncorrectable tag array parity error & miss & L1CSR0$_{DCEA}$=‘01’ & (line potentially locked (locked or lock parity error) or line potentially dirty (dirty or dirty parity error))	LD	yes

Table 7-9. Error report machine check exceptions (continued)

Synchronous machine check source	Error type	MCSR updates	Precise[1]
dcblc instruction	DCache tag array parity error & cacheable miss & L1CSR0$_{DCEA}$=‘00’ & line potentially locked (locked or lock parity error)	LD	yes
	DCache uncorrectable tag array parity error & cacheable miss & L1CSR0$_{DCEA}$=‘01’ & line potentially locked (locked or lock parity error)	LD	yes
dcbtls, dcbtstls instruction	(DCache tag array parity error or lock error) & miss & L1CSR0$_{DCEA}$=‘00’	LD	yes
	DCache uncorrectable tag array parity error & cacheable miss & L1CSR0$_{DCEA}$=‘01’ & (line potentially locked (locked or lock parity error) was invalidated, or line potentially dirty (dirty or dirty parity error))	LD	yes
	cacheable miss & L1CSR0$_{DCEA}$=’00’ & any line with lock parity error, or dirty parity error on replacement line	LD	yes
	cacheable miss & L1CSR0$_{DCEA}$=’01’ & line with uncorrectable lock parity error was invalidated	LD	yes
	External termination error on linefill	LD, [G][2]	yes
dcbz instruction[7]	(DCache tag array parity error or lock error) & cacheable miss & L1CSR0$_{DCEA}$=‘00’	ST	yes
	DCache uncorrectable tag array parity error & cacheable miss & L1CSR0$_{DCEA}$=‘01’ & (line potentially locked (locked or lock parity error) was invalidated, or line potentially dirty (dirty or dirty parity error))	ST	yes
	cacheable miss & L1CSR0$_{DCEA}$=’00’ & any line with lock parity error, or dirty parity error on replacement line	ST	yes
dcbz instruction[7]	cacheable miss & L1CSR0$_{DCEA}$=’01’ & line with uncorrectable lock parity error was invalidated	ST	yes
L1FINV0 flush or flush with invalidate operation	DCache tag parity error & L1CSR0$_{DCEA}$=‘00’and line potentially dirty (dirty or dirty parity error)	LD	yes
	DCache uncorrectable tag parity error & L1CSR0$_{DCEA}$=‘01’and line potentially dirty (dirty or dirty parity error)		

Table 7-9. Error report machine check exceptions (continued)

Synchronous machine check source	Error type	MCSR updates	Precise[1]
icblc instruction	ICache tag array parity error & cacheable miss & L1CSR1$_{ICEA}$='00' & line potentially locked (locked or lock parity error)	IF	yes
	ICache uncorrectable tag array parity error & cacheable miss & L1CSR1$_{ICEA}$='01' & line potentially locked (locked or lock parity error) was invalidated	IF	yes
icbtls instruction	(ICache tag array parity error or lock error) & cacheable miss & L1CSR1$_{ICEA}$='00'	IF	yes
	ICache uncorrectable tag array parity error & cacheable miss & L1CSR1$_{ICEA}$='01' & line potentially locked (locked or lock parity error) was invalidated	IF	yes
	External termination error on linefill	IF	yes
Exception vectoring	ISI, ITLB, or Bus Error on first instruction fetch for an exception handler	IF	yes

1 MCSRR0 will point to the instruction associated with the machine check condition
2 G will be set if the load was a guarded load.
3 Can only occur if the load and reserve causes a dirty line to be flushed
4 Only reported if the store was a cache-inhibited guarded store
5 Can only occur if the store conditional causes a dirty line to be flushed
6 Only reported if the store was a guarded store.
7 Alignment error may be generated concurrently

7.7.2.1.2 Non-maskable interrupt machine check exceptions

Non-maskable interrupt exceptions are reported via the **p_nmi_b** input pin, which is transition sensitive. NMI exceptions are not gated by MSR$_{ME}$, thus are not necessarily recoverable if an NMI exception occurs during the initial part of a machine check exception handler. The MSR$_{RI}$ and MCSRR1$_{RI}$ bits are provide to assist software in determining recoverability.

For NMI machine check exceptions, MCSR$_{NMI}$ is updated (set) only when the machine check interrupt is actually taken. Hardware does not clear the MCSR$_{NMI}$ syndrome bit. Software is responsible for clearing this syndrome bit by writing a '1' to the bit(s) to be cleared. Hardware will not clear an NMI bit once it is set.

The MCAR is not updated for NMI machine check exceptions.

7.7.2.1.3 Asynchronous machine check exceptions

The remainder of machine check exceptions are classified as asynchronous machine check exceptions, as they are reported directly by the subsystem or resource that detected the condition. For many cases, the asynchronous condition will be reported simultaneously with a corresponding error report condition. These conditions are reported by immediately setting the corresponding MCSR "async mchk" syndrome bit, regardless of the state of MSR_{ME}. Interrupts due to asynchronous machine check exceptions are gated by MSR_{ME}. If MSR_{ME}=0 at the time an async mchk bit becomes set, the interrupt will be postponed until MSR_{ME} is later set to ‘1’ (although a machine check interrupt may occur at the time of the event due to an error report exception). Asynchronous events are cumulative; hardware does not clear an async mchk syndrome bit. Software is responsible for clearing these syndrome bits by writing a ‘1’ to the bit(s) to be cleared. Hardware will not clear an async mchk bit once it is set.

If $MCSR_{MAV}$ is cleared at the time an asynchronous machine check exception occurs that has a corresponding address (either an effective or real address) to log in the MCAR, then the MCAR and the $MCSR_{MEA}$ bit are updated, and the $MCSR_{MAV}$ bit is set. If $MCSR_{MAV}$ was previously set, then the MCAR and the $MCSR_{MEA}$ bit are not affected.

Table 7-10 details all asynchronous machine check sources.

Table 7-10. Asynchronous machine check exceptions

Asynchronous machine check source	**Transaction source**	**Error type**	MCSR update[1]		**MCAR update[2]**
External	n/a	Machine Check Input Pin[3]	MCP		none
Instruction Cache	Instruction Fetch	Tag array parity error & $L1CSR1_{ICEA}$=00	MAV	IC_TPERR	RA
		ICache hit, data array parity error & $L1CSR1_{ICEA}$=00		IC_DPERR	RA
		ICache cacheable miss, lock error, & $L1CSR1_{ICEA}$=00		IC_TPERR, IC_LKERR	RA
		$L1CSR1_{ICEA}$=01 & auto-invalidation of locked or potentially locked line due to uncorrectable tag parity error		IC_TPERR, IC_LKERR	RA
	icblc	Tag array parity error & cacheable miss & $L1CSR1_{ICEA}$=00 & line potentially locked (locked or lock parity error)		IC_TPERR, [IC_LKERR (if lock parity error)]	RA
	icbtls	(Tag array parity error or lock error) & cacheable miss & $L1CSR1_{ICEA}$=00		IC_TPERR, [IC_LKERR (if lock parity error)]	RA
	icblc icbtls	$L1CSR1_{ICEA}$=01 & Auto-invalidation of locked line due to uncorrectable tag parity error		IC_TPERR, IC_LKERR	RA
Data Cache	dcblc	Tag array parity error & cacheable miss & $L1CSR0_{DCEA}$=00 & line potentially locked (lock or lock parity error)	MAV	DC_TPERR, [DC_LKERR (if lock parity error)]	RA

Table 7-10. Asynchronous machine check exceptions (continued)

Asynchronous machine check source	Transaction source	Error type	MCSR update[1]		MCAR update[2]
Data Cache	load or store	Tag array parity error & L1CSR0$_{DCEA}$=00	MAV	DC_TPERR, [DC_LKERR (if lock parity error on line with tag parity error)]	RA
	L1FINV0 flush or flush w/inv & line dirty or potentially dirty	Tag array parity error & L1CSR0$_{DCEA}$=00		DC_TPERR	RA
	dcbtls dcbtstls dcbz	Tag array parity error & cacheable miss & L1CSR0$_{DCEA}$=00		DC_TPERR	RA
	dcbf	Tag array parity error & miss & L1CSR0$_{DCEA}$=00 & (line potentially locked (locked or lock parity error) or line potentially dirty (dirty or dirty parity error))		DC_TPERR, [DC_LKERR (if lock parity error)]	RA
	atomic load or store	Hit & L1CSR0$_{DCEA}$=00 & line has dirty parity error		DC_TPERR	RA
	dcbst, atomic load or store	Tag array parity error & miss & L1CSR0$_{DCEA}$=00 & line potentially dirty (dirty or dirty parity error)		DC_TPERR, [DC_LKERR (if lock parity error)]	RA
	load or store dcbtls dcbtstls dcbz	DCache cacheable miss & L1CSR0$_{DCEA}$='00' & lock parity error		DC_TPERR, DC_LKERR	RA
	load or store dcbtls dcbtstls dcbz	DCache cacheable miss & L1CSR0$_{DCEA}$='00' & dirty parity error on line to be replaced		DC_TPERR	RA
	load or store dcbtls dcbtstls dcbz	DCache uncorrectable tag array parity error & L1CSR0$_{DCEA}$='01' & (line potentially locked (locked or lock parity error) was invalidated, or line potentially dirty (dirty or dirty parity error))		DC_TPERR, [DC_LKERR]	RA

Table 7-10. Asynchronous machine check exceptions (continued)

Asynchronous machine check source	Transaction source	Error type	MCSR update[1]		MCAR update[2]
Data Cache	L1FINV0 flush w/inv	DCache uncorrectable tag array parity error & L1CSR0$_{DCEA}$=‘01’ & line potentially dirty (dirty or dirty parity error))	MAV	DC_TPERR	RA
	dcblc	DCache uncorrectable tag array parity error & L1CSR0$_{DCEA}$=‘01’ & (line potentially locked (locked or lock parity error) was invalidated		DC_TPERR, [DC_LKERR]	RA
	dcbst, atomic load or store	DCache uncorrectable tag array parity error & L1CSR0$_{DCEA}$=‘01’ & line potentially dirty (dirty or dirty parity error)		DC_TPERR, [DC_LKERR (if uncorrectable lock parity error)]	RA
	dcbf	DCache uncorrectable tag array parity error & L1CSR0$_{DCEA}$=‘01’ & (line potentially locked (locked or lock parity error) or line potentially dirty (dirty or dirty parity error))		DC_TPERR, [DC_LKERR (if uncorrectable lock parity error)]	RA
	L1FINV0 flush	DCache uncorrectable tag array parity error & L1CSR0$_{DCEA}$=‘01’ & line potentially dirty (dirty or dirty parity error)		DC_TPERR	RA
	load	DCache hit, data array parity error & L1CSR0$_{DCEA}$=00		DC_DPERR	RA
		DCache hit, data array parity error & L1CSR0$_{DCEA}$=‘01’ & line potentially dirty (dirty or dirty parity error)		DC_DPERR	RA
	replacement push dcbf push dcbst push L1FINV0 push reservation instruction forced-push	Data array push parity error		CP_PERR	RA

Table 7-10. Asynchronous machine check exceptions (continued)

Asynchronous machine check source	Transaction source	Error type	MCSR update[1]		MCAR update[2]
Data Cache	snoop lookup	Tag array parity error & (cacheable miss, or hit only to way with tag parity error)	MAV	DC_TPERR, SNPERR	RA (snoop address)
BIU	store or push	Bus error on write or push	MAV	BUS_WRERR	RA
	load store/w allocate dcbtls dcbtstls	Bus error on load fetch or linefill		BUS_DRERR	RA
	load	Bus error on error recovery refill		BUS_DRERR	RA
	instruction fetch	Bus error on error recovery refill		BUS_IRERR	RA
	icbtls CI or cache disabled Ifetch	Bus error on icbtls fill Bus error on CI Ifetch Bus error on cache disabled Ifetch		BUS_IRERR	RA
	load	Bus error on locked line error recovery refill		BUS_DRERR, DC_LKERR	RA
	instruction fetch	Bus error on locked line error recovery refill		BUS_IRERR, IC_LKERR	RA
Snoop Lookup	INV snoop command type	Tag array parity error & (miss, or hit only to way with tag parity error)	MAV	SNPERR, DC_TPERR	RA[4]
Exception Vectoring	first instruction fetch for an exception handler	ISI or Bus Error on first instruction fetch for an exception handler	MAV	EXCP_ERR	RA
	first instruction fetch for an exception handler	ITLB Error on first instruction fetch for an exception handler	MAV	EXCP_ERR	EA

[1] The MCSR update column indicates which bits in the MCSR will be updated when the exception is logged.

[2] The MCAR update column indicates whether or not the error will provide either a real address (RA), effective address (EA), or no address (none) that is associated with the error.

[3] The machine check input pin is used by the platform logic to indicate machine check type errors that are detected by the platform. Software must query error logging information within the platform logic to determine the specific error condition and source.

[4] The RA stored in the MCAR for this case will be Snoop Address value, with the index bits set to 0.

Table 7-11details the priority of asynchronous machine check updates to the MCAR when multiple simultaneous async machine check conditions occur. Note that since a lower priority condition may occur and then a higher priority condition may subsequently occur prior to the machine check interrupt handler reading the MCSR and MCAR, the interrupt handler may not necessarily see the higher priority MCAR value, even though multiple MCSR bits are set.

Table 7-11. Asynchronous machine check MCAR update priority

Priority (0 — highest)	Asynchronous machine check source	Transaction source	Error type	(MCSR update)
0	Exception Vectoring	first instruction fetch for an exception handler	ISI or Bus Error on first instruction fetch for an exception handler	EXCP_ERR
		first instruction fetch for an exception handler	ITLB Error on first instruction fetch for an exception handler	EXCP_ERR
1	Data Cache	replacement push dcbf push dcbst push L1FINV0 push reservation-type instruction forced push	Dirty push parity error	CP_PERR
2	BIU	store or push	Bus error on write or push	BUS_WRERR
3	Data Cache	load or store dcblc dcbtls dcbtstls dcbz	Uncorrectable tag array parity error & L1CSR0$_{DCEA}$=01 & locked line invalidated	DC_TPERR, DC_LKERR
4	Instruction Cache	icblc icbtls instruction fetch	Uncorrectable tag array parity error & L1CSR1$_{ICEA}$=01 & locked line invalidated	IC_TPERR, IC_LKERR
5	BIU	load	Bus error on locked line error recovery refill	BUS_DRERR, DC_LKERR
6	BIU	instruction fetch	Bus error on locked line error recovery refill	BUS_IRERR, IC_LKERR
7	Data Cache	load or store dcbf dcbtls dcbtstls dcbz L1FINV0 flush or flush w/inv & line dirty	Tag array parity error & L1CSR0$_{DCEA}$=00	DC_TPERR
			Uncorrectable tag array parity error & L1CSR0$_{DCEA}$=01 & line dirty or potentially dirty	
7	Data Cache	load or store dcbtls dcbtstls dcbz	Cacheable miss & L1CSR0$_{DCEA}$='00' & dirty parity error on line to be replaced	DC_TPERR

Table 7-11. Asynchronous machine check MCAR update priority (continued)

Priority (0 — highest)	Asynchronous machine check source	Transaction source	Error type	(MCSR update)
7	Data Cache	load or store dcbtls dcbtstls dcbz	Cacheable miss & L1CSR0$_{DCEA}$=00 & lock parity error	DC_TPERR, DC_LKERR
			Cacheable miss & L1CSR0$_{DCEA}$=01 & uncorrectable lock parity error	
8	Data Cache	dcbst	Tag array parity error & L1CSR0$_{DCEA}$=00 & line potentially dirty (dirty or dirty parity error)	DC_TPERR, [DC_LKERR (if lock parity error)]
			Uncorrectable tag array parity error & L1CSR0$_{DCEA}$=01 & line potentially dirty (dirty or dirty parity error)	DC_TPERR, [DC_LKERR (if uncorrectable lock parity error)]
9	Data Cache	dcblc	Tag array parity error & L1CSR0$_{DCEA}$=00 & line potentially locked (locked or lock parity error)	DC_TPERR, [DC_LKERR (if lock parity error)]
			Uncorrectable tag array parity error & L1CSR0$_{DCEA}$=01 & line potentially locked (locked or lock parity error)	DC_TPERR, [DC_LKERR (if uncorrectable lock parity error)]
10	Data Cache	load	Data array parity error & L1CSR0$_{DCEA}$=00	DC_DPERR
			Data array parity error & line dirty or potentially dirty & L1CSR0$_{DCEA}$=01	
11	Instruction Cache	icblc	Tag array parity error & L1CSR1$_{ICEA}$=00 & line locked or lock parity error	IC_TPERR, [IC_LKERR]
		icbtls	Tag array parity error & L1CSR1$_{ICEA}$=00	IC_TPERR
			Cacheable miss & L1CSR1$_{ICEA}$=00 & lock parity error	IC_TPERR, IC_LKERR
			Cacheable miss & L1CSR1$_{ICEA}$=01 & uncorrectable lock parity error	

Table 7-11. Asynchronous machine check MCAR update priority (continued)

Priority (0 — highest)	Asynchronous machine check source	Transaction source	Error type	(MCSR update)
12	BIU	load store/w allocate dcbtls dcbtstls	Bus error on load or linefill or data refill	BUS_DRERR
13	BIU	icbtls CI or cache disabled Ifetch	Bus error on linefill or data refill Bus error on CI Ifetch Bus error on cache disabled Ifetch	BUS_IRERR
14	Data Cache	snoop lookup	Tag parity error & (miss, or hit only to way with tag parity error)	DC_TPERR, SNPERR
15	Instruction Cache	Instruction Fetch	Tag array parity error & L1CSR1$_{ICEA}$=00	IC_TPERR
16	Instruction Cache		Data array parity error & L1CSR1$_{ICEA}$=00	IC_DPERR
17	Instruction Cache	Instruction Fetch	Cacheable miss & L1CSR1$_{ICEA}$=00 & lock parity error	IC_TPERR, IC_LKERR
			Cacheable miss & L1CSR1$_{ICEA}$=01 & uncorrectable lock parity error	

7.7.2.2 Machine check interrupt actions

Machine Check interrupts for "error report" conditions and NMI are enabled and taken regardless of the state of MSR$_{ME}$. Machine check interrupts due to an "async mchk" syndrome bit being set in MCSR are only taken when MSR$_{ME}$ =1. When a Machine Check interrupt is taken, registers are updated as shown in Table 7-12.

Table 7-12. Machine check interrupt — register settings

Register	Setting description		
MCSRR0	On a best-effort basis e200z759n3 sets this to the address of some instruction that was executing or about to be executing when the machine check condition occurred.		
MCSRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 SPE 0 WE 0 CE 0 EE 0 PR 0	FP 0 ME 0 FE0 0 DE 0/—[1]	FE1 0 IS 0 DS 0 PMM 0 RI 0

Table 7-12. Machine check interrupt — register settings (continued)

Register	Setting description
ESR	Unchanged
MCSR	Updated to reflect the source(s) of a machine check. Hardware only sets appropriate bits, no previously set bits are cleared by hardware.
MCAR	See Table 7-10
Vector	$IVPR_{0:15}$ || $IVOR1_{16:27}$ || 4b0000

[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.

The Machine Check Syndrome register is provided to identify the source(s) of a machine check, and in conjunction with $MCSRR1_{RI}$, may be used to identify recoverable events.

The MSR_{RI} status bit is provided for software use in determining if multiple nested machine check exceptions have occurred. Software may interrogate the $MCSRR1_{RI}$ bit to determine if a machine check occurred during the initial portion of a machine check handler prior to handler code that sets MSR_{RI} to ‘1’ to indicate that the handler can now tolerate another machine check condition without losing state necessary for recovery. The interrupt handler should set MSR_{RI} as soon as possible after saving off working registers and MCSRR0,1 to avoid loss of state if another machine check condition were to occur.

The Machine Check input pin **p_mcp_b** can be masked by $HID0_{EMCP}$.

The Non-Maskable Interrupt machine check input pin **p_nmi_b** is never masked.

Precise external termination errors occur when a load or cache-inhibited or guarded store is terminated by assertion of **p_tea_b** (external bus ERROR termination response); these result in both an “error report” and an “async mchk” machine check exception.

Some machine check exceptions are unrecoverable in the sense that execution cannot resume in the context that existed before the interrupt; however, system software can use the machine check interrupt handler to try to identify and recover from the machine check condition.

7.7.2.3 Checkstop state

Machine checks no longer result in a checkstop and there is no checkstop state implemented on Zen z7.

7.7.3 Data Storage interrupt (IVOR2)

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

- Read or Write Access Control exception condition
- Byte Ordering exception condition
- Cache Locking exception condition

Access control is defined as in *PowerISA 2.06*. A Byte Ordering exception condition occurs for any misaligned access across a page boundary to pages with mismatched E bits. Cache locking exception conditions occur for any attempt to execute a **dcbtls**, **dcbtstls**, **dcblc**, **icbtls**, or **icblc** in user mode with $MSR_{UCLE} = 0$.

Table 7-13 lists register settings when a DSI is taken.

Table 7-13. 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 PMM 0 RI —
ESR	Access: Byte ordering: Cache locking:	[ST], [SPE], [VLEMI]. All other bits cleared. [ST], [SPE], [VLEMI], BO. All other bits cleared. (DLK, ILK), [VLEMI], [ST]. All other bits cleared.	
MCSR	Unchanged		
DEAR	For Access and Byte ordering exceptions, set to the effective address of a byte within the page whose access caused the violation. Undefined on Cache locking exceptions (Zen does not update the DEAR on a cache locking exception)		
Vector	$IVPR_{0:15}$ || $IVOR2_{16:27}$ || 4b0000		

7.7.4 Instruction Storage interrupt (IVOR3)

An Instruction Storage interrupt (ISI) occurs when no higher priority exception exists and an Execute Access Control exception occurs. This interrupt is implemented as defined by *PowerISA 2.06.*,with the addition of Misaligned Instruction Fetch exceptions, and the extension of the Byte Ordering exception status to also cover Mismatched Instruction Storage exceptions.

Exception extensions implemented in e200z759n3 for PowerISA VLE involve extending the definition of the Instruction Storage Interrupt to include Byte Ordering exceptions for instruction accesses, and Misaligned Instruction Fetch exceptions, and corresponding updates to the ESR as shown in Table 7-14 and Table 7-15.

Table 7-14. ISI exceptions and conditions

Interrupt type	Interrupt vector offset register	Causing conditions
Instruction Storage	IVOR 3	• Access control. • Byte ordering due to misaligned instruction across page boundary to pages with mismatched VLE bits, or access to page with VLE set, and E indicating little-endian. • Misaligned Instruction fetch due to a change of flow to an odd halfword instruction boundary on a BookE (non-VLE) instruction page

Table 7-15 lists register settings when an ISI is taken.

Table 7-15. 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 PMM 0 RI —
ESR	[BO, MIF, VLEMI]. All other bits cleared.		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	$IVPR_{0:15}$ || $IVOR3_{16:27}$ || 4b0000		

7.7.5 External Input interrupt (IVOR4)

An External Input exception is signaled to the processor by the assertion of the external interrupt pin (**p_extint_b**). The **p_extint_b** input is a level-sensitive signal expected to remain asserted until e200z759n3 acknowledges the external interrupt. If **p_extint_b** is negated early, recognition of the interrupt request is not guaranteed. When e200z759n3 detects the exception, if the exception is enabled by MSR_{EE}, e200z759n3 takes the External Input interrupt.

An External Input interrupt may be delayed by other higher priority exceptions or if MSR_{EE} is cleared when the exception occurs.

Table 7-16 lists register settings when an External Input interrupt is taken.

Table 7-16. 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 PMM 0 RI —
ESR	Unchanged		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	$IVPR_{0:15}$ || $IVOR4_{16:27}$ || 4b0000 $IVPR_{0:15}$ || **p_voffset[0:11]** || 4b0000 (non-autovectored)		

IVOR4 is the vector offset register used by autovectored External Input interrupts to determine the interrupt handler location. e200z759n3 also provides the capability to directly vector External Input interrupts to multiple handlers by allowing a External Input interrupt request to be accompanied by a vector offset. The **p_voffset[0:11]** input signals are used in place of the value in IVOR4 when a External Input interrupt request is not autovectored (**p_avec_b** negated when **p_extint_b** asserted).

7.7.6 Alignment interrupt (IVOR5)

e200z759n3 implements the Alignment interrupt as defined by *PowerISA 2.06*. An Alignment exception is generated when any of the following occurs:

- The operand of **lmw** or **stmw** not word aligned.
- The operand of **lwarx** or **stwcx.** not word aligned.
- The operand of **lharx** or **sthcx.** not halfword aligned.
- Execution of a **dcbz** instruction is attempted with a disabled cache.
- Execution of a **dcbz** instruction with an enabled cache and W or I =1.
- Execution of a SPE APU load or store instruction that is not properly aligned.

Table 7-17 lists register settings when an alignment interrupt is taken.

Table 7-17. Alignment interrupt—register settings

Register	Setting description		
SRR0	Set to the effective address of the excepting load/store instruction.		
SRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 SPE 0 WE 0 CE — EE 0 PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0 PMM 0 RI —
ESR	[ST], [SPE], [VLEMI]. All other bits cleared.		
MCSR	Unchanged		
DEAR	Set to the effective address of a byte of the load or store whose access caused the violation.		
Vector	$IVPR_{0:15}$ ‖ $IVOR5_{16:27}$ ‖ 4b0000		

7.7.7 Program interrupt (IVOR6)

e200z759n3 implements the Program interrupt as defined by *PowerISA 2.06*. A program interrupt occurs when no higher priority exception exists and one or more of the following exception conditions defined in *PowerISA 2.06* occur:

- Illegal Instruction exception
- Privileged Instruction exception
- Trap exception

- Unimplemented Operation exception

e200z759n3 will invoke an Illegal Instruction program exception on attempted execution of the following instructions:

- Unimplemented instructions
- Instruction from the illegal instruction class
- **mtspr** and **mfspr** instructions with an undefined SPR specified
- **mtdcr** and **mfdcr** instructions with an undefined DCR specified

e200z759n3 will invoke 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).

e200z759n3 will invoke an Trap exception on execution of the **tw** and **twi** instructions if the trap conditions are met and the exception is not also enabled as a Debug interrupt.

e200z759n3 will invoke an Illegal instruction program exception on attempted execution of the instructions **lswi**, **lswx**, **stswi**, **stswx**, **mfapidi**, **mfdcrx**, **mtdcrx**, or on any *PowerISA 2.06* floating point instruction when MSR_{FP}=1. All other defined or allocated instructions that are not implemented by e200z759n3 will cause a illegal instruction program exception.

Table 7-18 lists register settings when a Program interrupt is taken.

Table 7-18. Program interrupt—register settings

Register	Setting description		
SRR0	Set to the effective address of the excepting instruction.		
SRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 SPE 0 WE 0 CE — EE 0 PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0 PMM 0 RI —
ESR	Illegal: Privileged: Trap:	PIL, [VLEMI]. All other bits cleared. PPR, [VLEMI]. All other bits cleared. PTR, [VLEMI]. All other bits cleared.	
MCSR	Unchanged		
DEAR	Unchanged		
Vector	$IVPR_{0:15}$ || $IVOR6_{16:27}$ || 4b0000		

7.7.8 Floating-Point Unavailable interrupt (IVOR7)

The Floating-point Unavailable exception is not used by e200z759n3.

7.7.9 System Call interrupt (IVOR8)

A System Call interrupt occurs when a System Call (**sc, se_sc)** instruction is executed and no higher priority exception exists.

Exception extensions implemented in e200z759n3 for PowerISA VLE include modification of the System Call Interrupt definition to include updating the ESR.

Table 7-19 lists register settings when a System Call interrupt is taken.

Table 7-19. System Call interrupt—register settings

Register	Setting description		
SRR0	Set to the effective address of the instruction *following* the **sc** instruction.		
SRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 SPE 0 WE 0 CE — EE 0 PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0 PMM 0 RI —
ESR	[VLEMI] All other bits cleared.		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	IVPR$_{0:15}$ || IVOR8$_{16:27}$ || 4b0000		

7.7.10 Auxiliary Processor Unavailable interrupt (IVOR9)

An Auxiliary Processor Unavailable exception is defined by *PowerISA 2.06* to occur when an attempt is made to execute an APU instruction that is implemented but configured as unavailable, and no higher priority exception condition exists.

e200z759n3 does not utilize this interrupt.

7.7.11 Decrementer interrupt (IVOR10)

e200z759n3 implements the Decrementer exception as described in Chapter 8, “Timer Facilities” beginning on page 181 in *Book E: Enhanced PowerPC^{tm} Architecture v0.99*. A Decrementer interrupt occurs when no higher priority exception exists, a Decrementer exception condition exists (TSR$_{DIS}$=1), and the interrupt is enabled (both TCR$_{DIE}$ and MSR$_{EE}$=1).

The Timer Status Register (TSR) holds the Decrementer interrupt bit set by the Timer facility when an exception is detected. Software must clear this bit in the interrupt handler to avoid repeated Decrementer interrupts.

Table 7-20 lists register settings when a Decrementer interrupt is taken.

Table 7-20. Decrementer interrupt—register settings

Register	Setting description		
SRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present.		
SRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 SPE 0 WE 0 CE — EE 0 PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0 PMM 0 RI —
ESR	Unchanged		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	$IVPR_{0:15}$ || $IVOR10_{16:27}$ || 4b0000		

7.7.12 Fixed-Interval Timer interrupt (IVOR11)

e200z759n3 implements the Fixed-Interval Timer (FIT) exception as described in Chapter 8, “Timer Facilities” beginning on page 181 in *Book E: Enhanced PowerPC^{tm} Architecture v0.99*. The triggering of the exception is caused by selected bits in the Time Base register changing from 0 to 1.

A Fixed-Interval Timer interrupt occurs when no higher priority exception exists, a FIT exception exists (TSR_{FIS}=1), and the interrupt is enabled (both TCR_{FIE} and MSR_{EE}=1).

The Timer Status Register (TSR) holds the FIT interrupt bit set by the Timer facility when an exception is detected. Software must clear this bit in the interrupt handler to avoid repeated FIT interrupts.

Table 7-21 lists register settings when a FIT interrupt is taken.

Table 7-21. Fixed-Interval Timer interrupt—register settings

Register	Setting description		
SRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present.		
SRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 SPE 0 WE 0 CE — EE 0 PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0 PMM 0 RI —
ESR	Unchanged		
MCSR	Unchanged		

Table 7-21. Fixed-Interval Timer interrupt—register settings (continued)

DEAR	Unchanged
Vector	$IVPR_{0:15}$ || $IVOR11_{16:27}$ || 4b0000

7.7.13 Watchdog Timer interrupt (IVOR12)

e200z759n3 implements the Watchdog Timer (WDT) exception as described in Chapter 8, “Timer Facilities” beginning on page 181 in *Book E: Enhanced PowerPC^{tm} Architecture v0.99*. The triggering of the exception is caused by the first enabled watchdog time-out.

A Watchdog Timer interrupt occurs when no higher priority exception exists, a Watchdog Timer exception exists (TSR_{WIS}=1), and the interrupt is enabled (both TCR_{WIE} and MSR_{CE}=1).

The Timer Status Register (TSR) holds the Watchdog interrupt bit set by the Timer facility when an exception is detected. Software must clear this bit in the interrupt handler to avoid repeated Watchdog interrupts.

Table 7-22 lists register settings when a Watchdog Timer interrupt is taken.

Table 7-22. Watchdog Timer interrupt—register settings

Register	Setting description		
CSRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present.		
CSRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 SPE 0 WE 0 CE 0 EE 0 PR 0	FP 0 ME — FE0 0 DE 0/—[1]	FE1 0 IS 0 DS 0 PMM 0 RI —
ESR	Unchanged		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	$IVPR_{0:15}$ || $IVOR12_{16:27}$ || 4b0000		

[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.

The MSR_{DE} bit is not automatically cleared by a Watchdog Timer interrupt, but can be configured to be cleared via the HID0 register ($HID0_{CICLRDE}$). Refer to Section 2.4.11, Hardware Implementation Dependent Register 0 (HID0).

7.7.14 Data TLB Error interrupt (IVOR13)

A Data TLB Error interrupt occurs when no higher priority exception exists and a Data TLB Error exception exists due to a data translation lookup miss in the TLB.

Table 7-23 lists register settings when a DTLB interrupt is taken.

Table 7-23. Data TLB Error interrupt—register settings

Register	Setting description		
SRR0	Set to the effective address of the excepting load/store instruction.		
SRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 SPE 0 WE 0 CE — EE 0 PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0 PMM 0 RI —
ESR	[ST], [SPE], [VLEMI]. All other bits cleared.		
MCSR	Unchanged		
DEAR	Set to the effective address of a byte of the load or store whose access caused the violation.		
Vector	$\text{IVPR}_{0:15} \Vert \text{IVOR13}_{16:27} \Vert$ 4b0000		

7.7.15 Instruction TLB Error interrupt (IVOR14)

A Instruction TLB Error interrupt occurs when no higher priority exception exists and an Instruction TLB Error exception exists due to an instruction translation lookup miss in the TLB.

Exception extensions implemented in e200z759n3 for PowerISA VLE involve extending the definition of the Instruction TLB Error Interrupt to include updating the ESR.

Table 7-24 lists register settings when an ITLB interrupt is taken.

Table 7-24. 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 PMM 0 RI —
ESR	[MIF] All other bits cleared.		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	$\text{IVPR}_{0:15} \Vert \text{IVOR14}_{16:27} \Vert$ 4b0000		

7.7.16 Debug interrupt (IVOR15)

e200z759n3 implements the Debug Interrupt as defined in *PowerISA 2.06* with the following changes:

- When the Debug APU is enabled, Debug is no longer a critical interrupt, but uses DSRR0 and DSRR1 for saving machine state on context switch
- A Return from debug interrupt instruction (**rfdi** or **se_rfdi**) is implemented to support the new machine state registers
- A Critical Interrupt Taken debug event is defined to allow critical interrupts to generate a debug event
- A Critical Return debug event is defined to allow debug events to be generated for **rfci** and **se_rfci** instructions

There are multiple sources that can signal a Debug exception. A Debug interrupt occurs when no higher priority exception exists, a Debug exception exists in the Debug Status Register, and Debug interrupts are enabled (both $DBCR0_{IDM}$=1 (internal debug mode) and MSR_{DE}=1). Enabling debug events and other debug modes are discussed further in Chapter 12, Debug Support. With the Debug APU enabled, (See Section 2.4.11, Hardware Implementation Dependent Register 0 (HID0)) the Debug interrupt has its own set of machine state save/restore registers (DSRR0, DSRR1) to allow debugging of both critical and non-critical interrupt handlers. In addition, the capability is provided to allow interrupts to be handled while in a debug software handler. External and Critical interrupts are not automatically disabled when a Debug interrupt occurs but can be configured to be cleared via the HID0 register ($HID0_{DCLREE, DCLRCE}$). Refer to Section 2.4.11, Hardware Implementation Dependent Register 0 (HID0). When the Debug APU is disabled, Debug interrupts use the CSRR0 and CSRR1 registers to save machine state.

NOTE

For additional details regarding the following descriptions of debug exception types, refer to Section 12.2, Software debug events and exceptions.

An Instruction Address Compare (IAC) debug exception occurs when there is an instruction address match as defined by the debug control registers and Instruction Address Compare events are enabled. This could either be a direct instruction address match or a selected set of instruction addresses. IAC has the highest interrupt priority of all instruction-based interrupts, even if the instruction itself may have encountered an Instruction TLB error or Instruction Storage exception.

A Branch Taken (BRT) debug exception is signaled when a branch instruction is considered taken by the branch unit and branch taken events are enabled. The Debug interrupt is taken when no higher priority exception is pending.

A Data Address Compare (DAC) exception is signaled when there is a data access address match as defined by the debug control registers and Data Address Compare events are enabled. This could either be a direct data address match or a selected set of data addresses, or a combination of data address and data value matching. The Debug interrupt is taken when no higher priority exception is pending.

The e200z759n3 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 12, Debug Support, for more details.

A Trap (TRAP) debug exception occurs when a program trap exception is generated while trap events are enabled. If MSR_{DE} is set, the Debug exception has higher priority than the Program exception in this case, and will be taken instead of a Trap type Program Interrupt. The Debug interrupt is taken when no higher priority exception is pending. If MSR_{DE} is cleared when a trap debug exception occurs, a Trap exception type Program interrupt will occur instead.

A Return (RET) debug exception occurs when executing an **rfi** or **se_rfi** instruction and return debug events are enabled. Return debug exceptions are not generated for **rfci** or **se_rfci** instructions. If MSR_{DE}=1 at the time of the execution of the **rfi** or **se_rfi**, a Debug interrupt will occur provided there exists no higher priority exception that is enabled to cause an interrupt. CSRR0 (Debug APU disabled) or DSRR0 (Debug APU enabled) will be set to the address of the **rfi** or **se_rfi** instruction. If MSR_{DE}=0 at the time of the execution of the **rfi** or **se_rfi**, a Debug interrupt will not occur immediately, but the event will be recorded by setting the $DBSR_{RET}$ and $DBSR_{IDE}$ status bits.

A Critical Return (CRET) debug exception occurs when executing an **rfci** or **se_rfci** instruction and critical return debug events are enabled. Critical return debug exceptions are only generated for **rfci** or **se_rfci** instructions. If MSR_{DE}=1 at the time of the execution of the **rfci** or **se_rfci**, a Debug interrupt will occur provided there exists no higher priority exception that is enabled to cause an interrupt. CSRR0 (Debug APU disabled) or DSRR0 (Debug APU enabled) will be set to the address of the **rfci** or **se_rfci** instruction. If MSR_{DE}=0 at the time of the execution of the **rfci** or **se_rfci**, a Debug interrupt will not occur immediately, but the event will be recorded by setting the $DBSR_{CRET}$ and $DBSR_{IDE}$ status bits. Note that critical return debug events should not normally be enabled unless the Debug APU is enabled to avoid corruption of CSRR0/1.

An Instruction Complete (ICMP) debug exception is signaled following execution and completion of an instruction while this event is enabled.

A **mtmsr** or **mtdbcr0** that causes both MSR_{DE} and $DBCR0_{IDM}$ to end up set, enabling precise debug mode, may cause an Imprecise (Delayed) Debug exception to be generated due to an earlier recorded event in the Debug Status register.

An Interrupt Taken (IRPT) debug exception occurs when a 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 will only occur when detecting a non-critical interrupt on e200z759n3. The value saved in CSRR0/DSRR0 will be the address of the non-critical interrupt handler.

A Critical Interrupt Taken (CIRPT) debug exception occurs when a critical interrupt context switch is detected. This exception is imprecise and unordered with respect to the program flow. Note that a CIRPT Debug interrupt will only occur when detecting a critical interrupt on e200z759n3. The value saved in CSRR0/DSRR0 will be the address of the critical interrupt handler. Note that Critical Interrupt Taken debug events should not normally be enabled unless the Debug APU is enabled to avoid corruption of CSRR0/1.

An Unconditional Debug Event (UDE) exception occurs when the Unconditional Debug Event pin (**p_ude**) transitions to the asserted state.

Debug Counter Debug exceptions occur when enabled and one of the Debug counters decrements to zero.

External Debug exceptions occur when enabled and one of the External Debug Event pins (**p_devt1**, **p_devt2**) transitions to the asserted state.

The Debug Status Register (DBSR) provides a *syndrome* to differentiate between debug exceptions that can generate the same interrupt. For more details see Chapter 12, Debug Support”.

Table 7-25 lists register settings when a Debug interrupt is taken.

Table 7-25. Debug interrupt—register settings

Register	Setting description		
CSRR0/ DSRR0[1]	Set to the effective address of the excepting instruction for IAC, BRT, RET, CRET, and TRAP. Set to the effective address of the next instruction to be executed *following* the excepting instruction for DAC and ICMP. For a UDE, IRPT, CIRPT, DCNT, or DEVT type exception, set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present.		
CSRR1/ DSRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 SPE 0 WE 0 CE —/0[2] EE —/0[2] PR 0	FP 0 ME — FE0 0 DE 0	FE1 0 IS 0 DS 0 PMM 0 RI —
DBSR[3]	Unconditional Debug Event: Instr. Complete Debug Event: Branch Taken Debug Event: Interrupt Taken Debug Event: Critical Interrupt Taken Debug Event: Trap Instruction Debug Event: Instruction Address Compare: Data Address Compare: Return Debug Event: Critical Return Debug Event: Debug Counter Event: External Debug Event: and optionally, an Imprecise Debug Event flag	UDE ICMP BRT IRPT CIRPT TRAP {IAC1, IAC2, IAC3, IAC4} {DAC1R, DAC1W, DAC2R, DAC2W} RET CRET {DCNT1, DCNT2} {DEVT1, DEVT2} {IDE}	
ESR	Unchanged		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	$IVPR_{0:15}$ || $IVOR15_{16:27}$ || 4b0000		

[1] assumes that the Debug interrupt is precise

[2] conditional based on control bits in HID0

[3] Note that multiple DBSR bits may be set

7.7.17 System Reset interrupt

e200z759n3 implements the System Reset interrupt as defined in *PowerISA 2.06*. 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 by either asserting the **p_reset_b** input signal or during power-on reset by asserting **m_por**. The **m_por** signal must be asserted during power up and must remain asserted for a period that allows internal logic to be reset. The **p_reset_b** signal must also remain asserted for a period that allows internal logic to be reset. This period is specified in the hardware specifications. If **m_por** or **p_reset_b** are asserted for less than the required interval, the results are not predictable.

When a reset request occurs, the processor branches to the system reset exception vector (value on **p_rstbase[0:29]** concatenated with 2'b00) without attempting to reach a recoverable state. If reset occurs during normal operation, all operations cease and the machine state is lost. CPU internal state after a reset is defined in Section 2.6, Reset settings.

Reset may also be initiated by Watchdog Timer or Debug Reset Control. Watchdog Timer and Debug Reset Control provide the capability to assert the **p_wrs[0:1]** and **p_dbrstc[0:1]** signals. External logic may factor this into the **p_reset_b** input signal to cause a e200z759n3 reset to occur.

Table 7-26 shows the TSR register bits associated with Watchdog Timer reset status. Note that these bits will be cleared when a processor reset occurs, thus if the **p_wrs[0:1]** outputs are factored into **p_reset_b**, they will only be seen in the "00" state by software.

Table 7-26. TSR Watchdog Timer reset status

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

Table 7-27 shows the DBSR register bits associated with reset status.

Table 7-27. DBSR most recent reset

Bits	Name	Function
2:3 (34:35)	MRR	00 No reset occurred since these bits were last cleared by software 01 A reset occurred since these bits were last cleared by software 10 Reserved 11 Reserved

Table 7-28 lists register settings when a System Reset interrupt is taken.

Table 7-28. System Reset Interrupt—register settings

Register	Setting description		
CSRR0	Undefined.		
CSRR1	Undefined.		
MSR	UCLE 0 SPE 0 WE 0 CE 0 EE 0 PR 0	FP 0 ME 0 FE0 0 DE 0	FE1 0 IS 0 DS 0 PMM 0 RI 0

Table 7-28. System Reset Interrupt—register settings (continued)

Register	Setting description
ESR	Cleared
DEAR	Undefined
Vector	[**p_rstbase[0:29]**] ǁ 2'b00

7.7.18 SPE/EFPU APU Unavailable interrupt (IVOR32)

The SPE APU Unavailable exception is taken if MSR_{SPE} is cleared and execution of a SPE or EFPU APU instruction other than the scalar floating-point instructions (**efs**$_{\mathbf{xxx}}$) or **brinc** is attempted. When the SPE/EFPU APU Unavailable exception occurs, the processor suppresses execution of the instruction causing the exception. Table 7-29 lists register settings when a SPE/EFPU Unavailable interrupt is taken.

Table 7-29. SPE/EFPU Unavailable interrupt—register settings

Register	Setting description		
SRR0	Set to the effective address of the excepting SPE/EFPU instruction.		
SRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 SPE 0 WE 0 CE — EE 0 PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0 PMM 0 RI —
ESR	SPE, [VLEMI]. All other bits cleared.		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	$IVPR_{0:15}$ ǁ $IVOR32_{16:27}$ ǁ 4b0000		

7.7.19 Embedded Floating-point Data interrupt (IVOR33)

The Embedded Floating-point Data interrupt is taken if no higher priority exception exists and a EFPU Floating-point Data exception is generated. When a Floating-point Data exception occurs, the processor suppresses execution of the instruction causing the exception.

Table 7-30 lists register settings when a EFPU Floating-point Data interrupt is taken.

Table 7-30. Embedded Floating-point Data interrupt—register settings

Register	Setting description
SRR0	Set to the effective address of the excepting EFPU instruction.
SRR1	Set to the contents of the MSR at the time of the interrupt

Table 7-30. Embedded Floating-point Data interrupt—register settings (continued)

Register	Setting description		
MSR	UCLE 0 SPE 0 WE 0 CE — EE 0 PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0 PMM 0 RI —
ESR	SPE, [VLEMI]. All other bits cleared.		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	$IVPR_{0:15}$ || $IVOR33_{16:27}$ || 4b0000		

7.7.20 Embedded Floating-point Round interrupt (IVOR34)

The Embedded Floating-point Round interrupt is taken when a EFPU floating-point instruction generates an inexact result and inexact exceptions are enabled.

Table 7-31 lists register settings when a EFPU Floating-point Round interrupt is taken.

Table 7-31. Embedded Floating-point Round interrupt—register settings

Register	Setting description		
SRR0	Set to the effective address of the instruction following the excepting EFPU instruction.		
SRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 SPE 0 WE 0 CE — EE 0 PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0 PMM 0 RI —
ESR	SPE, [VLEMI]. All other bits cleared.		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	$IVPR_{0:15}$ || $IVOR34_{16:27}$ || 4b0000		

7.7.21 Performance monitor interrupt (IVOR35)

Zen Z7 provides a performance monitor interrupt that may be generated by an enabled condition or event. An enabled condition or event is as follows:

A PMC*x* register overflow condition occurs with the following settings:

- $PMLCax_{CE}$ = 1; that is, for the given counter the overflow condition is enabled.
- $PMCx_{OV}$ = 1; that is, the given counter indicates an overflow.

For a performance monitor interrupt to be signaled on an enabled condition or event, $PMGC0_{PMIE}$ must be set.

Although an exception condition may occur with MSR_{EE} = 0, the interrupt cannot be taken until MSR_{EE} = 1.

The priority of the performance monitor interrupt is below all other asynchronous interrupts. For details, see Section 8.4, Performance monitor interrupt.

Table 7-32 lists register settings when an performance monitor interrupt is taken.

Table 7-32. Performance monitor interrupt—register settings

Register	Setting description		
SRR0	Set to the effective address of the next instruction to be executed.		
SRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 SPE 0 WE 0 CE — EE 0 PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0 PMM 0 RI —
ESR	Unchanged		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	$IVPR_{0:15}$ || $IVOR35_{16:27}$ || 4b0000		

7.8 Exception recognition and priorities

The following list of exception categories describes how e200z759n3 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, non-maskable, possibly non-recoverable:

    Non-maskable interrupt by assertion of **p_nmi_b**

    Has priority over any other pending exception except system reset conditions. Recoverability is dependent on whether MCSRR0/1 are holding essential state info and are overwritten when the NMI occurs.

— Asynchronous, maskable/non-maskable, recoverable/non-recoverable:

  Machine check interrupt

  Has priority over any other pending exception except system reset conditions. Recoverability is dependent on the source of the exception.

— Asynchronous, maskable, recoverable:

  External Input, Fixed-Interval Timer, Decrementer, Critical Input, Performance Monitor, Unconditional Debug, External Debug Event, Debug Counter Event, and Watchdog Timer interrupts

  Before handling this type of exception, the processor needs to reach a recoverable state. A maskable recoverable exception will remain pending until taken or cancelled by software.

- Synchronous, non instruction-based interrupts. The only exception is this category is the Interrupt Taken debug exception, recognized by an interrupt taken event. It is not considered instruction-based but is synchronous with respect to the program flow.

  — Synchronous, maskable, recoverable:

  Interrupt Taken debug event

  The machine will be in a recoverable state due to the state of the machine at the context switch triggering this event.

- Instruction-based interrupts. These interrupts are further organized by the point in instruction processing in which they generate an exception.

  — Instruction Fetch:

  Instruction Storage, Instruction TLB, and Instruction Address Compare debug exceptions

  Once these types of exceptions are detected, the excepting instruction is tagged. When the excepting instruction is next to begin execution and a recoverable state has been reached, the interrupt is taken. If an event prior to the excepting instruction causes a redirection of execution, the instruction fetch exception is discarded (but may be encountered again).

  — Instruction Dispatch/Execution:

  Program, System Call, Data Storage, Alignment, SPE/EFPU Unavailable, Data TLB, Embedded Floating-point Data, Embedded Floating-point Round, Debug (Trap, Branch Taken, Ret) interrupts

  These types of exceptions are determined during decode or execution of an instruction. The exception remains pending until all instructions before the exception causing instruction in program order complete. The interrupt is then taken without completing the exception-causing instruction. If completing previous instructions causes an exception, that exception takes priority over the pending instruction dispatch/execution exception, which is discarded (but may be encountered again when instruction processing resumes).

  — Post-Instruction Execution:

  Debug (Data Address Compare, Instruction Complete) interrupt

These Debug exceptions are generated following execution and completion of an instruction while the event is enabled. If executing the instruction produces conditions for another type of exception with higher priority, that exception is taken and the post-instruction exception is discarded for the instruction (but may be encountered again when instruction processing resumes).

7.8.1 Exception priorities

Exceptions are prioritized as described in Table 7-33. Some exceptions may be masked or imprecise, which will affect their priority. Non-maskable exceptions such as reset and machine check may occur at any time and are not delayed even if an interrupt is being serviced, thus state information for any interrupt may be lost. Reset and certain machine checks are non-recoverable.

Table 7-33. e200z759n3 exception priorities

Priority	Exception	Cause	IVOR
Asynchronous exceptions			
0	System reset	Assertion of **p_reset_b**, Watchdog Timer Reset Control, or Debug Reset Control	none
1	Machine check	Assertion of **p_mcp_b**, assertion of **p_nmi_b**, Cache Parity errors, exception on fetch of first instruction of an interrupt handler, external bus errors	1
2	—	—	
3[1]	Debug: • UDE • DEVT1 • DEVT2 • DCNT1 • DCNT2 • IDE	• Assertion of **p_ude** (Unconditional Debug Event) • Assertion of **p_devt1** and event enabled (External Debug Event 1) • Assertion of **p_devt2** and event enabled (External Debug Event 2) • Debug Counter 1 exception • Debug Counter 2 exception • Imprecise Debug Event (event imprecise due to previous higher priority interrupt	15
4[1]	Critical Input	Assertion of **p_critint_b**	0
5[1]	Watchdog Timer	Watchdog Timer first enabled time-out	12
6[1]	External Input	Assertion of **p_extint_b**	4
7[1]	Fixed-Interval Timer	Posting of a FIT exception in TSR due to programmer-specified bit transition in the Time Base register	11
8[1]	Decrementer	Posting of a Decrementer exception in TSR due to programmer-specified Decrementer condition	10
9[1]	Performance Monitor	Performance Monitor Enabled Condition or Event	35
Instruction Fetch exceptions			
10	Debug: • IAC (unlinked)	• Instruction address compare match for enabled IAC debug event and DBCR0$_{IDM}$ asserted	15
11	ITLB Error	Instruction translation lookup miss in the TLB	14

Table 7-33. e200z759n3 exception priorities (continued)

Priority	Exception	Cause	IVOR
12	Instruction Storage	• Access control. • Byte ordering due to misaligned instruction across page boundary to pages with mismatched VLE bits, or access to page with VLE set, and E indicating little-endian. • Misaligned Instruction fetch due to a change of flow to an odd halfword instruction boundary on a BookE (non-VLE) instruction page, due to value in LR, CTR, or xSRR0	3
Instruction Dispatch/Execution interrupts			
13	Program: • Illegal	• Attempted execution of an illegal instruction.	6
14	Program: • Privileged	• Attempted execution of a privileged instruction in user-mode	6
15	SPE/EFPU Unavailable	Any SPE or EFPU unavailable exception condition.	32
16	Program: • Unimplemented	• Attempted execution of an unimplemented instruction. (unused by e200z759n3)	6
17	Debug: • BRT • Trap • RET • CRET	• Attempted execution of a taken branch instruction • Condition specified in **tw** or **twi** instruction met. • Attempted execution of a **rfi** instruction. • Attempted execution of an **rfci** instruction. **Note:** Exceptions requires corresponding debug event enabled, MSR_{DE}=1, and $DBCR0_{IDM}$=1.	15
18	Program: • Trap	• Condition specified in **tw** or **twi** instruction met and not trap debug.	6
	System Call	Execution of the System Call (**sc, se_sc**) instruction.	8
	EFPU Floating-point Data	Denormalized, NaN, or Infinity data detected as input or output, or underflow, overflow, divide by zero, or invalid operation in the EFPU APU.	33
	EFPU Round	Inexact Result	34
19	Alignment	**lmw**, **stmw, lwarx,** or **stwcx.** not word aligned. **lharx,** or **sthcx.** not halfword aligned. **dcbz** with cache disabled.	5

Table 7-33. e200z759n3 exception priorities (continued)

Priority	Exception	Cause	IVOR
20	Debug: Debug with concurrent DTLB or DSI exception, or concurrent async machine check: • DAC/IAC linked[2] • DAC unlinked[2]	Debug with concurrent DTLB or DSI exception, or async machine check condition on the DAC. $DBSR_{IDE}$ also set. • Data Address Compare linked with Instruction Address Compare • Data Address Compare unlinked **Note:** Exceptions requires corresponding debug event enabled, MSR_{DE}=1, and $DBCR0_{IDM}$=1. In this case, the Debug exception is considered imprecise, and $DBSR_{IDE}$ will be set. Saved PC will point to the load or store instruction causing the DAC event.	15
21	Data TLB Error	Data translation lookup miss in the TLB.	13
22	Data Storage	• Access control. • Byte ordering due to misaligned access across page boundary to pages with mismatched E bits. • Cache locking due to attempt to execute a **dcbtls**, **dcbtstls**, **dcblc**, **icbtls**, or **icblc** in user mode with MSR_{UCLE} = 0.	2
23	Alignment	**dcbz** to W=1 or I=1 storage with cache enabled	5
24	Debug: • IRPT • CIRPT	• Interrupt taken (non-critical) • Critical Interrupt taken (critical only) **Note:** Exceptions requires corresponding debug event enabled, MSR_{DE}=1, and $DBCR0_{IDM}$=1.	15
Post-instruction execution exceptions			
25	Debug: • DAC/IAC linked[2] • DAC unlinked[2]	• Data Address Compare linked with Instruction Address Compare • Data Address Compare unlinked **Note:** Exceptions requires corresponding debug event enabled, MSR_{DE}=1, and $DBCR0_{IDM}$=1. Saved PC will point to the instruction following the load or store instruction causing the DAC event.	15
26	Debug: • ICMP	• Completion of an instruction. **Note:** Exceptions requires corresponding debug event enabled, MSR_{DE}=1, and $DBCR0_{IDM}$=1.	15

[1] These asynchronous exceptions are sampled at instruction boundaries, thus may actually occur after exceptions 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, e200z759n3 implements the data address compare debug exceptions as post-instruction exceptions, which differ from the *PowerISA 2.06* definition. When a TEA (either a DTLB error or DSI or Machine Check (external TEA)) occurs in conjunction with an enabled DAC or linked DAC/IAC on a load or store class instruction, or a Debug Counter event based on a counted DAC, the Debug Interrupt takes priority, and the saved PC value will point to the load or store class instruction, rather than to the next instruction.

7.9 Interrupt processing

When an interrupt is taken, the processor uses SRR0/SRR1 for non-critical interrupts, CSRR0/CSRR1 for critical interrupts, MCSRR0/MCSRR1 for machine check interrupts, and either CSRR0/CSRR1 or DSRR0/DSRR1 for debug interrupts to save the contents of the MSR and to assist in identifying where instruction execution should resume after the interrupt is handled.

When an interrupt occurs, one of SRR0/CSRR0/DSRR0/MCSRR0 is set to the address of the instruction that caused the exception, or to the following instruction if appropriate.

SRR1 is used to save machine state (selected MSR bits) on non-critical interrupts and to restore those values when an **rfi** instruction is executed.

CSRR1 is used to save machine status (selected MSR bits) on critical interrupts and to restore those values when an **rfci** instruction is executed.

DSRR1 is used to save machine status (selected MSR bits) on debug interrupts when the Debug APU is enabled and to restore those values when an **rfdi** instruction is executed.

MCSRR1 is used to save machine status (selected MSR bits) on machine check interrupts and to restore those values when an **rfmci** instruction is executed.

The Exception Syndrome register is loaded with information specific to the exception type. Some interrupt types can only be caused by a single exception type, and thus do not use an ESR setting to indicate the interrupt cause.

The Machine State register is updated to preclude unrecoverable interrupts from occurring during the initial portion of the interrupt handler. Specific settings are described in Table 7-34.

For Alignment, Data Storage, or Data TLB Miss interrupts, the Data Exception Address Register (DEAR) is loaded with the address that caused the interrupt to occur.

For Machine Check interrupts, the Machine Check Syndrome register is loaded with information specific to the exception type. For certain machine checks, the MCAR is loaded with an address corresponding to the machine check.

Instruction fetch and execution resumes, using the new MSR value, at a location specific to the exception type. The location is determined by the Interrupt Vector Prefix Register (IVPR), and an Interrupt Vector Offset Register (IVOR) specific for each type of interrupt (see Table 7-2).

Table 7-34 shows the MSR settings for different interrupt categories.

Table 7-34. MSR setting due to interrupt

Bits	MSR definition	Reset setting	Non-critical interrupt	Critical interrupt	Debug Interrupt	Machine Check interrupt
5 (37)	UCLE	0	0	0	0	0
6 (38)	SPE	0	0	0	0	0
13 (45)	WE	0	0	0	0	0
14 (46)	CE	0	—	0	—/0[1]	0

Table 7-34. MSR setting due to interrupt (continued)

Bits	MSR definition	Reset setting	Non-critical interrupt	Critical interrupt	Debug Interrupt	Machine Check interrupt
16 (48)	EE	0	0	0	—/0[1]	0
17 (49)	PR	0	0	0	0	0
18 (50)	FP	0	0	0	0	0
19 (51)	ME	0	—	—	—	0
20 (52)	FE0	0	0	0	0	0
22 (54)	DE	0	—	—/0[1]	0	—/0[1]
23 (55)	FE1	0	0	0	0	0
26 (58)	IS	0	0	0	0	0
27 (59)	DS	0	0	0	0	0
29 (61)	PMM	0	0	0	0	0
30 (62)	RI	0	—	—	—	0

Reserved and preserved bits are unimplemented and read as 0.

[1] Conditionally cleared based on control bits in HID0

7.9.1 Enabling and disabling exceptions

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

- System reset exceptions cannot be masked.
- Machine check exceptions cannot be masked from sources other than the machine check pin, and certain other async machine check status settings. Assertion of **p_mcp_b** is only recognized if the machine check pin enable bit ($HID0_{EMCP}$) is set. Certain machine check exceptions can be enabled and disabled through bit(s) in the HID0 register.
- Asynchronous, maskable non-critical exceptions (such as the External Input and Decrementer) are enabled by setting MSR_{EE}. When MSR_{EE}=0, recognition of these exception conditions is delayed. MSR_{EE} is cleared automatically when a non-critical or critical interrupt is taken to mask further recognition of conditions causing those exceptions.
- Asynchronous, maskable critical exceptions (such as Critical Input and Watchdog Timer) are enabled by setting MSR_{CE}. When MSR_{CE}=0, recognition of these exception conditions is delayed. MSR_{CE} is cleared automatically when a critical interrupt is taken to mask further recognition of conditions causing those exceptions.
- Synchronous and asynchronous Debug exceptions are enabled by setting MSR_{DE}. When MSR_{DE}=0, recognition of these exception conditions is masked. MSR_{DE} is cleared automatically when a Debug interrupt is taken to mask further recognition of conditions causing those exceptions. See Chapter 12, Debug Support, for more details on individual control of debug exceptions.

7.9.2 Returning from an interrupt handler

The return from interrupt (**rfi, se_rfi**), return from critical interrupt (**rfci, se_rfci**) return from debug interrupt (**rfdi, se_rfdi**), and return from machine check interrupt (**rfmci, se_rfmci**) instructions perform context synchronization by allowing previously-issued instructions to complete before returning to the interrupted process. In general, execution of return from interrupt type instructions ensures the following:

- All previous instructions have completed to a point where they can no longer cause an exception. This includes post-execute type exceptions.
- Previous instructions complete execution in the context (privilege and protection) under which they were issued.
- The **rfi and se_rfi** instructions copy SRR1 bits back into the MSR.
- The **rfci and se_rfci** instructions copy CSRR1 bits back into the MSR.
- The **rfdi and se_rfdi** instructions copy DSRR1 bits back into the MSR.
- The **rfmci and se_rfmci** instructions copy MCSRR1 bits back into the MSR.
- Instructions fetched after this instruction execute in the context established by this instruction.
- Program execution resumes at the instruction indicated by SRR0 for **rfi** and **se_rfi**, CSRR0 for **rfci** and **se_rfci**, MCCSRR0 for **rfmci** and **se_rfmci**, and DSRR0 for **rfdi** and **se_rfdi**.

Note that the return instructions **rfi** and **se_rfi** may be subject to a Return type debug exception, and that the return from critical interrupt instructions **rfci** and **se_rfci** may be subject to a Critical Return type debug exception. For a complete description of context synchronization, refer to *Book E: Enhanced PowerPC^{tm} Architecture*.

7.10 Process switching

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

- The **msync** instruction orders the effects of data memory instruction execution. All instructions previously initiated appear to have completed before the **msync** instruction completes, and no subsequent instructions appear to be initiated until the **msync** instruction completes.
- The **isync** instruction waits for all previous instructions to complete and then discards any fetched instructions, causing subsequent instructions to be fetched (or refetched) from memory and to execute in the context (privilege, translation, and protection) established by the previous instructions.
- The **stwcx.** instructions clears any outstanding reservations, ensuring that a load and reserve instruction in an old process is not paired with a store conditional instruction in a new one.

Chapter 8
Performance Monitor

This chapter describes the performance monitor, which is generally defined by the *Freescale EIS* and implemented as an APU on the e200z759n3 core. Although the programming model is defined by the EIS, some features are defined by the implementation; in particular, the events that can be counted.

8.1 Overview

The performance monitor provides the ability to count predefined events and processor clocks associated with particular operations, such as cache misses, mispredicted branches, or the number of cycles an execution unit stalls. The count of such events can be used to trigger the performance monitor interrupt.

The performance monitor can be used to do the following:

- Improve system performance by monitoring software execution and then recoding algorithms for more efficiency. For example, memory hierarchy behavior can be monitored and analyzed to optimize task scheduling or data distribution algorithms.
- Characterize processors in environments not easily characterized by benchmarking.
- Help system developers bring up and debug their systems.

The performance monitor comprises the following resources:

- The performance monitor mark bit in the MSR (MSR$_{PMM}$). This bit controls which programs are monitored.
- The move to/from performance monitor registers (PMR) instructions, **mtpmr** and **mfpmr**.
- The external inputs **p_pm_qual** and ***p_pm_event***.
- The external outputs **p_pmc0_ov**, **p_pmc1_ov**, **p_pmc2_ov**, and **p_pmc3_ov**
- PMRs:
  - — The performance monitor counter registers PMC0–PMC3 are 32-bit counters used to count software-selectable events. UPMC0–UPMC3 provide user-level read access to these registers. Counted events are those that should be of general value. They are identified in Table 8-10.
  - — The performance monitor global control register PMGC0 controls the counting of performance monitor events. It takes priority over all other performance monitor control registers. UPMGC0 provides user-level read access to PMGC0.
  - — The performance monitor local control registers PMLCa0–PMLCa3 and PMLCb0–PMLCb3 control individual performance monitor counters. Each counter has a corresponding PMLCa and PMLCb register. UPMLCa0–UPMLCa3 and UPMLCb0–UPMLCb3 provide user-level read access to PMLCa0–PMLCa3 and PMLCb0–PMLCb3.
- The performance monitor interrupt follows the Book E interrupt model and is assigned to interrupt vector offset register 35 (IVOR35). It has the lowest priority of all asynchronous interrupts.

Software communication with the performance monitor APU is achieved through PMRs rather than SPRs.

8.2 Performance Monitor APU instructions

The Performance Monitor APU defines the **mfpmr** and **mtpmr** instructions for reading and writing the PMRs as shown below.

mfpmr mfpmr

Move from Performance Monitor Register

mfpmr **r**D,**PMRN** Form: X

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	rD	$PMRN_{5:9}$	$PMRN_{0:4}$	0 1 0 1 0 0 1 1 1 0	/

```
GPR(rD) ← PMREG(PMRN)
```

The contents of the performance monitor register designated by PMRN are placed into GPR[**r**D].

When MSR_{PR} = 1, specifying a performance monitor register that is not implemented or is write-only and is not privileged (i.e. $PMRN_5$=0) results in an illegal instruction exception-type Program Interrupt. When MSR_{PR} = 1, specifying a performance monitor register that is not implemented or is write-only and is privileged (i.e. $PMRN_5$=1) results in a privileged instruction exception-type Program Interrupt. When MSR_{PR} = 0, specifying a performance monitor register that is not implemented or is write-only results in an illegal instruction exception type Program Interrupt.

mtpmr mtpmr

Move to Performance Monitor Register

mtpmr **PMRN, r**S Form: X

0 5	6 10	11 15	16 20	21 30	31
0 1 1 1 1 1	rS	$PMRN_{5:9}$	$PMRN_{0:4}$	0 1 1 1 0 0 1 1 1 0	/

```
PMREG(PMRN) ← GPR(rS)
```

The contents of GPR[**r**S] are placed into the performance monitor register designated by PMRN.

When MSR_{PR} = 1, specifying a performance monitor register that is not implemented or is read-only and is not privileged (i.e. $PMRN_5$=0) results in an illegal instruction exception-type Program Interrupt. When MSR_{PR} = 1, specifying a performance monitor register that is not implemented or is read-only and is privileged (i.e. $PMRN_5$=1) results in a privileged instruction exception-type Program Interrupt. When MSR_{PR} = 0, specifying a performance monitor register that is not implemented or is read-only results in an illegal instruction exception type Program Interrupt.

8.3 Performance Monitor APU registers

The *Freescale EIS* defines a set of register resources used exclusively by the performance monitor. PMRs are similar to the SPRs defined in the Book E architecture and are accessed by **mtpmr** and **mfpmr** instructions, which are also defined by the *Freescale EIS*. Table 8-1 lists supervisor-level (privileged) PMRs.

Table 8-1. Supervisor-level PMRs (PMR[5] = 1)

Name	Register name	PMR number	pmr[0–4]	pmr[5–9]	Section/ page
PMC0	Performance monitor counter 0	16	00000	10000	8.3.9/8-540
PMC1	Performance monitor counter 1	17	00000	10001	
PMC2	Performance monitor counter 2	18	00000	10010	
PMC3	Performance monitor counter 3	19	00000	10011	
PMGC0	Performance monitor global control register 0	400	01100	10000	8.3.3/8-532
PMLCa0	Performance monitor local control a0	144	00100	10000	8.3.5/8-534
PMLCa1	Performance monitor local control a1	145	00100	10001	
PMLCa2	Performance monitor local control a2	146	00100	10010	
PMLCa3	Performance monitor local control a3	147	00100	10011	
PMLCb0	Performance monitor local control b0	272	01000	10000	8.3.7/8-535
PMLCb1	Performance monitor local control b1	273	01000	10001	
PMLCb2	Performance monitor local control b2	274	01000	10010	
PMLCb3	Performance monitor local control b3	275	01000	10011	

User-level PMRs in Table 8-2 are read-only and are accessed with **mfpmr**.

Table 8-2. User-level PMRs (PMR[5] = 0) (read-only)

Name	Register Name	PMR Number	pmr[0–4]	pmr[5–9]	Section/ Page
UPMC0	User performance monitor counter 0	0	00000	00000	8.3.10/8-541
UPMC1	User performance monitor counter 1	1	00000	00001	
UPMC2	User performance monitor counter 2	2	00000	00010	
UPMC3	User performance monitor counter 3	3	00000	00011	

Table 8-2. User-level PMRs (PMR[5] = 0) (read-only) (continued)

Name	Register Name	PMR Number	pmr[0–4]	pmr[5–9]	Section/ Page
UPMGC0	User performance monitor global control register 0	384	01100	00000	8.3.4/8-534
UPMLCa0	User performance monitor local control a0	128	00100	00000	8.3.6/8-535
UPMLCa1	User performance monitor local control a1	129	00100	00001	
UPMLCa2	User performance monitor local control a2	130	00100	00010	
UPMLCa3	User performance monitor local control a3	131	00100	00011	
UPMLCb0	User performance monitor local control b0	256	01000	00000	8.3.8/8-540
UPMLCb1	User performance monitor local control b1	257	01000	00001	
UPMLCb2	User performance monitor local control b2	258	01000	00010	
UPMLCb3	User performance monitor local control b3	259	01000	00011	

8.3.1 Invalid PMR references

Behavior when an invalid PMR is referenced depends on the privilege level of the register and MSR_{PR}. Table 8-3 shows the response for various references to invalid PMRs.

Table 8-3. Response to an invalid PMR reference

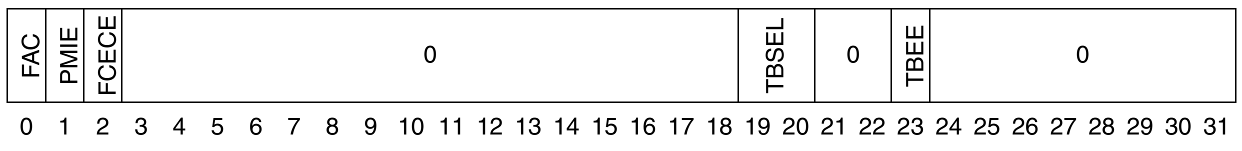
PMR address bit 5	MSR_{PR}	Response
0 (user)	x	Illegal exception
1 (supervisor)	0 (supervisor)	Illegal exception
	1 (user)	Privileged exception

8.3.2 References to read-only PMRs

If a **mtpmr** instruction is executed to a read-only PMR, e200z759n3 will take an Illegal exception.

8.3.3 Performance Monitor Global Control Register 0 (PMGC0)

The performance monitor global control register PMGC0 shown in Figure 8-1 controls all performance monitor counters.



PMR - 400; Read/Write; Reset - 0x0

Figure 8-1. Performance Monitor Global Control Register (PMGC0)

PMGC0 is cleared by reset. Reading this register does not change its contents. Table 8-4 describes PMGC0 fields.

Table 8-4. PMGC0 field descriptions

Bits	Name	Description
0 (32)	FAC	Freeze All Counters. 0 The PMCs are incremented (if permitted by other PMGC/PMLC control bits). 1 The PMCs are not incremented. When FAC is set to 1 by hardware or software, it has no effect on PMLCax_{FC}; PMLCax_{FC} maintains it's current value until changed by software. FAC setting by hardware is controlled by PMGC0$_{FCECE.}$
1 (33)	PMIE	Performance monitor interrupt enable 0 Performance monitor interrupts are disabled. 1 Performance monitor interrupts are enabled and occur when an enabled condition or event occurs, at which time PMGC0$_{PMIE}$ is cleared Software can clear PMIE to prevent performance monitor interrupts. Performance monitor interrupts are caused by time base events or PMCx counter overflows.
2 (34)	FCECE	Freeze Counters on Enabled Condition or Event 0 The PMCs can be incremented (if permitted by other PM control bits). 1 The PMCs can be incremented (if permitted by other PM control bits) only until an enabled condition or event occurs. When an enabled condition or event occurs, PMGC0$_{FAC}$ is set to 1. It is up to software to clear PMGC0$_{FAC}$ to 0. An enabled condition or event is defined as one of the following: • When the msb = 1 in PMC*x* and PMLCax_{CE} = 1. • When the time-base bit specified by PMGC0$_{TBSEL}$ transitions to 1 and PMGC0$_{TBEE}$=1. The use of the trigger and freeze counter conditions depends on the enabled conditions and events described in Section 7.2, “Performance Monitor Interrupt.”
3:18 (35:50)	—	Reserved, should be cleared.
19:20 (51:52)	TBSEL	Time Base Selector. Selects the time base bit that can cause a time base transition event (the event occurs when the selected bit changes from 0 to 1). 00 TB_{63} (TBL_{31}) 01 TB_{55} (TBL_{23}) 10 TB_{51} (TBL_{19}) 11 TB_{47} (TBL_{15}) Time-base frequency is implementation-dependent, so software should invoke a system service program to obtain the frequency before choosing a TBSEL value.
21:22 (53:54)	—	Reserved, should be cleared.
23 (55)	TBEE	Time base transition Event Enable 0 Time base transition events are disabled. 1 Time base transition events are enabled. A time base transition is signaled to the performance monitor if the TB bit specified in PMGC0$_{TBSEL}$ changes from 0 to 1. Time base transition events can be used to freeze counters (PMGC0$_{FCECE}$) or signal an exception (PMGC0$_{PMIE)}$. Although the exception signal condition may occur with MSR$_{EE}$ = 0, the interrupt cannot be taken until MSR$_{EE}$ = 1. Changing PMGC0$_{TBSEL}$ while PMGC0$_{TBEE}$ is enabled may cause a false 0 to 1 transition that signals the specified action (freeze, exception) to occur immediately.
24:31 (56:63)	—	Reserved, should be cleared.

8.3.4 User Performance Monitor Global Control Register 0 (UPMGC0)

UPMGC0 provides user-level read access to PMGC0. UPMGC0 can be read by user-level software with the **mfpmr** instruction using PMR 384.

8.3.5 Performance Monitor Local Control A Registers (PMLCa0–PMLCa3)

The local control A registers (PMLCa0–PMLCa3) function as event selectors and give local control for the corresponding performance monitor counters. PMLCa is used in conjunction with the corresponding PMLCb register. PMLCa registers are shown in Figure 8-2.

FC	FCS	FCU	FCM1	FCM0	CE	0		EVENT								0	PMP			0											
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

PMR - 144, 145, 146, 147; Read/Write; Reset - 0x0

Figure 8-2. Performance Monitor Local Control A Registers (PMLCa0–PMLCa3)

PMLCa registers are cleared by reset. Table 8-5 describes PMLCa fields.

Table 8-5. PMLCa0–PMLCa3 field descriptions

Bits	Name	Description
0 (32)	FC	Freeze Counter. 0 The PMC can be incremented (if enabled by other performance monitor control fields). 1 The PMC will not be incremented.
1 (33)	FCS	Freeze Counter in Supervisor state. 0 The PMC can be incremented (if enabled by other performance monitor control fields). 1 The PMC will not be incremented if MSR_{PR} is cleared.
2 (34)	FCU	Freeze Counter in User state. 0 The PMC can be incremented (if enabled by other performance monitor control fields). 1 The PMC will not be incremented if MSR_{PR} is set.
3 (35)	FCM1	Freeze Counter while Mark is set. 0 The PMC can be incremented (if enabled by other performance monitor control fields). 1 The PMC will not be incremented if MSR_{PMM} is set.
4 (36)	FCM0	Freeze Counter while Mark is cleared. 0 The PMC can be incremented (if enabled by other performance monitor control fields). 1 The PMC will not be incremented if MSR_{PMM} is cleared.
5 (37)	CE	Condition Enable. 0 verflow conditions for PMC*n* cannot occur (PMC*n* cannot cause interrupts or freeze counters) 1 An overflow condition is present when the most-significant-bit of PMC*n* is equal to 1. It is recommended that CE be cleared when counter PMC*n* is selected for chaining.
6:7 (38:39)	—	Reserved for EVENT expansion, should be cleared.
8:15 (40:47)	EVENT	Event selector. See Section 8.7, Event selection

Table 8-5. PMLCa0–PMLCa3 field descriptions (continued)

Bits	Name	Description
16 (48)	—	Reserved, should be cleared.
17:19 (49:51)	PMP	Performance Monitor Watchpoint Periodicity Select 000 Performance Monitor Watchpoint *x* asserts on any change of counter$_x$ bit 32 (period=2^{31}) 001 Performance Monitor Watchpoint *x* asserts on any change of counter$_x$ bit 43 (period=2^{20}) 010 Performance Monitor Watchpoint *x* asserts on any change of counter$_x$ bit 49 (period=2^{14}) 011 Performance Monitor Watchpoint *x* asserts on any change of counter$_x$ bit 55 (period=2^{8}) 100 Performance Monitor Watchpoint *x* asserts on any change of counter$_x$ bit 59 (period=2^{4}) 101 Performance Monitor Watchpoint *x* asserts on any change of counter$_x$ bit 61 (period=2^{2}) 110 Performance Monitor Watchpoint *x* asserts on any change of counter$_x$ bit 62 (period=2^{1}) 111 Performance Monitor Watchpoint *x* asserts on any change of counter$_x$ bit 63 (period=2^{0})[1]
20:31 (52:63)	—	Reserved, should be cleared.

[1] For certain events that may count an even number of times per cycle, this watchpoint is not guaranteed to assert with PMP=111.

8.3.6 User Performance Monitor Local Control A Registers (UPMLCa0–UPMLCa3)

The PMLCa register contents are aliased to UPMLCa0–UPMLCa3, which can be read by user-level software with **mfpmr** using PMR numbers in Table 8-2.

8.3.7 Performance Monitor Local Control B Registers (PMLCb0–PMLCb3)

Local control B registers PMLCb0–PMLCb3) specify triggering conditions, a threshold value and a multiple to apply to a threshold event selected for the corresponding performance monitor counter. For the e200z759n3, thresholding is supported only for PMC0 and PMC1. PMLCb is used in conjunction with the corresponding PMLCa register.

0	1–3	4	5–7	8	9–12	13	14–17	18	19–20	21–23	24–25	26–31
0	TRIGONCTL	0	TRIGOFFCTL	0	TRIGONSEL	0	TRIGOFFSEL	TRIGGERED	0	THRESHMUL	0	THRESHOLD

PMR - 272, 273, 274, 275; Read/Write; Reset - 0x0

Figure 8-3. Performance Monitor Local Control B Registers (PMLCb0–PMLCb3)

PMLCb is cleared by reset. Table 8-6 describes PMLCb fields.

Table 8-6. PMLCb0–PMLCb3 field descriptions

Bits	Name	Description
0 (32)	—	Reserved, should be cleared.
1:3 (33:35)	TRIGONCNTL	Trigger-on Control Class - Class of Trigger-on source 000 Trigger-on control is disabled if TRIGONSEL is 0000 (i.e. counting is not affected by triggers). All other values for TRIGONSEL are reserved. 001 Trigger-on control based on selected PMC condition(s) 010 Trigger-on based on selected processor event(s) 011 Trigger-on based on selected hardware signal(s) 100Trigger-on based on selected watchpoint occurrence (watchpoint #0–15) 101 Trigger-on based on selected watchpoint occurrence (extension for watchpoint #16-31) 11x Reserved Indicates the condition under which triggering to start counting occurs. No triggering will occur while PMGC0$_{FAC}$ or PMLCan_{FC} is set to '1'.
4 (36)	—	Reserved, should be cleared.
5:7 (37:39)	TRIGOFFCNTL	Trigger-off Control Class - Class of Trigger-off source 000 Trigger-off control is disabled if TRIGOFFSEL is 0000 (i.e. counting is not affected by triggers) All other values for TRIGOFFSEL are reserved. 001 Trigger-off control based on selected PMC condition(s) 010 Trigger-off based on selected processor event(s) 011 Trigger-off based on selected hardware signal(s) 100 Trigger-off based on selected watchpoint occurrence (watchpoint #0–15) 101 Trigger-off based on selected watchpoint occurrence (extension for watchpoint #16-31) 11x Reserved Indicates the condition under which triggering to stop counting occurs. No triggering will occur while PMGC0$_{FAC}$ or PMLCan_{FC} is set to '1'.
8 (40)	—	Reserved, should be cleared.

Table 8-6. PMLCb0–PMLCb3 field descriptions (continued)

Bits	Name	Description
9:12 (41:44)	TRIGONSEL	Trigger-on Source Select - Source Select based on setting of TRIGONCTL TRIGONCTL = 000: 0000 Trigger-on control is disabled 0001 –1111 : Reserved TRIGONCTL = 001: This field should be to the ID of the PMCy that should trigger event counting to start. When PMCy overflows, the trigger will be generated. When TRIGONSEL = PMCx (i.e. self-select), no triggering will occur due to any counter change. If TRIGONSEL = TRIGOFFSEL, triggering results are undefined. 0000 Trigger-on when $PMC0_{OV}$ transitions to a ‘1’. 0001 Trigger-on when $PMC1_{OV}$ transitions to a ‘1’. 0010 Trigger-on when $PMC2_{OV}$ transitions to a ‘1’. 0011 Trigger-on when $PMC3_{OV}$ transitions to a ‘1’. 0100 – 1111 : Reserved TRIGONCTL = 010: 0000 Trigger-on when next processor interrupt occurs (software may want to set $PMGC0_{PMIE}$ = 0 for this setting). 0001 – 1111 : Reserved TRIGONCTL = 011: 0000 Trigger on assertion of **p_devnt_out[0]** 0001 Trigger on assertion of **p_devnt_out[1]** 0010 Trigger on assertion of **p_devnt_out[2]** 0011 Trigger on assertion of **p_devnt_out[3]** 0100 Trigger on assertion of **p_devnt_out[4]** 0101 Trigger on assertion of **p_devnt_out[5]** 0110 Trigger on assertion of **p_devnt_out[6]** 0111 Trigger on assertion of **p_devnt_out[7]** 1000 Trigger on rise of **p_pmc*n*_qual** input 1001 – 1111 : Reserved TRIGONCTL = 100: 0000 Trigger-on based on watchpoint #0 occurrence 0001 Trigger-on based on watchpoint #1 occurrence 0010 Trigger-on based on watchpoint #2 occurrence . . . 1110 Trigger-on based on watchpoint #14 occurrence 1111 Trigger-on based on watchpoint #15 occurrence TRIGONCTL = 101: 0000 Trigger-on based on watchpoint #16 occurrence 0001 Trigger-on based on watchpoint #17 occurrence 0010 Trigger-on based on watchpoint #18 occurrence . . . 1100 Trigger-on based on watchpoint #28 occurrence 1101 Trigger-on based on watchpoint #29 occurrence 1110 – 1111 : Reserved
13 (45)	—	Reserved, should be cleared.

Table 8-6. PMLCb0–PMLCb3 field descriptions (continued)

Bits	Name	Description
14:17 (46:49)	TRIGOFFSEL	Trigger-off Source Select - Source Select based on setting of TRIGOFFCTL TRIGOFFCTL = 000: 0000 Trigger-off control is disabled 0001 – 1111 : Reserved TRIGOFFCTL = 001: This field should be to the ID of the PMCy that should trigger event counting to stop. When PMCy overflows, the trigger will be generated. When TRIGOFFSEL = PMCx (i.e. self-select), no triggering will occur due to any counter change. If TRIGONSEL = TRIGOFFSEL, triggering results are undefined. 0000 Trigger-off when $PMC0_{OV}$ transitions to a ‘1’. 0001 Trigger-off when $PMC1_{OV}$ transitions to a ‘1’. 0010 Trigger-off when $PMC2_{OV}$ transitions to a ‘1’. 0011 Trigger-off when $PMC3_{OV}$ transitions to a ‘1’. 0100 – 1111 : Reserved TRIGOFFCTL = 010: 0000 Trigger-on when next processor interrupt occurs (software may want to set $PMGC0_{PMIE}$ = 0 for this setting). 0001 – 1111 : Reserved TRIGOFFCTL = 011: 0000 Trigger-off based on assertion of **p_devnt_out[0]** 0001 Trigger-off based on assertion of **p_devnt_out[1]** 0010 Trigger-off based on assertion of **p_devnt_out[2]** 0011 Trigger-off based on assertion of **p_devnt_out[3]** 0100 Trigger-off based on assertion of **p_devnt_out[4]** 0101 Trigger-off based on n assertion of **p_devnt_out[5]** 0110 Trigger-off based on assertion of **p_devnt_out[6]** 0111 Trigger-off based on assertion of **p_devnt_out[7]** 1000 Trigger-off based on fall of **p_pmc*n*_qual** input 1001 – 1111 : Reserved TRIGOFFCTL = 100: 0000 Trigger-off based on watchpoint #0 occurrence 0001 Trigger-off based on watchpoint #1 occurrence 0010 Trigger-off based on watchpoint #2 occurrence . . . 1110 Trigger-off based on watchpoint #14 occurrence 1111 Trigger-off based on watchpoint #15 occurrence TRIGOFFCTL = 101: 0000 Trigger-off based on watchpoint #16 occurrence 0001 Trigger-off based on watchpoint #17 occurrence 0010 Trigger-off based on watchpoint #18 occurrence . . . 1100 Trigger-off based on watchpoint #28 occurrence 1101 Trigger-off based on watchpoint #29 occurrence 1110 – 1111 : Reserved

Table 8-6. PMLCb0–PMLCb3 field descriptions (continued)

Bits	Name	Description
18 (50)	TRIGGERED	Triggered 0 Counter has not been triggered 1 Counter has been triggered TRIGGERED can be set or cleared by hardware or software. TRIGGERED setting by hardware is controlled by PMLCbx$_{TRIGONCTL}$. If PMLCbx$_{TRIGONCTL}$ is set to enable trigger-on control, TRIGGERED will be set by hardware when the next trigger-on event occurs and TRIGGERED is currently cleared. TRIGGERED clearing by hardware is controlled by PMLCbx$_{TRIGOFFCTL}$. If PMLCbx$_{TRIGOFFCTL}$ is set to enable trigger-off control, TRIGGERED will be cleared by hardware when the next trigger-off event occurs and TRIGGERED is currently set. The state of TRIGGERED qualifies counting if either PMLCbx$_{TRIGONCTL}$ or PMLCbx$_{TRIGOFFCTL}$ is set to enable triggering (other qualifiers on counting such as PMGC0$_{FAC}$ and PMLCa controls operate independently of TRIGGERED). If both PMLCbx$_{TRIGONCNTL}$ and PMLCbx$_{TRIGOFFCTL}$ are cleared to disable triggering, the state of TRIGGERED has no effect on counting. TRIGGERED has no effect on PMLCa*x*$_{FC}$; PMLCa*x*$_{FC}$ maintains it's current value until changed by software.
19:20 (51:52)	—	Reserved, should be cleared.
21:23 (53:55)	THRESHMUL[1]	Threshold multiple. 000 Threshold field is multiplied by 1 (PMLCb*n*$_{THRESHOLD}$ × 1) 001 Threshold field is multiplied by 2 (PMLCb*n*$_{THRESHOLD}$ × 2) 010 Threshold field is multiplied by 4 (PMLCb*n*$_{THRESHOLD}$ × 4) 011 Threshold field is multiplied by 8 (PMLCb*n*$_{THRESHOLD}$ × 8) 100 Threshold field is multiplied by 16 (PMLCb*n*$_{THRESHOLD}$ × 16) 101 Threshold field is multiplied by 32 (PMLCb*n*$_{THRESHOLD}$ × 32) 110 Threshold field is multiplied by 64 (PMLCb*n*$_{THRESHOLD}$ × 64) 111 Threshold field is multiplied by 128 (PMLCb*n*$_{THRESHOLD}$ × 128)
24:25 (56:57)	—	Reserved, should be cleared.
26:31 (58:63)	THRESHOLD[1]	Threshold Only events that exceed this value multiplied by THRESHMUL are counted. Events to which a threshold value applies are implementation dependent, as are the unit (for example duration in cycles) and the granularity with which the threshold value is interpreted. By varying the threshold value, software can obtain a profile of the event characteristics subject to thresholding by monitoring a program repeatedly using a different threshold value each time.

[1] These Fields are not implemented in PMLCb2 and PMLCb3, and read as zero.

8.3.8 User Performance Monitor Local Control B registers (UPMLCb0–UPMLCb3)

The contents of PMLCb0–PMLCb3 are aliased to UPMLCb0–UPMLCb3, which can be read by user-level software with **mfpmr** using PMR numbers in Table 8-2.

8.3.9 Performance Monitor Counter registers (PMC0–PMC3)

The performance monitor counter registers PMC0–PMC3 shown in Figure 8-4 are 32-bit counters that can be programmed to generate overflow event signals when they overflow. Each counter is enabled to count up to 128 processor events.

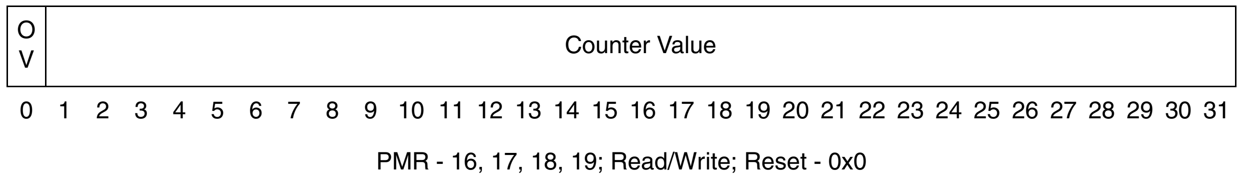


Figure 8-4. Performance Monitor Counter registers (PMC0–PMC3)

PMCs are cleared by reset. Table 8-7 describes the PMC register fields.

Table 8-7. PMC0–PMC3 field descriptions

Bits	Name	Description
0 (32)	OV	Overflow 0 - Counter has not reached an overflow state. 1 - Counter has reached an overflow state. Note: this bit is not sticky, thus will not remain set if the counter subsequently counts past 0xFFFF_FFFF.
1:31 (33:63)	Counter Value	Indicates the number of occurrences of the specified event.

The minimum value for a counter is 0 (0x0000_0000) and the maximum value is 4,294,967,295 (0xFFFF_FFFF). A counter can increment by 0, 1, 2, 3, or 4 (based on the number of events occurring in a given counter cycle) up to the maximum value and then wraps to the minimum value.

A counter enters the overflow state when the high-order bit is set. A performance monitor interrupt handler can easily identify overflowed counters, even if the interrupt is masked for many cycles (during which the counters may continue incrementing). A high-order bit is normally set only when the counter increments from a value below 2,147,483,648 (0x8000_0000) to a value greater than or equal to 2,147,483,648 (0x8000_0000).

NOTE

Initializing PMCs to overflowed values is discouraged. If an overflowed value is loaded into a PMC*n* that held a non-overflowed value (and $\text{PMGC0}_{\text{PMIE}}$, $\text{PMLCa}n_{\text{CE}}$, and MSR_{EE} are set), an interrupt may be falsely generated before any events are counted.

The response to an overflow condition depends on the configuration, as follows:

- If PMLCan_{CE} is clear, no special actions occur on overflow of PMCn: the counter continues incrementing, and no event is signaled.
- If PMLCan_{CE} and PMGC0$_{FCECE}$ are both set, all counters are frozen when PMCn overflows.
- If PMLCan_{CE} and PMGC0$_{PMIE}$ are set, an exception is signaled on overflow of PMCn. Performance Monitor Interrupts are masked when MSR$_{EE}$ =0. An exception may be signaled while MSR$_{EE}$ =0, but the interrupt is not taken until MSR$_{EE}$ =1 and is only guaranteed to be taken if the overflow condition is still present (i.e., the counter has not counted past 0xFFFF_FFFF, in which case the OV bit would become cleared) and the configuration has not been changed in the meantime to disable the exception. If PMLCan_{CE} or PMGC0$_{PMIE}$ is cleared, the exception is no longer signaled.

The following sequence is recommended for setting counter values and configurations:

1. Set PMGC0$_{FAC}$ to freeze the counters.
2. Using **mtpmr** instructions, initialize counters and configure control registers.
3. Release the counters by clearing PMGC0$_{FAC}$ with a final **mtpmr**.

8.3.10 User Performance Monitor Counter registers (UPMC0–UPMC3)

The contents of PMC0–PMC3 are aliased to UPMC0–UPMC3, which can be read by user-level software with the **mfpmr** instruction using PMR numbers in Table 8-2.

8.4 Performance monitor interrupt

The performance monitor interrupt is triggered by an enabled condition or event. The enabled condition or events defined for the e200z759n3 are the following:

- A PMCn overflow condition occurs when both of the following are true:
  — The counter's overflow condition is enabled; PMLCan_{CE} is set.
  — The counter indicates an overflow; PMCn_{OV} is set.
- A time base event occurs with the following settings:
  — Time base events are enabled with PMGC0$_{TBEE}$ = 1
  — The TBL bit specified in PMGC0$_{TBSEL}$ changes from 0 to 1

The two performance monitor exception conditions are treated differently with respect to whether or not the conditions are level-sensitive or edge-sensitive. A performance monitor exception condition that is caused by a PMCn overflow condition is level-sensitive to the values of PMLCAn_{CE} and PMCn_{OV}. This means that as long as these values are both set to '1', then the exception condition continues to exist and the performance monitor interrupt can be taken if the remainder of the performance monitor interrupt gating conditions are met. However, the exception due to the time base event is set only when both PMGC0$_{TBEE}$=1 and the transition from '0' to '1' occurs in the specified TBL bit. This condition is not cleared once it occurs, regardless of whether the TBL bit subsequently transitions to a '0', but this exception is automatically cleared whenever any performance monitor interrupt is subsequently taken.

If PMGC0$_{PMIE}$ is set, an enabled condition or event triggers the signaling of a performance monitor exception.

If PMGC0$_{FCECE}$ is set, an enabled condition or event forces all performance monitor counters to freeze.

Although the performance monitor exception condition may occur with MSR$_{EE}$ =0, the interrupt cannot be taken until MSR$_{EE}$ =1. If PMC*n* overflows and would signal an exception (PMLCa*n*$_{CE}$=1 and PMGC0$_{PMIE}$=1) while MSR$_{EE}$ =0, and freezing of the counters is not enabled (PMGC0$_{FCECE}$ is clear), it is possible that PMC*n* could wrap around to all zeros again without the performance monitor interrupt being taken.

Interrupt handlers should clear a counter overflow condition or the corresponding Condition Enable to avoid a repeated interrupt to occur for the same event.

The priority of the performance monitor interrupt is specified in Section 7.8.1, Exception priorities.

8.5 Event counting

This section describes configurability and specific unconditional counting modes.

8.5.1 MSR-based context filtering

Counting can be configured to be conditionally enabled if conditions in the processor state match a software-specified condition. Because a software task scheduler may switch a processor’s execution among multiple processes and because statistics on only a particular process may be of interest, a facility is provided to mark a process. The performance monitor mark bit, MSR$_{PMM}$, is used for this purpose. System software may set this bit when a marked process is running. This enables statistics to be gathered only during the execution of the marked process. The states of MSR$_{PR}$ and MSR$_{PMM}$ define a state that the processor (supervisor or user) and the process (marked or unmarked) may be in at any time. If this state matches an individual state specified by the PMLCa*n*$_{FCS,FCU,FCM1,FCM0}$ fields, counting is enabled for PMC*n*.

For the e200z759n3 implementation, a given event may or may not support MSR-based context filtering. For events that do not support MSR-based context filtering, the FCS, FCU, FCM1, and FCM0 controls have no effect on the counting of that event.

The processor states and the settings of the FCS, FCU, FCM1, and FCM0 bits in PMLCa*n* necessary to enable monitoring of each processor state are shown in Table 8-8.

Table 8-8. Processor States and PMLCa0–PMLCa3 bit settings

Processor State	FCS	FCU	FCM1	FCM0
All (no context filtering)	0	0	0	0
Marked	0	0	0	1
Not marked	0	0	1	0
Supervisor	0	1	0	0
Marked and supervisor	0	1	0	1

Table 8-8. Processor States and PMLCa0—PMLCa3 bit settings (continued)

Processor State	FCS	FCU	FCM1	FCM0
Not marked and supervisor	0	1	1	0
User	1	0	0	0
Marked and user	1	0	0	1
Not marked and user	1	0	1	0
None (counting disabled)	X	X	1	1
None (counting disabled)	1	1	X	X

8.6 Examples

The following sections provide examples of how to use the performance monitor facility.

8.6.1 Chaining counters

The counter chaining feature can be used to allow a higher event count than is possible with a single counter. Chaining two counters together effectively adds 32 bits to a counter register where rollover of the first counter generates a carry out feeding the second counter. By defining the event of interest to be another PMC's rollover occurrence, the chained counter increments each time the first counter rolls over to zero. Multiple counters may be chained together.

Because the entire chained value cannot be read in a single instruction, a rollover may occur between counter reads, producing an inaccurate value. A sequence like the following is necessary to read the complete chained value when it spans multiple counters and the counters are not frozen. The example shown is for a two-counter case.

```
loop:   mfpmr             Rx,pmctr1         #load from upper counter
        mfpmr             Ry,pmctr0         #load from lower counter
        mfpmr             Rz,pmctr1         #load from upper counter
        cmp               cr0,0,Rz,Rx       #see if ‘old’ = ‘new’
        bc                4,2,loop          #loop if carry occurred between reads
```

The comparison and loop are necessary to ensure that a consistent set of values has been obtained. The above sequence is not necessary if the counters are frozen.

8.6.2 Thresholding

Threshold event measurement enables the counting of duration and usage events. For example, data cache load miss cycles (events C0:xx and C1:xx) require a threshold value. A data cache load miss cycles event is counted only when the number of cycles spent waiting for the miss is greater than the threshold. Because this event is supported by two counters and each counter has an individual threshold, one execution of a performance monitor program can sample two different threshold values. Measuring code performance with multiple concurrent thresholds may expedite code profiling significantly.

8.7 Event selection

Event selection is specified through the PMLCa*n* registers described in Section 8.3.5, Performance Monitor Local Control A Registers (PMLCa0–PMLCa3). The event-select fields in PMLCan_{EVENT} are described in Table 8-10, which lists encodings for the selectable events to be monitored. Table 8-10 establishes a correlation between each counter, events to be traced, and the pattern required for the desired selection.

The Spec/Nonspec column indicates whether the event count includes any occurrences due to processing that was not architecturally required by the Power Architecture sequential execution model (speculative processing).

- Speculative counts include speculative operations that were later flushed.
- Nonspeculative counts do not include speculative operations, which are flushed.

The PR, PMM filtering column indicates whether a given event supports MSR-based context filtering.

Table 8-9 describes how event types are indicated in Table 8-10.

Table 8-9. Event types

Event type	Label	Description
Reference	Ref:#	Shared across counters PMC0–PMC3.
Common	Com:#	Shared across counters PMC0–PMC3.
Counter-specific	C[0–3]:#	Counted only on one or more specific counters. The notation indicates the counter to which an event is assigned. For example, an event assigned to counter PMC0 is shown as C0:#.

Table 8-10 describes performance monitor events.

Table 8-10. Performance monitor event selection

Number	Event	Spec/ nonspec	PR, PMM filtering[1]	Count description
General events				
Com:0	Nothing	Nonspec	-	Register counter holds current value
Ref:1[2]	Processor cycles	Nonspec	yes	Every processor cycle not in waiting, halted, stopped states and not in a debug session.
Com:2[3]	Instructions completed	Nonspec	yes	Completed instructions. 0, 1, 2, or 3 per cycle.
Com:3[2]	Processor cycles with 0 instructions issued	Nonspec	yes	Ref:1 cycles with no instructions entering execution
Com:4[2]	Processor cycles with 1 instruction issued	Nonspec	yes	Ref:1 cycles with one instruction entering execution
Com:5[2]	Processor cycles with 2 instructions issued	Nonspec	yes	Ref:1 cycles with two instructions entering execution

Table 8-10. Performance monitor event selection (continued)

Number	Event	Spec/ nonspec	PR, PMM filtering[1]	Count description
Com:6[3]	Instruction words fetched	Spec	yes	Fetched instruction words. 0, 1, or 2, 3, or 4 per cycle. (note that an instruction word may hold 1 or 2 instructions, or 2 partial instructions when fetching from a VLE page)
Com:7	—	—	—	—
Com:8	PM_EVENT transitions	—	—	0 to 1 transitions on the *p_pm_event* input.
Com:9	PM_EVENT cycles	—	—	Processor (Ref:1) cycles that occur when the *p_pm_event* input is asserted.
Instruction types completed				
Com:10[3]	Branch instructions completed	Nonspec	yes	Completed branch instructions, includes branch and link type instructions
Com:11[3]	Branch and link type instructions completed	Nonspec	yes	Completed branch and link type instructions
Com:12[3]	Conditional branch instructions completed	Nonspec	yes	Completed conditional branch instructions
Com:13[3]	Taken Branch instructions completed	Nonspec	yes	Completed branch instructions that were taken. Includes branch and link type instructions.
Com:14[3]	Taken Conditional Branch instructions completed	Nonspec	yes	Completed conditional branch instructions that were taken.
Com:15[3]	Load instructions completed	Nonspec	yes	Completed load, load-multiple type instructions
Com:16[3]	Store instructions completed	Nonspec	yes	Completed store, store-multiple type instructions
Com:17[3]	Load micro-ops completed	Nonspec	yes	Completed load micro-ops. (**l***, **evl***, load-update (1 load micro-op), load-multiple (1–32 micro-ops), **dcbt**, **dcbtls**, **dcbtst**, **dcbtstls**, and **dcbtst**, **dcbf**, **dcblc**, **dcbst**, **icbi**, **icblc**, **icbt**, **icbtls)**. Misaligned loads crossing a 64-bit boundary count as two micro-ops.
Com:18[3]	Store micro-ops completed	Nonspec	yes	Completed store micro-ops. (**st***, **evst***, store-update (1 store micro-op), store-multiple (1–32 micro-ops), **dcbi**, **dcbz**). Misaligned stores crossing a 64-bit boundary count as two micro-ops.
Com:19[3]	Integer instructions completed	Nonspec	yes	Completed simple integer instructions (not a load-type/store-type/branch/mul/div, EFPU, or SPE)
Com:20[3]	Multiply instructions completed	Nonspec	yes	Completed Multiply instructions (non-EFPU)
Com:21[3]	Divide instructions completed	Nonspec	yes	Completed Divide instructions including SPE (non-EFPU)

Table 8-10. Performance monitor event selection (continued)

Number	Event	Spec/ nonspec	PR, PMM filtering[1]	Count description
Com:22[3]	Divide instruction execution cycles	Nonspec	yes	Cycles of execution for all Divide instructions (non-EFPU)
Com:23[3]	SPE/EFPU instructions completed	Nonspec	yes	Completed SPE/EFPU instructions. Does not include SPE/EFPU load and store instructions.
Com:24[3]	SPE simple instructions completed	Nonspec	yes	Completed SPE simple instructions. All SPE instructions included except SPE load and store instructions, div, dotp, mul and mac-type instructions.
Com:25[3]	SPE mul/mac/dotp instructions completed	Nonspec	yes	Completed SPE mul/mac/dotp instructions. Does not include other SPE instructions, or **brinc** instructions.
Com:26[3]	EFPU FP instructions completed	Nonspec	yes	Completed EFPU FP (evfs, efs) instructions.
Com:27[3]	Number of return from interrupt instructions	Nonspec	yes	Includes all types of return from interrupts (i.e. **rfi, rfci, rfdi, rfmci,** and VLE variants)
Branch prediction and execution events				
Com:28[3]	Finished branches that miss the BTB	Spec	yes	Includes all taken branch instructions that missed in the BTB
Com:29[3]	Branches mispredicted (for any reason)	Spec	yes	Counts branch instructions mispredicted due to direction or target (for example if the LR or CTR contents change).
Com:30[3]	Branches in the BTB mispredicted due to direction prediction.	Spec	yes	Counts branch instructions that hit the BTB with mispredicted due to direction prediction.
Com:31[3]	Incorrect target prediction using the link stack	Spec	yes	—
Com:32[3]	BTB hits	Spec	yes	Branch instructions that hit in the BTB
Com:33	—	—	—	—
Com:34	—	—	—	—
Pipeline stalls				
Com:35	—	—	—	—
Com:36	—	—	—	—
Com:37[2]	Cycles decode stalled due to no instructions available	Spec	yes	No instruction available to decode
Com:38[2]	Cycles issue stalled	Spec	yes	Cycles the issue buffer is not empty but 0 instructions issued

Table 8-10. Performance monitor event selection (continued)

Number	Event	Spec/ nonspec	PR, PMM filtering[1]	Count description
Com:39[2]	Cycles branch issue stalled	Spec	yes	Branch held in decode awaiting resolution
Com:40[2]	Cycles execution stalled waiting for load data	Spec	yes	load stalls
Com:41[2]	Cycles execution stalled waiting for non-load/store SPE/EFPU result data	Spec	yes	Stalled waiting on mul, div, FP or MAC results
Load/store, data cache, and data line fill events				
Com:42	—	—	—	—
Com:43	—	—	—	—
Com:44[3]	Total translation hits	Spec	yes	—
Com:45[3]	Load translation hits	Spec	yes	Cacheable **l*** or **evl*** micro-ops translated. (includes load micro-ops from load-multiple and load-update instructions)
Com:46[3]	Store translation hits	Spec	yes	Cacheable **st*** or **evst*** micro-ops translated. (includes micro-ops from store-multiple, and store-update instructions)
Com:47[3]	Touch translation hits	Spec	yes	Cacheable **dcbt** and **dcbtst** instructions translated (L1 only) and causing linefills. (Doesn't count touches that are converted to nops i.e. exceptions, non-cacheable, HID0[NOPTI] is set, cache hits, etc.)
Com:48[3]	Data cache op translation hits	Spec	yes	**dcba**, **dcbf**, **dcbst**, and **dcbz** instructions translated
Com:49[3]	Data cache lock set instructions completed	Nonspec	yes	**dcbtls** and **dcbtstls** instructions completed
Com:50[3]	Data cache lock clear instructions completed	Nonspec	yes	**dcblc** instructions completed
Com:51[3]	Cache-inhibited load access translation hits	Spec	yes	Cache inhibited load accesses translated
Com:52[3]	Cache-inhibited store access translation hits	Spec	yes	Cache inhibited store accesses translated
Com:53[3]	Guarded load translation hits	Spec	yes	Guarded loads translated
Com:54[3]	Guarded store translation hits	Spec	yes	Guarded stores translated
Com:55[3]	Write-through store translation hits	Spec	yes	Write-through stores translated
Com:56[3]	Misaligned load or store accesses translated	Spec	yes	Misaligned load or store accesses translated. Count once per misaligned load or store.

Table 8-10. Performance monitor event selection (continued)

Number	Event	Spec/ nonspec	PR, PMM filtering[1]	Count description
Com:57[3]	DCache linefills	Spec	yes	Counts DCache reloads for any reason, including touch-type reloads. Typically used to determine approximate data cache miss rate (along with loads/stores completed).
Com:58[3]	DCache copybacks	Spec	yes	Does not count copybacks due to **dcbf**, **dcbst**, or L1FINV0 operations
Com:59[3]	DCache sequential accesses	Spec	yes	Number of sequential accesses
Com:60[3]	DCache stream hits	Spec	yes	Number of load hits due to streaming
Com:61[3]	DCache linefill buffer hits	Spec	yes	Number of load hit to the linefill buffer
Com:62[3]	Store stalls due to store to line of active linefill	Spec	yes	Stall cycles due to store to linefill in progress
Com:63[3]	Store buffer full stalls	Spec	yes	Stall cycles due to store buffer full
Com:64[2]	DCache throttling stalls	Spec	yes	Cycles the data cache asserts **p_d_halt_zlb**, which actually cause a CPU stall
Com:65[3]	DCache recycled accesses	Spec	yes	Number of loads or stores recycled for a re-lookup
Com:66[3]	DCache recycled access stalls	Spec	yes	Number of stall cycles due to recycled accesses for a re-lookup
Com:67[3]	DCache CPU aborted accesses	Spec	yes	Number of aborted requests
Com:68[3]	Data MMU miss	Spec	yes	Counts number of DTLB events
Com:69[3]	Data MMU error	Spec	yes	Counts number of DSI events
Fetch, instruction cache, instruction line fill, and instruction prefetch events				
Com:70	—	—	—	—
Com:71	—	—	—	—
Com:72[3]	ICache linefills	Spec	yes	Counts ICache reloads due to demand fetch. Used to determine instruction cache miss rate (along with instructions completed)
Com:73[3]	Number of fetches	Spec	yes	Counts fetches that write at least one instruction to the instruction buffer. (With instruction fetched (com:4), can used to compute instructions-per-fetch)
Com:74[3]	ICache lock set instructions completed	Nonspec	yes	**icbtls** instructions completed
Com:75[3]	ICache lock clear instructions completed	Nonspec	yes	**icblc** instructions completed

Table 8-10. Performance monitor event selection (continued)

Number	Event	Spec/ nonspec	PR, PMM filtering[1]	Count description
Com:76[3]	Cache-inhibited instruction access translation hits	Spec	yes	Cache-inhibited instruction accesses translated
Com:77[2]	ICache throttling stalls	Spec	yes	Cycles the instruction cache asserts **p_i_halt_zlb**, which actually causes a CPU stall
Com:78[3]	ICache recycled accesses	Spec	yes	Number of instruction access requests recycled for a re-lookup
Com:79[3]	ICache recycled access stalls	Spec	yes	Number of stall cycles due to recycled accesses for a re-lookup
Com:80[3]	ICache CPU aborted accesses	Spec	yes	Number of aborted requests
Com:81[3]	Instruction MMU miss	Spec	yes	Counts number of events
Com:82[3]	Instruction MMU error	Spec	yes	Counts number of events
BIU interface usage				
Com:83	—	—	—	—
Com:84	—	—	—	—
Com:85[3]	BIU instruction-side requests	Spec	yes	instruction-side transactions
Com:86[3]	BIU instruction-side cycles	Spec	yes	instruction-side transaction cycles
Com:87[3]	BIU data-side requests	Spec	yes	data-side transactions
Com:88[3]	BIU data-side copyback requests	Spec	yes	Replacement pushes including **dcbf**, **dcbst,** L1FINV0, copybacks.
Com:89[3]	BIU data-side cycles	Spec	yes	data-side transaction cycles
Com:90[3]	BIU single-beat write cycles	Non-Spec	yes	single beat write transaction cycles
Com:91	—	—	—	—
Snoop				
Com:92	Snoop requests	N/A	—	Externally generated snoop requests. (Counts snoop TSs.)
Com:93	Snoop hits	N/A	—	Snoop hits on all data-side resources regardless of the cache state (modified, shared, or exclusive)
Com:94[3]	Snoop induced CPU to DCache stalls	N/A	—	Cycles a pending DCache access from CPU is stalled due to contention with snoops
Com:95	Snoop Queue full cycles	N/A	—	Cycles the snoop queue is full
Com:96	—	—	—	—

Table 8-10. Performance monitor event selection (continued)

Number	Event	Spec/ nonspec	PR, PMM filtering[1]	Count description
Chaining events[4]				
Com:97	PMC0 rollover	N/A	—	$PMC0_{OV}$ transitions from 1 to 0.
Com:98	PMC1 rollover	N/A	—	$PMC1_{OV}$ transitions from 1 to 0.
Com:99	PMC2 rollover	N/A	—	$PMC2_{OV}$ transitions from 1 to 0.
Com:100	PMC3 rollover	N/A	—	$PMC3_{OV}$ transitioned from 1 to 0.
Interrupt events				
Com:101	—	—	—	—
Com:102	—	—	—	—
Com:103	Interrupts taken	Nonspec	—	—
Com:104	External input interrupts taken	Nonspec	—	—
Com:105	Critical input interrupts taken	Nonspec	—	—
Com:106	Watchdog timer interrupts taken	Nonspec	—	—
Com:107	System call and trap interrupts	Nonspec	yes	—
Com:108[2]	Cycles in which MSR_{EE}=0	Nonspec	—	—
Com:109[2]	Cycles in which MSR_{CE}=0	Nonspec	—	—
Ref:110	Transitions of TBL bit selected by $PMGC0_{TBSEL}$.	Nonspec	—	—
DEVENT events				
Com:111	DEVNT0 is generated	Nonspec	yes	assertion of **p_devnt_out0** detected
Com:112	DEVNT1 is generated	Nonspec	yes	assertion of **p_devnt_out1** detected
Com:113	DEVNT2 is generated	Nonspec	yes	assertion of **p_devnt_out2** detected
Com:114	DEVNT3 is generated	Nonspec	yes	assertion of **p_devnt_out3** detected
Com:115	DEVNT4 is generated	Nonspec	yes	assertion of **p_devnt_out4** detected
Com:116	DEVNT5 is generated	Nonspec	yes	assertion of **p_devnt_out5** detected
Com:117	DEVNT6 is generated	Nonspec	yes	assertion of **p_devnt_out6** detected
Com:118	DEVNT7 is generated	Nonspec	yes	assertion of **p_devnt_out7** detected
Watchpoint events				

Table 8-10. Performance monitor event selection (continued)

Number	Event	Spec/ nonspec	PR, PMM filtering[1]	Count description
Com:119[2]	Watchpoint #0 occurs	Nonspec	yes	assertion of **jd_watchpt0** detected
Com:120[2]	Watchpoint #1 occurs	Nonspec	yes	assertion of **jd_watchpt1** detected
Com:121[2]	Watchpoint #2 occurs	Nonspec	yes	assertion of **jd_watchpt2** detected
Com:122[2]	Watchpoint #3 occurs	Nonspec	yes	assertion of **jd_watchpt3** detected
Com:123[2]	Watchpoint #4 occurs	Nonspec	yes	assertion of **jd_watchpt4** detected
Com:124[2]	Watchpoint #5 occurs	Nonspec	yes	assertion of **jd_watchpt5** detected
Com:125[2]	Watchpoint #6 occurs	Nonspec	yes	assertion of **jd_watchpt6** detected
Com:126[2]	Watchpoint #7 occurs	Nonspec	yes	assertion of **jd_watchpt7** detected
Com:127[2]	Watchpoint #8 occurs	Nonspec	yes	assertion of **jd_watchpt8** detected
Com:128[2]	Watchpoint #9 occurs	Nonspec	yes	assertion of **jd_watchpt9** detected
Com:129	Watchpoint #10 occurs	Nonspec	yes	assertion of **jd_watchpt10** detected
Com:130	Watchpoint #11 occurs	Nonspec	yes	assertion of **jd_watchpt11** detected
Com:131	Watchpoint #12 occurs	Nonspec	yes	assertion of **jd_watchpt12** detected
Com:132	Watchpoint #13 occurs	Nonspec	yes	assertion of **jd_watchpt13** detected
Com:133[2]	Watchpoint #14 occurs	Nonspec	yes	assertion of **jd_watchpt14** detected
Com:134[2]	Watchpoint #15 occurs	Nonspec	yes	assertion of **jd_watchpt15** detected
Com:135[2]	Watchpoint #16 occurs	Nonspec	yes	assertion of **jd_watchpt16** detected
Com:136[2]	Watchpoint #17 occurs	Nonspec	yes	assertion of **jd_watchpt17** detected
Com:137[2]	Watchpoint #18 occurs	Nonspec	yes	assertion of **jd_watchpt18** detected
Com:138[2]	Watchpoint #19 occurs	Nonspec	yes	assertion of **jd_watchpt19** detected
Com:139	Watchpoint #20 occurs	Nonspec	yes	assertion of **jd_watchpt20** detected
Com:140	Watchpoint #21 occurs	Nonspec	yes	assertion of **jd_watchpt21** detected
Com:141	Watchpoint #22 occurs	Nonspec	yes	assertion of **jd_watchpt22** detected
Com:142	Watchpoint #23 occurs	Nonspec	yes	assertion of **jd_watchpt23** detected
Com:143	Watchpoint #24 occurs	Nonspec	yes	assertion of **jd_watchpt24** detected
Com:144	Watchpoint #25 occurs	Nonspec	yes	assertion of **jd_watchpt25** detected
Com:145	Watchpoint #26 occurs	Nonspec	yes	assertion of **jd_watchpt26** detected
Com:146[2]	Watchpoint #27 occurs	Nonspec	yes	assertion of **jd_watchpt27** detected
Com:147[2]	Watchpoint #28 occurs	Nonspec	yes	assertion of **jd_watchpt28** detected
Com:148[2]	Watchpoint #29 occurs	Nonspec	yes	assertion of **jd_watchpt29** detected
Com:149	—	—	—	—
Com:150	—	—	—	—

Table 8-10. Performance monitor event selection (continued)

Number	Event	Spec/ nonspec	PR, PMM filtering[1]	Count description
NEXUS events				
Com:151[3]	Cycle CPU is stalled by Nexus3 FIFO full	Nonspec	yes	OVCR stall control set to stall on FIFO fullness
Threshold events				
C0:152[3] C1:152[3]	Data cache load miss cycles	Spec	yes	Instances when the number of cycles between a load miss in the data cache and update of the data cache exceeds the threshold.
C0:153[3] C1:153[3]	Instruction cache fetch miss cycles	Spec	yes	Instances when the number of cycles between miss in the instruction cache and update of the instruction cache exceeds the threshold.
C0:154[3] C1:154[3]	External input interrupt latency cycles	N/A	—	Instances when the number of cycles between request for interrupt (*p_int_b*) asserted (but possibly masked/disabled) and redirecting fetch to external interrupt vector exceeds threshold. Once the redirection has occurred, no further threshold comparisons are made until either the interrupt request negates, or the external input interrupt is re-enabled by setting MSR_{EE}.
C0:155[3] C1:155[3]	Critical input interrupt latency cycles	N/A	—	Instances when the number of cycles between request for critical interrupt (*p_critint_b*) is asserted (but possibly masked/disabled) and redirecting fetch to the critical interrupt vector exceeds threshold. Once the redirection has occurred, no further threshold comparisons begin until either the interrupt request negates and is then re-asserted, or the critical input interrupt is re-enabled by setting MSR_{CE}.
C0:156[3] C1:156[3]	Watchdog timer interrupt latency cycles	N/A	—	Instances when the number of cycles between watchdog timer time-out request for critical interrupt becomes pending (watchdog interrupt enabled (TCR_{WIE} set) and time-out occurs ($TSR_{ENW,WIS}$ become 0b11)) and redirecting fetch to the critical interrupt vector exceeds the threshold. Once the redirection has occurred, no further threshold comparisons begin until either the watchdog interrupt request negates and is then re-asserted, or the watchdog interrupt is re-enabled by setting MSR_{CE}.

Table 8-10. Performance monitor event selection (continued)

Number	Event	Spec/ nonspec	PR, PMM filtering[1]	Count description
C0:157[3] C1:157[3]	External input interrupt pending latency cycles	N/A	—	Instances when the number of cycles between external interrupt pending (enabled and pin asserted) and redirecting fetch to the external interrupt vector exceeds the threshold. Once the redirection has occurred, no further threshold comparisons are made until either the interrupt request negates and is then re-asserted, or the external input interrupt is re-enabled by setting $MSR_{EE.}$
C0:158[3] C1:158[3]	Critical input interrupt pending latency cycles	N/A		Instances when the number of cycles between pin request for critical interrupt pending (enabled and pin asserted) and redirecting fetch to the critical interrupt vector exceeds the threshold. Once the redirection has occurred, no further threshold comparisons are made until either the interrupt request negates and is then re-asserted, or the critical input interrupt is re-enabled by setting $MSR_{CE.}$

1 The notation for the PR, and PMM filtering column either contains a 'yes' or a '-'. A 'yes' indicates that the MSR-based context filtering function is available for that event. A '-' indicates that the MSR-based context filtering is not available for that event and will have no effect on the counting of that event. See Section 8.5.1, MSR-based context filtering, for more information.

2 This event is not counted while the processor is in the waiting, halted, or stopped states, or during a debug session

3 This event is not counted while the processor is in a debug session.

4 For chaining events, if a counter is configured to count its own rollover, the result is undefined.

Chapter 9
Power Management

9.1 Power management

Power management is supported by e200z759n3 cores to minimize overall system power consumption. The e200z759n3 core provides the ability to initiate power management from external sources as well as through software techniques. The power states on the e200z759n3 core are described below.

9.1.1 Active state

The Active state is the default state for the e200z759n3 core in which all of its internal units are operating at full processor clock speed. In this state, the e200z759n3 core still provides dynamic power management in which individual internal functional units may stop clocking automatically whenever they are idle.

9.1.2 Waiting state

The e200z759n3 core enters the Waiting state as a result of executing a **wait** instruction. Following entry into the waiting state, instruction execution and bus activity is suspended. Most internal clocks are gated off in this state. The e200z759n3 core asserts **p_waiting** to indicate it is in the waiting state. Prior to entering the waiting state, all outstanding instructions and bus transactions will be completed, and the cache's store and push buffers will be flushed. The **m_clk** input should remain running while in the waiting state to allow for interrupt sampling, and to allow further transitions into the Halted or Stopped state if requested and to keep the Time Base operational if it is using **m_clk** as the clock source.

In the waiting state, the core is waiting for a valid unmasked pending interrupt request. Once a pending interrupt request is received, the core will exit the waiting state and begin interrupt processing. The return program counter value will point to the next instruction after the **wait** instruction. The interrupt can be an external input interrupt, various critical interrupts, a debug interrupt (based on ICMP), a non-maskable interrupt, or a machine check interrupt (**p_mcp_b** assertion, etc.). Once the interrupt processing begins, the core will not return to the waiting state until another **wait** instruction is executed.

The waiting state can be temporarily exited and returned to if a request is made to enter hardware debug mode (various mechanisms), the Halted state, or the Stopped state. After exiting one of these states, the processor will return to the waiting state. While temporarily exited, the **p_waiting** output will negate, and will be re-asserted once the CPU returns to the waiting state.

9.1.3 Halted state

Instruction execution and bus activity is suspended in the Halted state. Most internal clocks are gated off in this state. The e200z759n3 core asserts **p_halted** to indicate it is in the halted state. Prior to entering the halted state, all outstanding bus transactions will be completed, and the cache's store and push buffers will be flushed. The **m_clk** input should remain running while in the Halted state to ensure that snoop requests continue to be processed, to allow further transitions into the Stopped state if requested, and to keep the Time Base operational if it is using **m_clk** as the clock source.

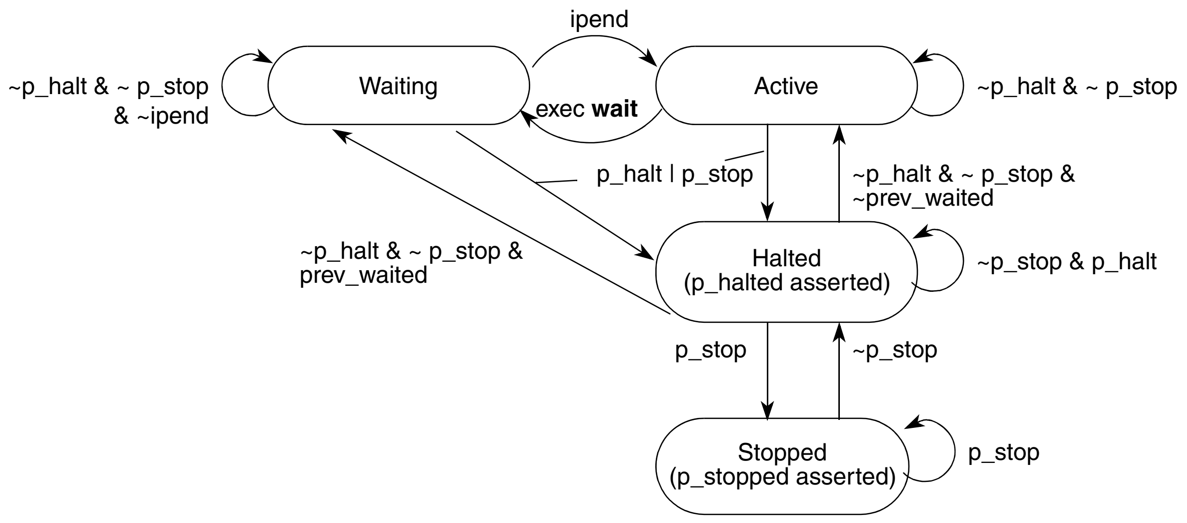
9.1.4 Stopped state

The Stopped state is characterized as having all internal functional units of the e200z759n3 core stopped except the Time Base unit and the clock control state machine logic. The internal **m_clk** may be kept running to keep the Time Base active and to allow quick recovery to the full on state. Clocks are not running to functional units in this state except for the Time Base. The Stopped state is reached after transitioning through the Halted state with the **p_stop** input asserted. The **p_stopped** output signal will be asserted once the Stopped state is reached. The CPU will not enter the Stopped state until all snoops have been processed and the snoop queue is empty. System logic is responsible for ensuring that snoop requests are no longer generated once the **p_stop** input is asserted, in order to allow a transition from the Halted to the Stopped state.

While in the Stopped state, further power savings may be achieved by disabling the Time Base by asserting **p_tbdisable**, or by stopping the **m_clk** input. This is done externally by the system after the e200z759n3 core is safely in the Stopped state and has asserted the **p_stopped** output signal. To exit from the Stopped state, the system must first restart the **m_clk** input.

Since the Time Base unit is off during the Stopped state if it is using **m_clk** as the clock source and **m_clk** is stopped, or if the Time Base clocking is disabled by the assertion of **p_tbdisable**, system software must usually have to access an external time base source after returning to the full on state in order to re-initialize the Time Base unit. In addition, it will not be possible to use a Time Base related interrupt source to exit low power states.

e200z759n3 also provides the capability of clocking the Time Base from an independent (but externally synchronized) clock source, which would allow the Time Base to be maintained during the Stopped state, and would allow a Time Base related interrupt to be generated to indicate an exit condition from the Stopped state.



ipend
~p_halt & ~ p_stop & ~ipend
Waiting
exec wait
Active
~p_halt & ~ p_stop
p_halt | p_stop
~p_halt & ~ p_stop & ~prev_waited
~p_halt & ~ p_stop & prev_waited
Halted (p_halted asserted)
~p_stop & p_halt
p_stop
~p_stop
Stopped (p_stopped asserted)
p_stop

Figure 9-1. Power management state diagram

9.1.5 Power management pins

p_waiting - output pin asserted when the e200z759n3 core is in the Waiting state.

p_halt - input pin is asserted by system logic to request the core to go into the Halted state. Negating this pin causes the e200z759n3 core to transition back into the Active or Waiting state if **p_stop** is also negated.

p_halted - output pin asserted when the e200z759n3 core is in the Halted state.

p_stop - input pin is asserted by system logic to request that the e200z759n3 core go into the Stopped state. Negating this pin causes the e200z759n3 core to transition back into the Halted state from the Stopped state.

p_stopped - output pin asserted when the e200z759n3 core is in the Stopped state.

p_tbdisable - input pin is asserted by system logic when clocking of the Time Base should be disabled.

p_tbint - output pin is asserted when an internal Time Base interrupt request is signaled.

p_doze, **p_nap**, and **p_sleep** output pins that reflects the state of $HID0_{DOZE}$, $HID0_{NAP}$, and $HID0_{SLEEP}$ respectively. These pins are qualified with MSR_{WE} = 1. Interpretation of these signals is done by the system logic.

p_wakeup - output pin asserted when an interrupt is pending or other condition that requires the clock to be running.

9.1.6 Power management control bits

The following bits are used by software to generate a request to enter a power-saving state and to choose the state to be entered:

- MSR_{WE}—The WE bit is used to qualify assertion of the **p_doze**, **p_nap**, and **p_sleep** output pins to the system logic. When MSR_{WE} is negated, these pins are negated. When MSR_{WE} is set, these pins reflect the state of their respective control bits in the HID0 register.
- $HID0_{DOZE}$ —The interpretation of the doze mode bit is done by the external system logic. Doze mode on the e200z759n3 core is intended to be the halted state with the clocks running.
- $HID0_{NAP}$ —The interpretation of the nap mode bit is done by the external system logic. Nap mode on the e200z759n3 core may be used for a powerdown state with the Time Base enabled.
- $HID0_{SLEEP}$ —The interpretation of the sleep mode bit is done by the external system logic. Sleep mode on the e200z759n3 core may be used for a powerdown state with the Time Base disabled.

9.1.7 Software considerations for power management using wait instructions

Executing a **wait** instruction causes the e200z759n3 core to complete instruction fetch and execution activity and await an interrupt. The **p_waiting** output is asserted once the Waiting state is entered. External system hardware may interpret the state of this signal and activate the **p_halt** and/or **p_stop** inputs to cause the e200z759n3 core to enter a quiescent state in which clocks may be disabled for low power operation. Alternatively, system hardware may utilize some other clock control mechanism while the processor is in the Waiting state, and **p_wakeup** remains negated.

9.1.8 Software considerations for power management using Doze, Nap or Sleep

Setting MSR[WE] generates a request to enter a power saving state. The power saving state (doze, nap, or sleep) must be previously determined by setting the appropriate HID0 bit. Setting MSR[WE] has no direct effect on instruction execution, but it simply reflected on **p_doze**, **p_nap**, and **p_sleep** depending on the setting of $\text{HID0}_{\text{DOZE}}$, HID0_{NAP}, and $\text{HID0}_{\text{SLEEP}}$ respectively. Note that the e200z759n3 core is not affected by assertion of these pins directly. External system hardware may interpret the state of these signals and activate the **p_halt** and/or **p_stop** inputs to cause the e200z759n3 core to enter a quiescent state in which clocks may be disabled for low power operation.

To ensure a clean transition into and out of a power saving mode, the following program sequence is recommended:

```
                  sync
                  mtmsr (WE)
                  isync
        loop:     br loop   (optionally use a wait instruction)
```

An interrupt is typically used to exit a power saving state. The **p_wakeup** output is used to indicate to the system logic that an interrupt (or a debug request) has become pending. System logic uses this output to re-enable the clocks and exit a low power state. The interrupt handler is responsible for determining how to exit the low power loop if one is used. Wait instructions will be exited automatically. The vectored interrupt capability provided by the core may be useful in assisting the determination if an external hardware interrupt is used to perform the wake-up.

9.1.9 Debug considerations for power management

When a debug request is presented to the e200z759n3 core while in either the Waiting, Halted or Stopped state, the p_wakeup signal will be asserted, and when m_clk is provided to the CPU, it will temporarily exit the Waiting, Halted or Stopped state and will enter Debug mode regardless of the assertion of p_halt or p_stop. The p_waiting, p_halted, and p_stopped outputs will be negated for the duration of the time the CPU remains in a debug session (jd_debug_b asserted). When the debug session is exited, the CPU will re-sample the p_halt and p_stop inputs and will re-enter the Halted or Stopped state as appropriate. If the CPU was previously waiting, and no interrupt was received while in the debug session, it will re-enter the Waiting state and re-assert p_waiting.

Chapter 10
Memory Management Unit

10.1 Overview

The e200z759n3 Memory Management Unit is a 32-bit *PowerISA 2.06* compliant implementation, with the following feature set:

- *Freescale EIS* MMU architecture compliant
- Translates from 32-bit effective to 32-bit real addresses
- 32-entry fully associative TLB with support for twenty-three page sizes (1 KB, 2 KB, 4 KB, 8 KB, 16 KB, 32 KB, 64 KB, 128 KB, 256 KB, 512 KB, 1 MB, 2 MB, 4 MB, 8 MB, 16 MB, 32 MB, 64 MB, 128 MB, 256 MB, 512 MB, 1 GB, 2 GB, 4 GB)
- Hardware assist for TLB miss exceptions
- Software managed by **tlbre**, **tlbwe**, **tlbsx**, **tlbsync**, and **tlbivax** instructions
- Support for external control of entry matching for a subset of TID values to support non-intrusive runtime mapping modifications

10.2 Effective to real address translation

10.2.1 Effective addresses

Instruction accesses are generated by sequential instruction fetches or due to a change in program flow (branches and interrupts). Data accesses are generated by load, store, and cache management instructions. The e200z759n3 instruction fetch, branch, and load/store units generate 32-bit effective addresses. The MMU translates this effective address to a 32-bit real address, which is then used for memory accesses.

The *PowerISA 2.06* 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 e200z759n3 MMU supports twenty-three page sizes (1 KB, 2 KB, 4 KB, 8 KB, 16 KB, 32 KB, 64 KB, 128 KB, 256 KB, 512 KB, 1 MB, 2 MB, 4 MB, 8 MB, 16 MB, 32 MB, 64 MB, 128 MB, 256 MB, 512 M, 1 GB, 2 GB, 4 GB). In order for an effective to real address translation to exist, a valid entry for the page containing the effective address must be in a Translation Lookaside Buffer (TLB). Addresses for which no TLB entry exists (a TLB miss) cause Instruction or Data TLB Errors.

10.2.2 Address spaces

Instruction accesses are generated by sequential instruction fetches or due to a change in program flow (branches and interrupts). Data accesses are generated by load, store, and cache management instructions.

The *PowerISA 2.06* architecture defines two effective address spaces for instruction accesses and two effective address spaces for data accesses. The current effective address space for instruction or data accesses is determined by the value of MSR[IS] and MSR[DS], respectively. The address space indicator (the value of either MSR[IS] or MSR[DS], as appropriate) is used in addition to the effective address generated by the processor for translation into a physical address by the TLB mechanism. Because

MSR[IS] and MSR[DS] are both cleared to ‘0’ when an interrupt occurs, an address space value of 0b0 can be used to denote interrupt-related address spaces (or possibly all system software address spaces), and an address space value of 0b1 can be used to denote non interrupt-related (or possibly all user address spaces) address spaces.

The address space associated with an instruction or data access is included as part of the virtual address in the translation process (AS). The **p_tc[1]** interface signal indicates the appropriate address space.

10.2.3 Process ID

The *PowerISA 2.06* architecture defines that a process ID (PID) value is associated with each effective address (instruction or data) generated by the processor. At the Book E level, a single PID register is defined as a 32-bit register, and it maintains the value of the PID for the current process. This PID value is included as part of the virtual address in the translation process (PID0). For the e200z759n3 MMU, the PID is 8 bits in length. The most-significant 24 bits are unimplemented and read as ‘0’. The **p_pid0[0:7]** interface signals indicate the current process ID.

10.2.4 Translation flow

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

Figure 10-1 shows the translation match logic for the effective address plus its attributes, collectively called the virtual address, and how it is compared with the corresponding fields in the TLB entries.

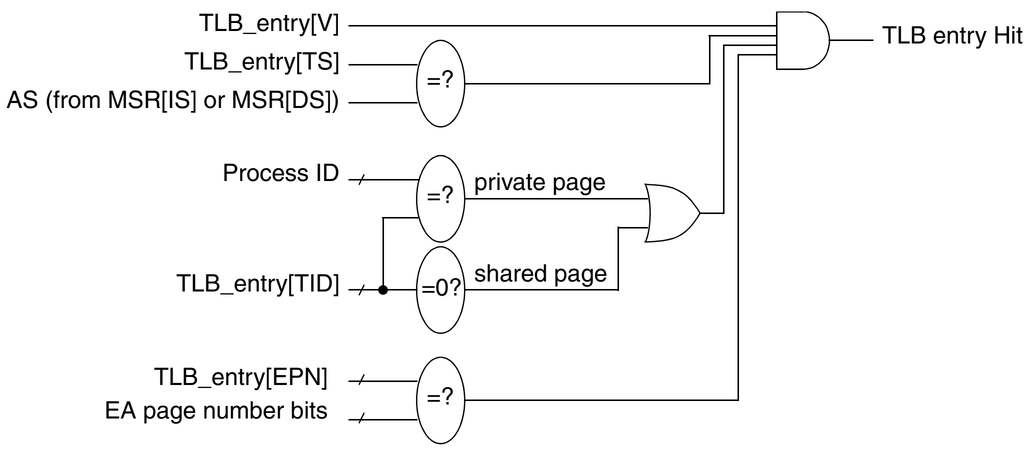


Figure 10-1. Virtual address and TLB entry compare process

The page size defined for a TLB entry determines how many bits of the effective address are compared with the corresponding EPN field in the TLB entry as shown in Table 10-1. On a TLB hit, the corresponding bits of the Real Page Number (RPN) field are used to form the real address.

Table 10-1. Page size field encodings and EPN field comparison

SIZE field	Page size (2^{SIZE}KB)	EA to EPN comparison
0b00000	1 KB	EA[0:21] =? EPN[0:21]
0b00001	2 KB	EA[0:20] =? EPN[0:20]
0b00010	4 KB	EA[0:19] =? EPN[0:19]
0b00011	8 KB	EA[0:18] =? EPN[0:18]
0b00100	16 KB	EA[0:17] =? EPN[0:17]
0b00101	32 KB	EA[0:16] =? EPN[0:16]
0b00110	64 KB	EA[0:15] =? EPN[0:15]
0b00111	128 KB	EA[0:14] =? EPN[0:14]
0b01000	256 KB	EA[0:13] =? EPN[0:13]
0b01001	512 KB	EA[0:12] =? EPN[0:12]
0b01010	1 MB	EA[0:11] =? EPN[0:11]
0b01011	2 MB	EA[0:10] =? EPN[0:10]
0b01100	4 MB	EA[0:9] =? EPN[0:9]
0b01101	8 MB	EA[0:8] =? EPN[0:8]
0b01110	16 MB	EA[0:7] =? EPN[0:7]
0b01111	32 MB	EA[0:6] =? EPN[0:6]
0b10000	64 MB	EA[0:5] =? EPN[0:5]
0b10001	128 MB	EA[0:4] =? EPN[0:4]
0b10010	256 MB	EA[0:3] =? EPN[0:3]
0b10011	512 MB	EA[0:2] =? EPN[0:2]
0b10100	1 GB	EA[0:1] =? EPN[0:1]
0b10101	2 GB	EA[0] =? EPN[0]
0b10110	4 GB	(none)

On a TLB hit, the generation of the physical address occurs as shown in Figure 10-2.

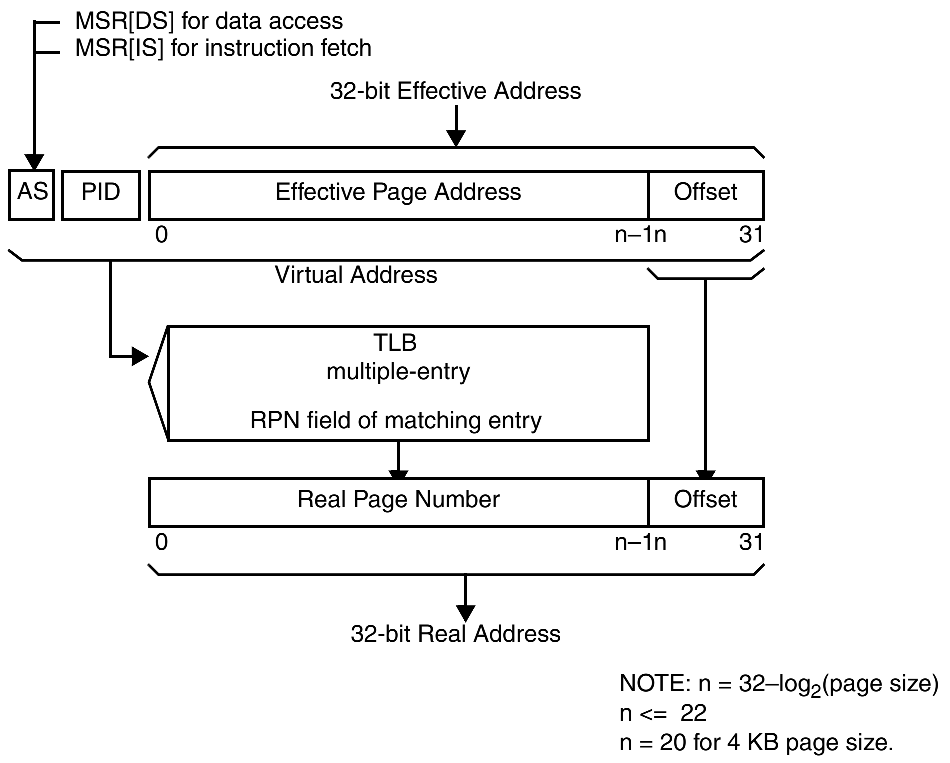


Figure 10-2. Effective to real address translation flow

10.2.5 Permissions

An operating system may restrict access to virtual pages by selectively granting permissions for user mode read, write, and execute, and supervisor mode read, write, and execute on a per page basis. These permissions can be set up for a particular system (for example, program code might be execute-only, data structures may be mapped as read/write/no-execute) and can also be changed by the operating system based on application requests and operating system policies.

The UX, SX, UW, SW, UR, and SR access control bits are provided to support selective permissions (access control):

- SR—Supervisor read permission. Allows loads and load-type cache management instructions to access the page while in supervisor mode (MSR[PR=0]).
- SW—Supervisor write permission. Allows stores and store-type cache management instructions to access the page while in supervisor mode (MSR[PR=0]).
- SX—Supervisor execute permission. Allows instruction fetches to access the page and instructions to be executed from the page while in supervisor mode (MSR[PR=0]).
- UR—User read permission. Allows loads and load-type cache management instructions to access the page while in user mode (MSR[PR=1]).
- UW—User write permission. Allows stores and store-type cache management instructions to access the page while in user mode (MSR[PR=1]).

- UX—User execute permission. Allows instruction fetches to access the page and instructions to be executed from the page while in user mode (MSR[PR=1]).

If the translation match was successful, the permission bits are checked as shown in Figure 10-3. If the access is not allowed by the access permission mechanism, the processor generates an Instruction or Data Storage interrupt (ISI or DSI). The current privilege level of an access is signaled to the MMU with the CPU's **p_tc[0]** output signal.

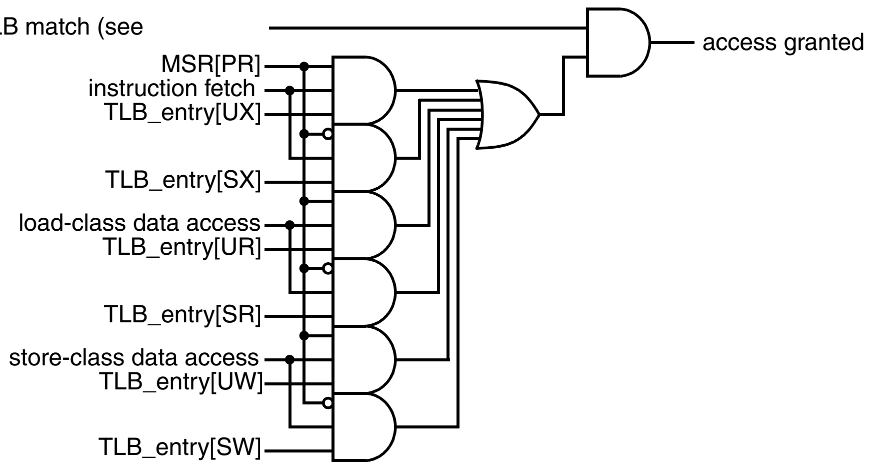


Figure 10-3. Granting of access permission

10.2.6 Restrictions on 1 KB and 2 KB page size usage

Because of certain implementation limitations regarding coherency lookup operations (lookup is done by physical address), if 1 KB or 2 KB pages are used, the low order virtual address bits used to index the cache (A[20:21] for 1 KB pages, A20 for 2 KB pages) must match the corresponding physical address bit value(s). For example, if logical page X maps to physical page P, then X and P must have the same values of A[20:21] for 1 KB pages, and A20 for 2 KB pages. This restriction must be followed for proper CPU operation.

10.3 Translation Lookaside Buffer (TLB)

The *Freescale EIS* architecture defines support for zero or more TLBs in an implementation, each with its own characteristics, and provides configuration information for software to query the existence and structure of the TLB(s) through a set of special purpose registers: MMUCFG, TLB0CFG, TLB1CFG, etc. By convention, TLB0 is used for a set associative TLB with fixed page sizes, TLB1 is used for a fully associative TLB with variable page sizes, and TLB2 is arbitrarily defined by an implementation. The e200z759n3 MMU supports a TLB that is fully associative and supports variable page sizes, thus it corresponds to TLB1.

TLB1 consists of a 32-entry, fully associative CAM array with support for twenty-three page sizes. To perform a lookup, the CAM is searched in parallel for a matching TLB entry. The contents of this TLB entry are then concatenated with the page offset of the original effective address. The result constitutes the real (physical) address of the access.

A hit to multiple TLB entries is considered to be a programming error. If this occurs, the TLB generates an invalid address but an exception will not be reported.

Table 10-2. TLB entry bit definitions

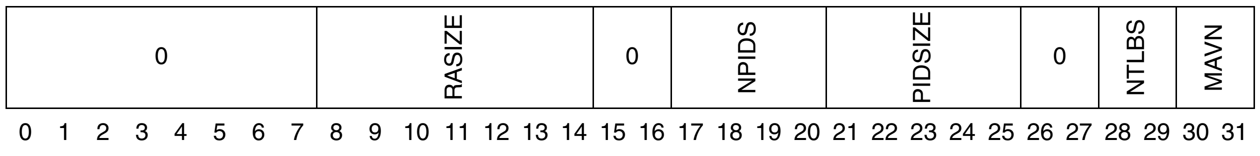
Field	Comments
V	Valid bit for entry
TS	Translation address space (compared against AS bit)
TID[0:7]	Translation ID (compared against PID0 or ‘0’)
EPN[0:21]	Effective page number (compared against effective address)
RPN[0:21]	Real page number (translated address)
SIZE[0:4]	Page size (see Table 10-1)
SX, SW, SR	Supervisor execute, write, and read permission bits
UX, UW, UR	User execute, write, and read permission bits
WIMGE	Translation attributes (write-through required, cache-inhibited, memory coherence required, guarded, endian)
U0-U3	User bits — used only by software
IPROT	Invalidation protect
VLE	VLE page indicator

10.4 Configuration information

Information about the configuration for a given MMU implementation is available to system software by reading the contents of the MMU configuration SPRs. These SPRs describe the architectural version of the MMU, the number of TLB arrays, and the characteristics of each TLB array.

10.4.1 MMU Configuration Register (MMUCFG)

The MMU Configuration Register (MMUCFG) is a 32-bit read-only register. The SPR number for MMUCFG is 1015 in decimal. MMUCFG provides information about the configuration of the e200z759n3 MMU design. The MMUCFG register is shown in Figure 10-4.



SPR - 1015; Read-Only

Figure 10-4. MMU Configuration Register (MMUCFG)

The MMUCFG bits are described in Table 10-3.

Table 10-3. MMUCFG field descriptions

Bits	Name	Function
0:7 [32:39]	—	Reserved[1]
8:14 [40:46]	RASIZE	Number of Bits of Real Address supported 0100000- This version of the MMU implements 32 real address bits
15:16 [47:48]	—	Reserved[1]
17:20 [49:52]	NPIDS	Number of PID Registers 0001 This version of the MMU implements one PID register (PID0)
21:25 [53:57]	PIDSIZE	PID Register Size 00111 PID registers contain 8 bits in this version of the MMU
26:27 [58:59]	—	Reserved[1]
28:29 [60:61]	NTLBS	Number of TLBs 01 This version of the MMU implements two TLB structures: a null TLB0 and a fully-associative TLB for TLB1
30:31 [62:63]	MAVN	MMU Architecture Version Number 00 This version of the MMU implements Version 1.0 of the *Freescale EIS* MMU Architecture

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

10.4.2 TLB0 Configuration Register (TLB0CFG)

The TLB0 Configuration Register (TLB0CFG) is a 32-bit read-only register. The SPR number for TLB0CFG is 688 in decimal. TLB0CFG provides information about the configuration of TLB0. Since the e200z759n3 MMU design does not implement TLB0, this register reads as all ‘0’. It is supplied to allow software to query it in a fashion compatible with other *Freescale EIS* designs. The TLB0CFG register is shown in Figure 10-5.

ASSOC	MINSIZE	MAXSIZE	IPROT	AVAIL	P2PSA	0	NENTRY
0 1 2 3 4 5 6 7	8 9 10 11	12 13 14 15	16	17	18	19	20 21 22 23 24 25 26 27 28 29 30 31

SPR - 688; Read-Only

Figure 10-5. TLB0 Configuration Register (TLB0CFG)

The TLB0CFG bits are described in Table 10-4.

Table 10-4. TLB0CFG field descriptions

Bits	Name	Function
0:7 [32:39]	ASSOC	Associativity 0
8:11 [40:43]	MINSIZE	Minimum Page Size 0
12:15 [44:47]	MAXSIZE	Maximum Page Size 0
16 [48]	IPROT	Invalidate Protect Capability 0 Not present in TLB0
17 [49]	AVAIL	Page Size Availability 0 No variable page sizes available
18 [50]	P2PSA	Power-of-2 Page Size Availability 0 No odd powers of 2 page sizes are supported
19 [51]	—	Reserved[1]
20:31 [52:63]	NENTRY	Number of Entries 0 TLB0 contains 0 entries

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

10.4.3 TLB1 Configuration Register (TLB1CFG)

The TLB1 Configuration Register (TLB1CFG) is a 32-bit read-only register. The SPR number for TLB1CFG is 689 in decimal. TLB1CFG provides information about the configuration of TLB1 in the e200z759n3 MMU. The TLB1CFG register is shown in Figure 10-6.

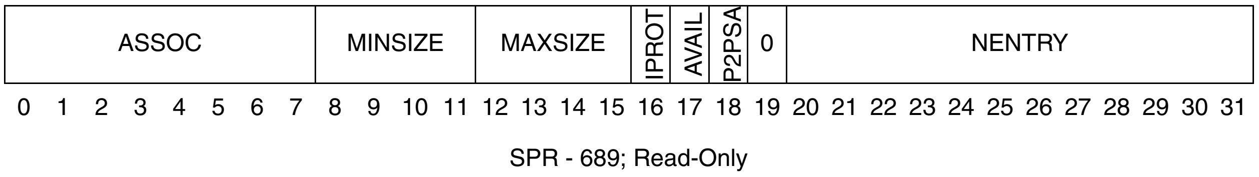


Figure 10-6. TLB1 Configuration Register (TLB1CFG)

The TLB1CFG bits are described in Table 10-5.

Table 10-5. TLB1CFG field descriptions

Bits	Name	Function
0:7 [32:39]	ASSOC	Associativity 0x20 Indicates that TLB1 associativity is 32

Table 10-5. TLB1CFG field descriptions (continued)

Bits	Name	Function
8:11 [40:43]	MINSIZE	Minimum Page Size 0x0Smallest page size is 1 KB
12:15 [44:47]	MAXSIZE	Maximum Page Size 0xB Largest page size is 4 GB
16 [48]	IPROT	Invalidate Protect Capability 1 Invalidate Protect Capability is supported in TLB1
17 [49]	AVAIL	Page Size Availability 1 All page sizes between MINSIZE and MAXSIZE are supported
18 [50]	P2PSA	Power-of-2 Page Size Availability 1 All odd powers of 2 page sizes between MINSIZE and MAXSIZE are supported (2 KB, 8 KB, 32 KB, etc.)
19 [51]	—	Reserved[1]
20:31 [52:63]	NENTRY	Number of Entries 0x20 Indicates that TLB1 contains 32 entries

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

10.5 Software interface and TLB instructions

The TLB is accessed indirectly through several MMU Assist (MAS) registers. Software can write and read the MMU Assist registers with **mtspr** and **mfspr** instructions. These registers contain information related to reading and writing a given entry within the TLB. Data is read from the TLB into the MAS registers with a **tlbre** (TLB read entry) instruction. Data is written to the TLB from the MAS registers with a **tlbwe** (TLB write entry) instruction.

Certain fields of the MAS registers are also written by hardware when an Instruction TLB Error or Data TLB Error interrupt occurs.

On a TLB Error interrupt, the MAS registers will be written by hardware with the proper EA, default attributes (TID, WIMGE, permissions, etc.), and TLB selection information, and an entry in the TLB to replace. Software manages this entry selection information by updating a replacement entry value during TLB miss handling. Software must provide the correct RPN and permission information in one of the MAS registers before executing a **tlbwe** instruction.

On taking a DSI or ISI interrupt, software should update the search PID (SPID) and search address space (SAS) fields in the MAS registers using PID0, and appropriate MSR[IS] or MSR[DS] values that were used when the DSI or ISI exception was recognized. During the interrupt handler, software can issue a TLB search instruction (**tlbsx**), which uses the SPID field along with the SAS field, to determine the entry related to the DSI or ISI exception. (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.

10.5.1 TLB read entry instruction (tlbre)

The TLB read entry instruction causes the content of a single TLB entry to be placed in the MMU assist registers. The entry is specified by the TLBSEL and ESEL fields of the MAS0 register. The entry contents are placed in the MAS1, MAS2, and MAS3 registers. See Table 10-15 for details on how MAS register fields are updated.

tlbre **tlbre**

tlb read entry

0–5	6–20	21–30	31
31	0	1 1 1 0 1 1 0 0 1 0	0

```
tlb_entry_id = MAS0(TLBSEL, ESEL)
result = MMU(tlb_entry_id)
MAS1, MAS2, MAS3 = result
```

10.5.2 TLB write entry instruction (tlbwe)

The TLB write entry instruction causes the contents of certain fields within the MMU assist registers MAS1, MAS2, and MAS3 to be written into a single TLB entry in the MMU. The entry written is specified by the TLBSEL, and ESEL fields of the MAS0 register.

tlbwe **tlbwe**

tlb write entry

0–5	6–20	21–30	31
31	0	1 1 1 1 0 1 0 0 1 0	0

```
tlb_entry_id = MAS0(TLBSEL, ESEL)
MMU(tlb_entry_id) = MAS1, MAS2, MAS3
```

10.5.3 TLB search instruction (tlbsx)

The TLB search instruction updates the MMU assist registers conditionally based on success or failure of a lookup of the TLB. The lookup is controlled by an effective address provided by GPR[RB] as specified in the instruction encoding, as well as by the SAS and SPID search fields in MAS6. The values placed into

MAS0, MAS1, MAS2, and MAS3 differ depending on a successful or unsuccessful search. See Table 10-15 for details on how MAS register fields are updated.

tlbsx tlbsx

TLB Search Indexed

tlbsx RA,RB Form X

31	0	RA	RB	1 1 1 0 0 1 0 0 1 0	0
0 5	6 10	11 15	16 20	21 30	31

```
if RA!=0 then EA = GPR(RA) + GPR(RB)
else EA = GPR(RB)
ProcessIDs = MAS6(SPID), 8’b00000000
AS = MAS6(SAS)
VA = AS || ProcessIDs || EA
if Valid_TLB_matching_entry_exists(VA)
then result = see Table 10-15, column labelled “tlbsx hit”
else result = see Table 10-15, column labelled “tlbsx miss”
MAS0, MAS1, MAS2, MAS3 = result
```

10.5.4 TLB Invalidate (tlbivax) Instruction

The TLB invalidate operation is performed whenever a TLB Invalidate Virtual Address Indexed (**tlbivax**) instruction is executed. This instruction invalidates TLB entries that correspond to the virtual address calculated by this instruction. The address is detailed in Table 10-6. No other information except for that shown in Table 10-6 is used for the invalidation (entry AS and TID values are don’t-cared).

Additional information about the targeted TLB entries is encoded in two of the lower bits of the effective address calculated by the **tlbivax** instruction. Bit 28 of the **tlbivax** effective address is the TLBSEL field. This bit should be set to ‘1’ to ensure TLB1 is targeted by the invalidate. Bit 29 of the **tlbivax** effective address is the INV_ALL field. If this bit is set, it indicates that the invalidate operation needs to completely invalidate all entries of TLB1 that are not marked as invalidation protected (IPROT bit of entry set to ‘1’).

The bits of EA used to perform the **tlbivax** invalidation of TLB1 are bits 0:21.

Table 10-6. tlbivax EA bit definitions

Bits	Description
0:21	EA[0:21]
22:27	Reserved[1]
28	TLBSEL(1=TLB1) Should be set to ‘1’ for future compatibility.

Table 10-6. tlbivax EA bit definitions

Bits	Description
29	INV_ALL
30:31	Reserved[1]

[1] These bits should be zero for future compatibility. They are ignored.

tlbivax

TLB Invalidate Virtual Address Indexed

tlbivax RA,RB Form X

31	0	RA	RB	1 1 0 0 0 1 0 0 1 0	0
0 5	6 10	11 15	16 20	21 30	31

```
if RA!=0 then EA = GPR(RA) + GPR(RB)
else EA = GPR(RB)
VA = EA
if (Valid_TLB_matching_entry_exists(VA) or INV_ALL) and Entry_IPROT_not_set
then Invalidate entry
```

10.5.5 TLB synchronize instruction (tlbsync)

The TLB synchronize instruction is treated as a privileged no-op by the e200z759n3.

tlbsync

TLB Synchronize

tlbsync

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

10.6 TLB operations

10.6.1 Translation reload

The TLB reload function is performed in software with some hardware assist. This hardware assist 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.

10.6.2 Reading the TLB

The TLB array can be read by first writing the necessary information into MAS0 using **mtspr** and then executing the **tlbre** instruction. To read an entry from the TLB, the TLBSEL field in MAS0 must be set to ‘01’, and the ESEL bits in MAS0 must be set to point to the desired entry. After executing the **tlbre** instruction, MAS1-MAS3 will be updated with the data from the selected TLB entry.

10.6.3 Writing the TLB

The TLB1 array can be written by first writing the necessary information into MAS0-MAS3 using **mtspr** and then executing the **tlbwe** instruction. To write an entry into the TLB, the TLBSEL field in MAS0 must be set to ‘01’, and the ESEL bits in MAS0 must be set to point to the desired entry. When the **tlbwe** instruction is executed, the TLB entry information stored in MAS1-MAS3 will be written into the selected TLB entry.

10.6.4 Searching the TLB

The TLB can be searched using the **tlbsx** instruction by first writing the necessary information into MAS6. The **tlbsx** instruction will search using EPN[0:21] from the GPR selected by the instruction, SAS (search AS bit) in MAS6, and SPID in MAS6. If the search is successful, the given TLB entry information will be 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 will be set; if unsuccessful it is cleared. The **tlbsx** instruction is useful for finding the TLB entry that caused a DSI or ISI exception.

10.6.5 TLB miss exception update

When a TLB miss exception occurs, MAS0-MAS3 are updated with the defaults specified in MAS4, and the AS and EPN[0:21] of the access that caused the exception. In addition, the ESEL bits are updated with the replacement entry value.

This sets up all the TLB entry data necessary for a TLB write except for the RPN[0:21], the U0-U3 user bits, and the UX/SX/UW/SW/UR/SR permission bits, all of which are stored in MAS3. Thus, if the defaults stored in MAS4 are applicable to the TLB entry to be loaded, the TLB miss exception handler will only have to update MAS3 via **mtspr** before executing **tlbwe**. If the defaults are not applicable to the TLB entry being loaded, then the TLB miss exception handler will have to update MAS0-MAS2 before performing the TLB write.

10.6.6 IPROT invalidation protection

The IPROT bit is used to protect TLB entries from invalidation. TLB entries with IPROT set are not invalidated by a **tlbivax** instruction (even when INV_ALL is indicated), nor by the MMUCSR0[TLB1_FI] control function. The IPROT bit is used to protect interrupt vectors/handlers, since the instruction fetch of those vectors must be guaranteed to never take a TLB miss exception.

10.6.7 TLB load on reset

During reset, all TLB entries except entry 0 are invalidated. TLB entry 0 is loaded with the values in the following table:

Table 10-7. TLB entry 0 values after reset

Field	Reset value	Comments
VALID	1	Entry is valid
TS	0	Address space 0
TID[0:7]	0x00	TID value for shared (global) page
EPN[0:21]	value of **p_rstbase[0:21]**	Page address present on **p_rstbase[0:29]**. See Section 14.2.2.5, Reset base (p_rstbase[0:29])
RPN[0:21]	value of **p_rstbase[0:21]**	Page address present on **p_rstbase[0:29]**. See Section 14.2.2.5, Reset base (p_rstbase[0:29])
SIZE[0:4]	00010	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 Section 14.2.2.6, Reset endian mode (p_rst_endmode)
U0-U3	0000	User bits

Table 10-7. TLB entry 0 values after reset

Field	Reset value	Comments
IPROT	1	Page is protected from invalidation
VLE	the value of **p_rst_vlemode**	Value present on **p_rst_vlemode signal.** See Section 14.2.2.7, Reset VLE Mode (p_rst_vlemode).

10.6.8 The G bit

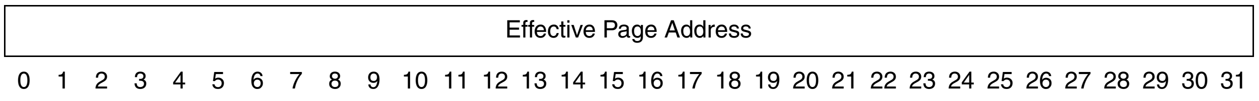
The G bit provides protection from bus accesses that could be cancelled due to an exception on a prior uncompleted instruction.

If G=1 (guarded), these types of accesses must stall (if they miss in the cache) until the exception status of the instruction(s) in progress is known. If G=0 (unguarded), then these accesses may be issued to the bus regardless of the completion status of other instructions. Since the e200z759n3 does not make requests to the bus for load or store instructions that miss in the cache until it is known that prior instructions will complete without exceptions, proper operation will always occur to guarded storage.

10.7 MMU control registers

10.7.1 Data Exception Address Register (DEAR)

The Data Exception Address register is loaded with the effective address of the data access that results in an Alignment, Data TLB Miss, or DSI exception.



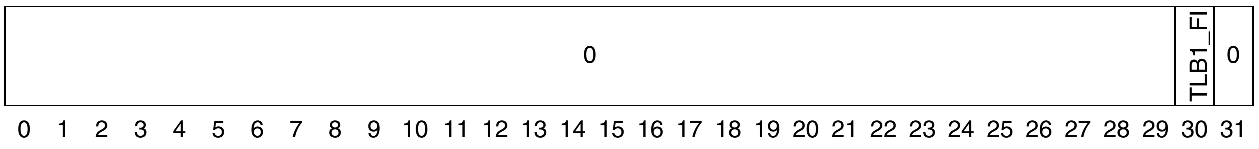
SPR - 61; Read/ Write; Reset - Unaffected

Figure 10-7. Data Exception Address Register (DEAR)

The DEAR register can be read or written using the **mfspr** and **mtspr** instructions.

10.7.2 MMU Control and Status Register 0 (MMUCSR0)

The MMU Control and Status Register 0 (MMUCSR0) is a 32-bit register. The SPR number for MMUCSR0 is 1012 in decimal. MMUCSR0 controls the state of the MMU. The MMUCSR0 register is shown in Figure 10-8.



SPR - 1012; Read/ Write; Reset - 0x0

Figure 10-8. MMU Control and Status Register 0 (MMUCSR0)

The MMUCSR0 bits are described in Table 10-8.

Table 10-8. MMUCSR0 field descriptions

Bits	Name	Description
0:29 [32:61]	—	Reserved[1]
30 [62]	TLB1_FI	TLB1 flash invalidate 0 No flash invalidate 1 TLB1 invalidation operation When written to a ‘1’, a TLB1 invalidation operation is initiated by hardware. Once complete, this bit is reset to ‘0’. Writing a ‘1’ while an invalidation operation is in progress will result in an undefined operation. Writing a ‘0’ to this bit while an invalidation operation is in progress will be ignored. TLB1 invalidation operations require 3 cycles to complete.
31 [63]	—	Reserved[1]

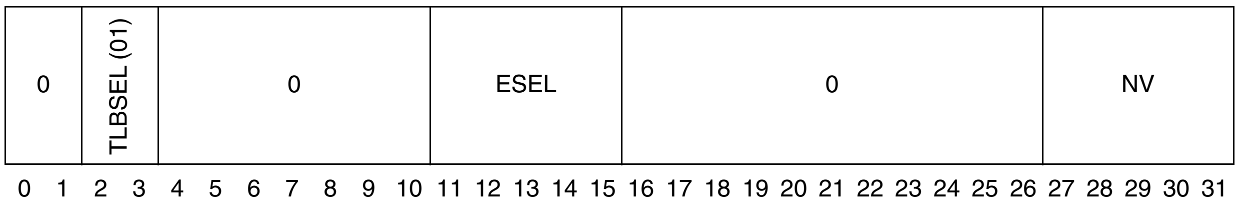
[1] These bits are not implemented, will be read as zero, and writes are ignored.

10.7.3 MMU assist registers (MAS)

The e200z759n3 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 e200z759n3 does not implement the MAS5 register, present in other Freescale Book E designs, because the **tlbsx** instruction only searches based on a single SPID value.

10.7.3.1 MMU Read/Write and Replacement Control register (MAS0)

The MAS0 register is shown in Figure 10-9. Fields are defined in Table 10-9.



SPR - 624; Read/ Write; Reset - Unaffected

Figure 10-9. MMU Assist Register 0 (MAS0)

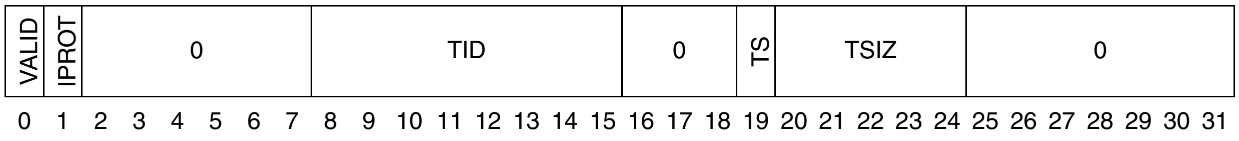
Table 10-9. MAS0 field descriptions

Bit	Name	Description
0:1 [32:33]	—	Reserved[1]
2:3 [34:35]	TLBSEL	Selects TLB for access: 00=TLB0, 01=TLB1 (ignored by Zen, should be written to 01 for future compatibility)
4:10 [36:42]	—	Reserved[1]
11:15 [43:47]	ESEL	Entry select for TLB.
16:25 [48:57]	—	Reserved[1]
27:31 [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 10-15)

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

10.7.3.2 Descriptor Context and Configuration Control register (MAS1)

The MAS1 register is shown in Figure 10-10. Fields are defined in Table 10-10.



SPR - 625; Read/ Write; Reset - Unaffected

Figure 10-10. MMU Assist Register 1 (MAS1)

Table 10-10. MAS1 field descriptions

Bit	Name	Description
0 [32]	VALID	TLB Entry Valid 0 This TLB entry is invalid 1 This TLB entry is valid
1 [33]	IPROT	Invalidation Protect 0 Entry is not protected from invalidation 1 Entry is protected from invalidation as described in Section 10.6.6, IPROT invalidation protection. Protects TLB entry from invalidation by **tlbivax** (TLB1 only), or flash invalidates through MMUSCR0[TLB1_FI].
2:7 [34:39]	—	Reserved[1]

Table 10-10. MAS1 field descriptions (continued)

Bit	Name	Description
8:15 [40:47]	TID	Translation ID bits This field is compared with the current process IDs of the effective address to be translated. A TID value of 0 defines an entry as global and matches with all process IDs.
16:18 [48:50]	—	Reserved[1]
19 [51]	TS	Translation address space This bit is compared with the IS or DS fields of the MSR (depending on the type of access) to determine if this TLB entry may be used for translation.
20:24 [52:56]	TSIZE	Entry's page size Supported page sizes are: 0b00000 — 1 KB 0b00001 — 2 KB 0b00010 — 4 KB 0b00011 — 8 KB 0b00100 — 16 KB 0b00101 — 32 KB 0b00110 — 64 KB 0b00111 — 128 KB 0b01000 — 256 KB 0b01001 — 512 KB 0b01010 — 1 MB 0b01011 — 2 MB 0b01100 — 4 MB 0b01101 — 8 MB 0b01110 — 16 MB 0b01111 — 32 MB 0b10000 — 64 MB 0b10001 — 128 MB 0b10010 — 256 MB 0b10011 — 512 MB 0b10100 — 1 GB 0b10101 — 2 GB 0b10110 — 4 GB All other values are undefined
25:31 [57:63]	—	Reserved[1]

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

10.7.3.3 EPN and Page Attributes register (MAS2)

The MAS2 register is shown in Figure 10-11. Fields are defined in Table 10-11.

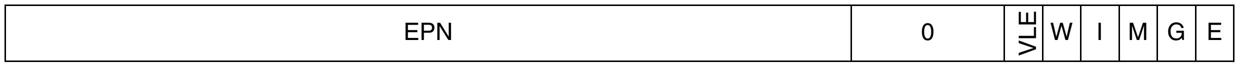


Figure 10-11. MMU Assist Register 2 (MAS2)

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

SPR - 626; Read/ Write; Reset - Unaffected

Figure 10-11. MMU Assist Register 2 (MAS2)

Table 10-11. MAS2 field descriptions

Bit	Name	Description
0:21 [32:53]	EPN	Effective page number [0:21]
22:25 [54:57]	—	Reserved[1]
26 [58]	VLE	PowerISA VLE 0 This page is a standard BookE page 1 This page is a PowerISA VLE page This bit will always read as zero and writes will be ignored if **p_vle_present** is negated.
27 [59]	W	Write-through Required 0 This page is considered write-back with respect to the caches in the system 1 All stores performed to this page are written through to main memory
28 [60]	I	Cache Inhibited 0 This page is considered cacheable 1 This page is considered cache-inhibited
29 [61]	M	Memory Coherence Required 0 Memory Coherence is not required 1 Memory Coherence is required
30 [62]	G	Guarded 0 Access to this page are not guarded, and can be performed before it is known if they are required by the sequential execution model 1 All loads and stores to this page are performed without speculation (i.e. they are known to be required) Zen Z7 uses the guarded attribute as described in Section 11.16, Page table control bits, for more information.
31 [63]	E	Endianness 0 The page is accessed in big-endian byte order. 1 The page is accessed in true little-endian byte order. Determines endianness for the corresponding page. Refer to Section 15.2.4, Byte lane specification, for more information

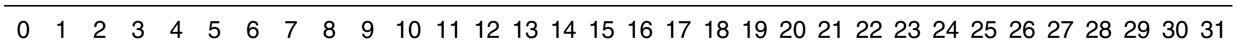
[1] These bits are not implemented, will be read as zero, and writes are ignored.

10.7.3.4 RPN and Access Control register (MAS3)

The MAS3 register is shown in Figure 10-12. Fields are defined in Table 10-12.

RPN	U 0	U 1	U 2	U 3	U X	S X	U W	S W	U R	S R

Figure 10-12. MMU Assist Register 3 (MAS3)



SPR - 627; Read/ Write; Reset - Unaffected

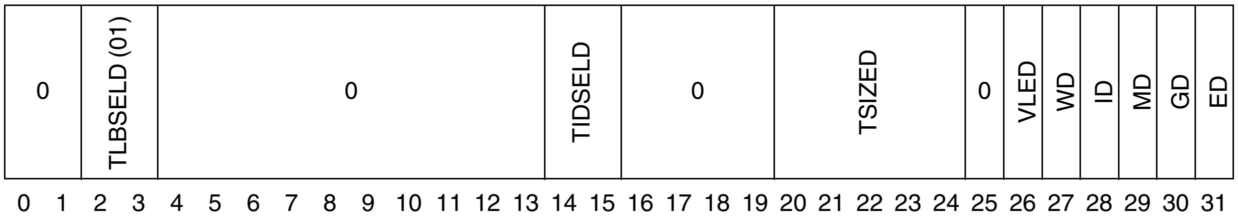
Figure 10-12. MMU Assist Register 3 (MAS3)

Table 10-12. MAS3 field descriptions

Bit	Name	Description
0:21 [32:53]	RPN	Real page number [0:21] Only bits that correspond to a page number are valid. Bits that represent offsets within a page are ignored and should be zero.
22:25 [54:57]	U0-U3	User bits [0-3] for use by system software
26:31 [58:63]	PERMIS	Permission bits (UX, SX, UW, SW, UR, SR)

10.7.3.5 Hardware Replacement Assist Configuration register (MAS4)

The MAS4 register is shown in Figure 10-13. Fields are defined in Table 10-13.



SPR - 628; Read/ Write; Reset - Unaffected

Figure 10-13. MMU Assist Register 4 (MAS4)

Table 10-13. MAS4 field descriptions

Bit	Name	Description
0:1 [32:33]	—	Reserved[1]
2:3 [34:35]	TLBSELD	Default TLB selected 00=TLB0, 01=TLB1
4:13 [36:45]	—	Reserved[1]
14:15 [46:47]	TIDSELD	Default PID# to load TID from 00 PID0 01 Reserved, do not use 10 Reserved, do not use 11 TIDZ (8'h00)) (Use all zeros, the globally shared value)

Table 10-13. MAS4 field descriptions (continued)

Bit	Name	Description
16:19 [48:51]	—	Reserved[1]
20:24 [52:56]	TSIZED	Default TSIZE value
25 [57]	—	Reserved[1]
26 [58]	VLED	Default VLE value
27:31 [59:63]	DWIMGE	Default WIMGE values

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

NOTE

MAS5 is not implemented on the MPC560xS.

10.7.3.6 TLB Search Context Register 0 (MAS6)

The MAS6 register is shown in Figure 10-14. Fields are defined in Table 10-14.

0								SPID								0															SAS
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

SPR - 630; Read/ Write; Reset - Unaffected

Figure 10-14. MMU Assist Register 6 (MAS6)

Table 10-14. MAS6 field descriptions

Bit	Name	Description
0:7 [32:39]	—	Reserved[1]
8:15 [40:47]	SPID	PID value for searches
16:30 [48:62]	—	Reserved[1]
31 [63]	SAS	AS value for searches

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

10.7.4 MAS registers summary

The MAS registers are summarized in Figure 10-15.

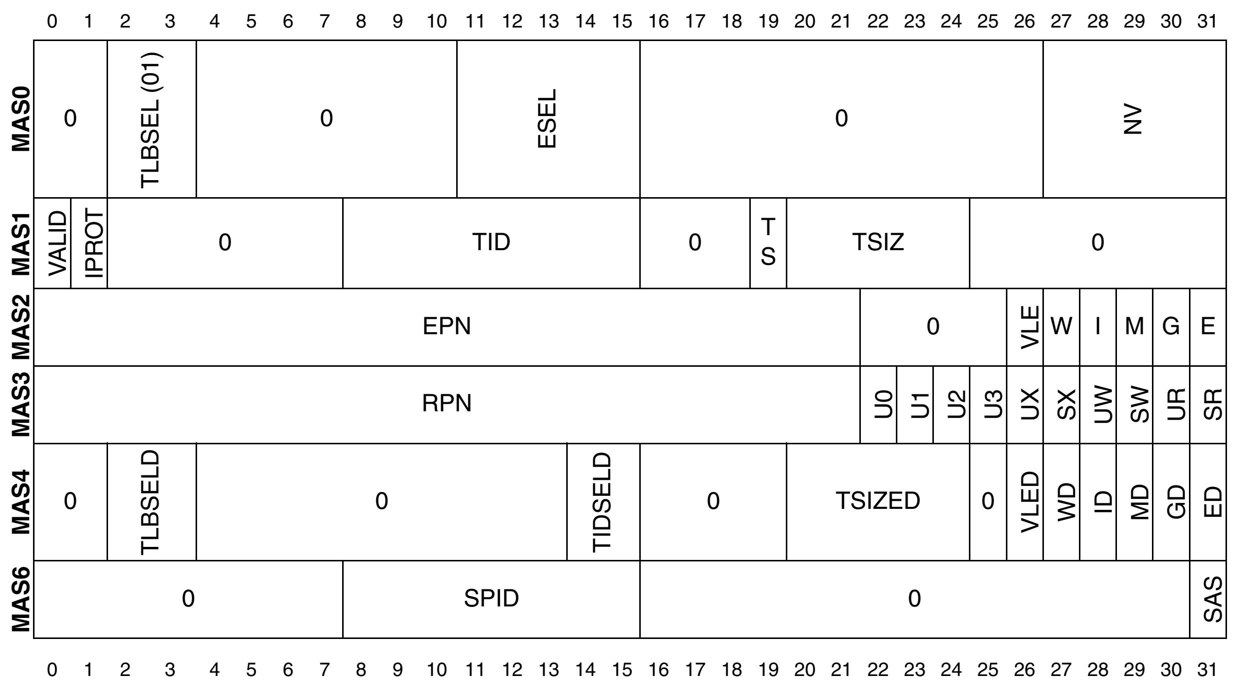


Figure 10-15. MMU assist registers summary

10.7.5 MAS register updates

Table 10-15 details the updates to each MAS register field for each update type.

Table 10-15. MMU assist register field updates

Bit/field	MAS affected	Instr/data TLB error	tlbsx hit	tlbsx miss	tlbre	tlbwe	ISI/DSI
TLBSEL	0	TLBSELD	‘Hitting TLB’	TLBSELD	NC	NC	NC
ESEL	0	NV	matched entry	NV	NC	NC	NC
NV	0	NC	NC	NC	NC	NC	NC
VALID	1	1	1	0	V(array)	NC	NC
IPROT	1	0	Matched IPROT if TLB1 hit, else 0	0	IPROT(array) if TBL1, else 0	NC	NC
TID[0:7]	1	TIDSELD (pid0,TIDZ)	TID(array)	SPID	TID(array)	NC	NC
TS	1	MSR(IS/DS)	SAS	SAS	TS(array)	NC	NC
TSIZE[0:4]	1	TSIZED	TSIZE(array)	TSIZED	TSIZE(array)	NC	NC

Table 10-15. MMU assist register field updates (continued)

Bit/field	MAS affected	Instr/data TLB error	tlbsx hit	tlbsx miss	tlbre	tlbwe	ISI/DSI
EPN[0:21]	2	I/D EPN	EPN(array)	**tlbsx** EPN	EPN(Array)	NC	NC
VWIMGE	2	Default values	VWIMGE(array)	Default values	VWIMGE(array)	NC	NC
RPN[0:21]	3	Zeroed	RPN(Array)	Zeroed	RPN(Array)	NC	NC
ACCESS (PERMISS + U0:U3)	3	Zeroed	Access(Array)	Zeroed	Access(Array)	NC	NC
TLBSELD	4	NC	NC	NC	NC	NC	NC
TIDSELD[0:1]	4	NC	NC	NC	NC	NC	NC
TSIZED[0:4]	4	NC	NC	NC	NC	NC	NC
Default VWIMGE	4	NC	NC	NC	NC	NC	NC
SPID	6	PID0	NC	NC	NC	NC	NC
SAS	6	MSR(IS/DS)	NC	NC	NC	NC	NC

10.8 TLB coherency control

The e200z759n3 core provides the ability to invalidate a TLB entry as described in the Book E Power Architecture architecture. The **tlbivax** instruction invalidates local TLB entries only. No broadcast is performed, as no hardware-based coherency support is provided.

The **tlbivax** instruction invalidates by effective address only. This means that only the TLB entry's EPN bits are used to determine if the TLB entry should be invalidated. It is therefore possible for a single **tlbivax** instruction to invalidate multiple TLB entries, since the AS and TID fields of the entries are ignored.

10.9 Core interface operation for MMU control instructions

MMU control instructions will utilize the normal CPU interface to perform MMU control instructions. The address bus will be driven with the effective address value calculated by the instruction (if any), the access will be treated as a Supervisor Data word-size write, and the Transfer Type encodings will be used to distinguish these operations from other load and store operations. These transfers will not cause debug Data Address Compare matches to occur regardless of the effective address that is driven.

10.9.1 Transfer type encodings for MMU control instructions

Transfer type encodings are used to indicate whether a normal access, atomic access, cache management control access, or MMU management control access is being requested. These attribute signals are driven with addresses when an access is requested. Table 10-16 shows the definitions of the **p_d_ttype[0:5]** encodings.

Table 10-16. Transfer type encoding

p_d_ttype[0:5][1]	Transfer type	Instruction
00000e	Normal	normal loads / stores
000010	Atomic	**lbarx**, **lharx**, **lwarx**, **stbcx.**, **sthcx.**, and **stwcx.**
00010e	Flush Data Block	dcbst
00011e	Flush and Invalidate Data Block	dcbf
00100e	Allocate and Zero Data Block	dcbz
001010	Invalidate Data Block	dcbi
00110e	Invalidate Instruction Block	icbi
001110	Multiple word load/store	lmw, stmw
010000	TLB Invalidate	tlbivax
010010	TLB Search	tlbsx
010100	TLB Read entry	tlbre
010110	TLB Write entry	tlbwe
011000	Touch for Instruction	icbt
011010	Lock Clear for Instruction	icblc
011100	Touch for Instruction and Lock Set	icbtls
011110	Lock Clear for Data	dcblc
10000e	Touch for Data	dcbt
10001e	Touch for Data Store	dcbtst
100100	Touch for Data and Lock Set	dcbtls
100110	Touch for Data Store and Lock Set	dcbtstls

[1] p_ttype[5] 'e' is set to set to 0.

10.10 Effect of hardware debug on MMU operation

Hardware debug facilities utilize normal CPU instructions to access register and memory contents during a *debug session*. If desired during a debug session, the debug firmware may disable the translation process and may substitute default values for the Access Protection (UX, UR, UW, SX, SR, SW) bits, and values obtained from the OnCE Control Register for Page Attribute (VLE, W, I, M, G, E) bits normally provided by a matching TLB entry. In addition, no address translation is performed, and instead, a 1:1 mapping of effective to real addresses is performed. When disabled during the debug session, no TLB miss or TLB Access Protection related DSI conditions will occur. If the debugger desires to use the normal translation process, the MMU may be left enabled in the OnCE OCR, and normal translation (including the possibility of a TLB Miss or DSI) will remain in effect. Refer to Section 12.4.6.3, e200z759n3 OnCE Control Register (OCR), for more detail on controlling MMU operation during debug sessions.

10.11 External translation alterations for realtime systems

In order to support realtime systems in which dynamic mapping of calibration or other data types is needed, the MMU provides special capabilities on a subset of TLB entries. These capabilities allow external hardware to dynamically select one of multiple mappings to one or more physical pages by the same logical address. This capability provides an inexpensive way of dynamically overlaying selected RAM pages on top of read-only memory during runtime. The particular physical page a given logical page maps to can be dynamically altered by means of the **p_extpid[6:7]** inputs. This capability is only provided for TLB1 entries #0 – #15, and only for a restricted subset of PID values.

Enabling of the dynamic mapping capability is controlled by the **p_extpid_en** control input. This input is sampled with the rising edge of the clock, and when asserted, allows for the dynamic remapping capability to be used.

When one or more of TLB1 entries #0 – #15 is programmed with a TID value of 8‘b1111xxxx, special entry-specific logic is enabled for the entry. This logic causes the sampled values of the **p_extpid[6:7]** inputs to be used in place of PID0[6:7] for the purposes of comparison of this entry with the current PID0 register contents to determine an entry hit condition.

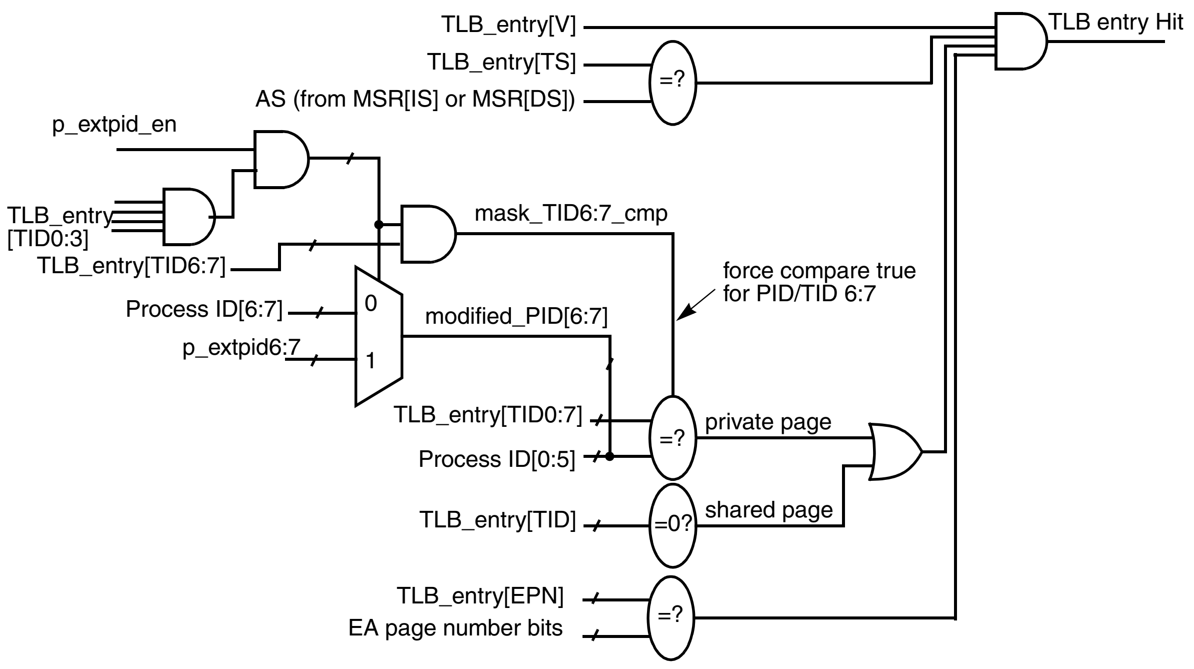
In addition, for those entries within entries #0 – #15 programmed with a TID value of 8‘b1111xx11, the comparison of TID[6:7] to PID0[6:7] for a match is always forced true. This means that the hit condition for these entries is independent of the sampled values of the **p_extpid[6:7]** inputs.

Entries within entries #0 – #15 programmed with a TID value of 8‘b1111nm00, will match a PID0 value of 8‘b1111nmxx when **p_extpid[6:7]** inputs are 00, Those programmed with a TID value of 8‘b1111nm01 will match a PID0 value of 8‘b1111nmxx when **p_extpid[6:7]** inputs are 01, and those programmed with a TID value of 8‘b1111nm10 will match a PID0 value of 8‘b1111nmxx when **p_extpid[6:7]** inputs are 10. Those entries within entries #0 –#15 programmed with a TID value of 8‘b1111nm11, will match a PID0 value of 8‘b1111nmxx regardless of the sampled values of the **p_extpid[6:7]** inputs.

This logic allows application software of this type to set up to three independent mappings for a set of calibration pages, and for external hardware to select between one of the three based on the driven values of the **p_extpid[6:7]** inputs. The other pages are mapped with a common set of entries with stored TID values of 1111xx11, which will match for all sets of calibration page selections. This specialized software must use PID values in the range of 111100xx to 111111xx.

Software is responsible for coordinating the modification to the **p_extpid[6:7]** inputs to ensure they only change when there is no possibility of an error induced by simultaneous use.

Figure 10-16 shows the equivalent logical operation of the capability.



Note: Functionality available for entry # 0-15 only

Figure 10-16. External translation alteration TLB entry compare process

Chapter 11
L1 Cache

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

The L1 Caches incorporate the following features:

- 16 KB I + 16 KB D harvard cache design
- Virtually indexed, Physically tagged
- 32-byte line size
- 64-bit data, 32-bit address
- Pseudo round-robin replacement algorithm
- 8-entry store buffer
- Push (copyback) buffer
- Linefill buffer
- Hit under fill/copyback
- Supports up to two outstanding misses
- Multi-bit EDC protection for the ICache data and tag arrays, with correction/auto-invalidation capability
- Multi-bit EDC protection for the DCache tag arrays, parity protection for the DCache data arrays; with correction/auto-invalidation capability

11.1 Overview

The e200z759n3 processor supports a pair of 16 KB 4-way set-associative split instruction and data caches with a 32-byte line size. The caches improve system performance by providing low-latency data to the e200z759n3 instruction and data pipelines, which decouples processor performance from system memory performance. The caches are virtually indexed and physically tagged.

Instruction and data addresses from the processor to the caches are virtual addresses used to index the cache array. The MMU provides the virtual to physical translation for use in performing the cache tag compare. If the physical address matches a valid cache tag entry, the access hits in the cache. For a read operation, the cache supplies the data to the processor, and for a write operation, the data from the processor updates the cache. If the access does not match a valid cache tag entry (misses in the cache) or a write access must be written through to memory, the cache performs a bus cycle on the system bus.

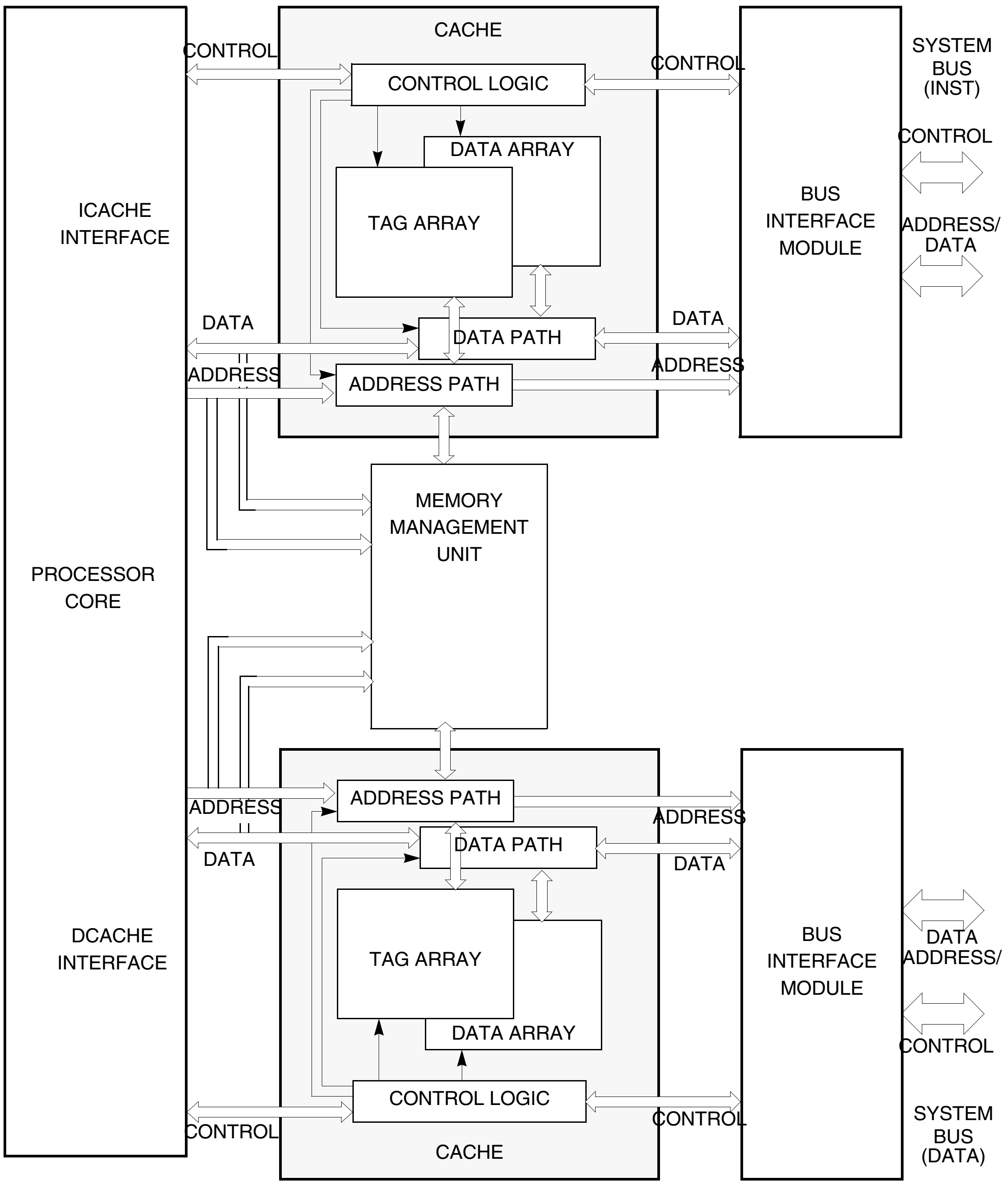


Figure 11-1. e200z759n3 caches

11.2 16 KB cache organization

Each e200z759n3 16 KB cache is organized as four ways of 128 sets with each line containing 32 bytes (four doublewords) of storage. Figure 11-2 illustrates the cache organization along with the cache line format.

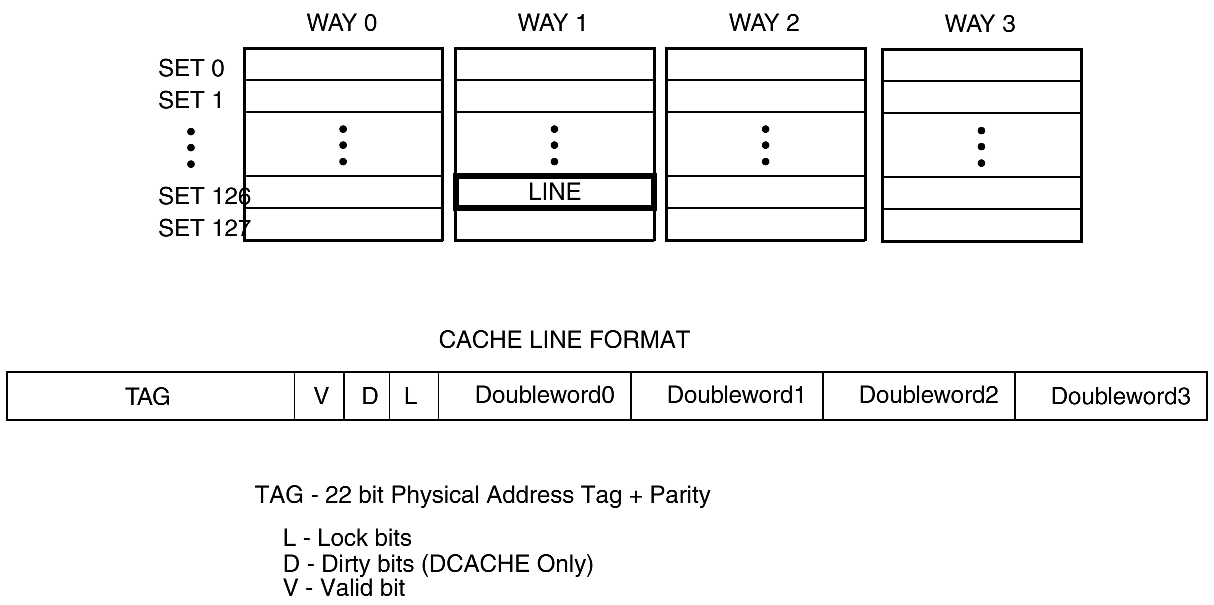


Figure 11-2. 16 KB cache organization and line format

Virtual address bits A[20:26] provide an index to select a set. Ways are selected according to the rules of set association.

Each line consists of a physical address tag, status bits, and four doublewords of data. Address bits A[27:29] select the word within the line.

11.3 Cache lookup

Once enabled, the appropriate cache will be searched for a tag match on instruction fetches and data accesses from the CPU. If a match is found, the cached data is forwarded on a read access to the instruction fetch unit or the load/store unit (data access), or 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 miss address is used to fetch a four doubleword cache line beginning with the requested doubleword (critical doubleword first). The line is fetched into a linefill buffer and the critical doubleword is forwarded to the CPU. Subsequent doublewords may be streamed to the CPU if they have been requested, or they may be forwarded from the linefill buffer if the data has already been received from the bus and is valid in the buffer.

When a write miss occurs, if there is a TLB hit, and the I and G bits of the hitting TLB entry are clear and write allocation is enabled via the L1CSR0[DCWA] control bit, the translated physical address is used to fetch a four doubleword cache line beginning with the doubleword corresponding to the store address (critical doubleword first). The line is fetched into the linefill buffer and merged with the store data. Subsequently, the line is placed into the appropriate cache block. If write allocation is disabled, or the write is not cacheable or is guarded, no cache line fetch is performed for the write.

During a cache line fill, doublewords received from the bus are placed into the cache linefill buffer, and may be forwarded (streamed) to the CPU if such a read request is pending. Accesses from the CPU following delivery of the critical doubleword may be satisfied from the cache (hit under fill, non-blocking) or from the linefill buffer if the requested information has been already received.

If write allocation is enabled, subsequent stores that hit the linefill buffer address while a linefill is in progress for a previous store or **dcbtst** miss will be merged into the linefill buffer. No merging of stores will be performed during a linefill initiated by a load miss.

When a cache linefill occurs, the linefill buffer contents are placed into the cache array using two accesses; each occurs after receiving a pair of doublewords.

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

The caches should be explicitly invalidated after a hardware reset; reset does not invalidate the cache lines. Following initial power-up, the cache contents will be undefined. The L, D and V bits may be set on some lines, necessitating the invalidation of the caches by software before being enabled.

Figure 11-3 illustrates the general flow of cache operation for each 16**16 KB Cache Organization and Line Format** cache to determine if the address is already allocated in the cache.

(1) the cache set index, virtual address bits A[20:26], are used to select one cache set. A set is defined as the grouping of lines (one from each way), corresponding to the same index into the cache array.

(2) The higher order physical address bits A[0:21] , are used as a tag reference or used to update the cache line tag field.

(3)The tags from the selected cache set are compared with the tag reference. If any one of the tags matches the tag reference and the tag status is valid, a cache hit has occurred.

(4) Virtual address bits A[27:28] are used to select one of the four doublewords in each line. A cache hit indicates that the selected doubleword in that cache line contain valid data (for a read access), or can be written with new data depending on the status of the W access control bit from the MMU (for a write access to the DCache).

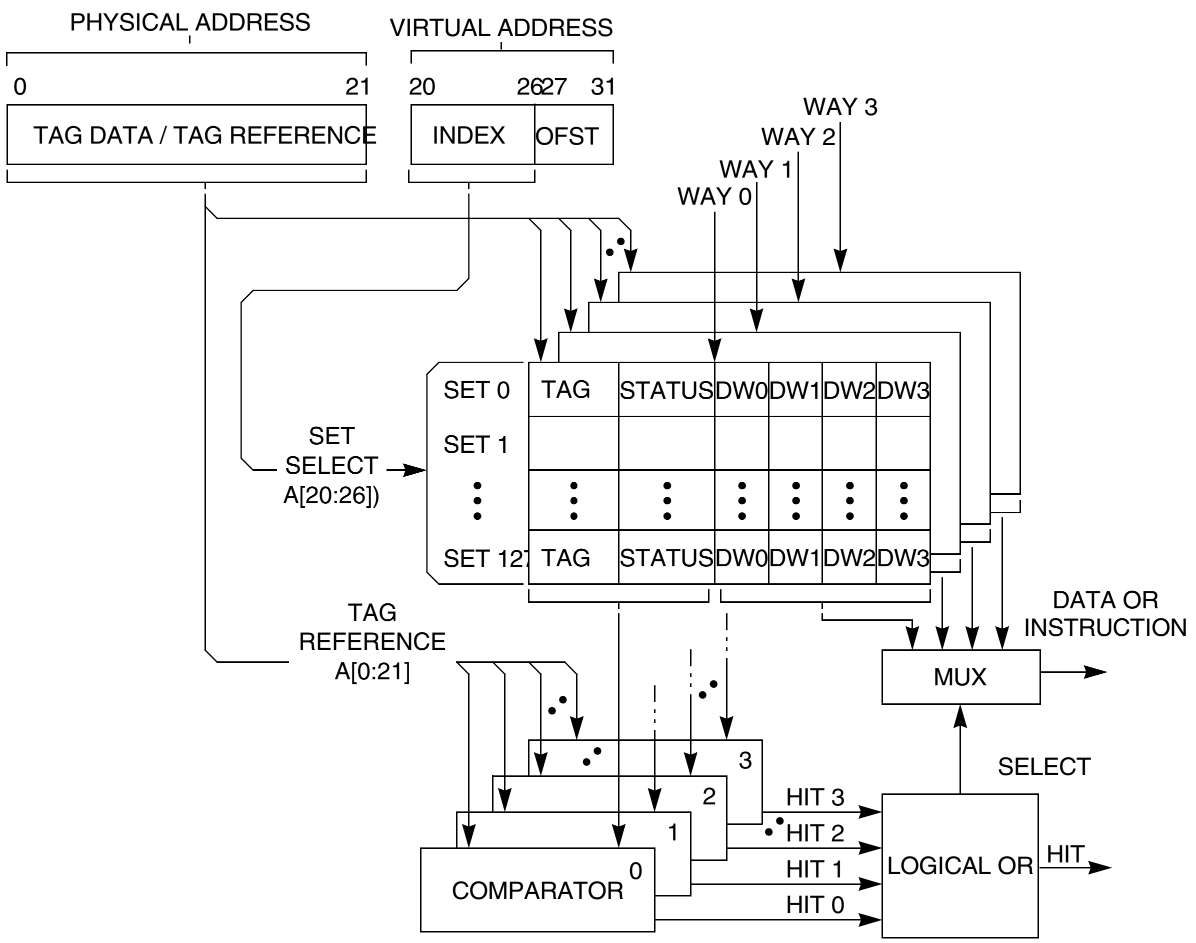


Figure 11-3. 16 KB cache lookup flow

11.4 Cache control

Control of the cache is provided by bits in the L1 Cache Control and Status registers (L1CSR0, L1CSR1). Control bits are provided to enable/disable the cache and to invalidate it of all entries. In addition, availability of each way of the caches may be selectively controlled for use. This way control provides cache way locking capability, as well as controlling way availability on a cache line replacement. Ways 0-3 may be selectively disabled for instruction miss replacements and data miss replacements in the respective caches by using the WID and WDD control bits. Software is responsible for maintaining coherency between instruction and data caches, since independent copies of a cache line may be present in both caches; one allocated by an instruction access, another by a data access.

11.4.1 L1 Cache Control and Status Register 0 (L1CSR0)

The L1 Cache Control and Status Register 0 (L1CSR0) is a 32-bit register used for general control of the data cache as well as providing general control over disabling ways in both caches. The L1CSR0 register is accessed using a **mfspr** or **mtspr** instruction. The SPR number for L1CSR0 is 1010 in decimal. The L1CSR0 register is shown in Figure 11-4.

WID	WDD	0	DCWM	DCWA	0	DCECE	DCEI	0	DCEDT	DCSLC	DCUL	DCLO	DCLFC	DCLOA	DCEA	0	DCBZ32	DCABT	DCINV	DCE
0 1 2 3	4 5 6 7	8 9 10	11	12 13	14	15	16	17	18 19	20	21	22	23	24	25 26	27	28	29	30	31

SPR - 1010; Read/Write; Reset - 0x0

Figure 11-4. L1 Cache Control and Status Register 0 (L1CSR0)

The L1CSR0 bits are described in Table 11-1.

Table 11-1. L1CSR0 field descriptions

Bits	Name	Description
0:3	WID	Way Instruction Disable. 0 The corresponding way in the instruction cache is available for replacement by instruction miss line fills. 1 The corresponding way instruction cache is not available for replacement by instruction miss line fills. Bit 0 corresponds to way 0. Bit 1 corresponds to way 1. Bit 2 corresponds to way 2. Bit 3 corresponds to way 3. The WID bits may be used for locking ways of the instruction cache, and also are used in determining the replacement policy of the instruction cache.
4:7	WDD	Way Data Disable. 0 The corresponding way in the data cache is available for replacement by data miss line fills. 1 The corresponding way in the data cache is not available for replacement by data miss line fills. Bit 4 corresponds to way 0. Bit 5 corresponds to way 1. Bit 6 corresponds to way 2. Bit 7 corresponds to way 3. The WDD bits may be used for locking ways of the data cache, and also are used in determining the replacement policy of the data cache.
8:10	—	Reserved[1]
11	DCWM	Data Cache Write Mode 0 Data Cache operates in writethrough mode 1 Data Cache operates in copyback mode When set to writethrough mode, the “W” page attribute from the MMU is ignored and all writes are treated as writethrough required. When set, write accesses are performed in copyback mode unless the “W” page attribute from the MMU is set.

Table 11-1. L1CSR0 field descriptions (continued)

Bits	Name	Description
12:13	DCWA	Data Cache Write Allocation Policy 00 Cache line allocation on a cacheable write miss is disabled 01 Cache line allocation on a cacheable copyback write miss is enabled 10 Cache line allocation on a cacheable copyback or writethrough write miss is enabled 11 Reserved This field also controls merging of store data into the linefill buffer while a cache linefill is in progress. Store data will not be merged when write allocation is disabled. If DCWA is non-zero, store data merging is enabled regardless of the type (writethrough/copyback) of write.
14	—	Reserved[1]
15	DCECE	Data Cache Error Checking Enable 0 Error Checking is disabled 1 Error Checking is enabled
16	DCEI	Data Cache Error Injection 0 Cache Error Injection is disabled 1 parity errors will be purposefully injected into every byte subsequently written into the cache. The parity bit of each 8-bit data element written will be inverted. This includes writes due to store hits as well as writes due to cache line refills. DCEI will cause injection of errors regardless of the setting of DCECE, although reporting of errors will be masked while DCECE=0.
17	—	Reserved[1]
18:19	DCEDT	Data Cache Error Detection Type 00 Reserved (defaults to DCEDT=01(EDC) actions) 01 EDC Error Detection is selected for the tag array and parity is selected for the data arrays 1x Reserved
20	DCSLC	Data Cache Snoop Lock Clear 0 Snoop has not invalidated a locked line 1 Snoop has invalidated a locked line Indicates a cache line lock was cleared by a snoop operation that caused an invalidation. This bit is set by hardware and will remain set until cleared by software writing 0 to this bit location.
21	DCUL	Data Cache Unable to Lock Indicates a lock set instruction was not effective in locking a cache line. This bit is set by hardware on an “unable to lock” condition (other than lock overflows), and will remain set until cleared by software writing 0 to this bit location.
22	DCLO	Data Cache Lock Overflow Indicates a lock overflow (overlocking) condition occurred. This bit is set by hardware on an “overlocking” condition, and will remain set until cleared by software writing 0 to this bit location.

Table 11-1. L1CSR0 field descriptions (continued)

Bits	Name	Description
23	DCLFC	Data Cache Lock Bits Flash Clear When written to a '1', a cache lock bits flash clear operation is initiated by hardware. Once complete, this bit is reset to '0'. Writing a '1' while a flash clear operation is in progress will result in an undefined operation. Writing a '0' to this bit while a flash clear operation is in progress will be ignored. Cache Lock Bits Flash Clear operations require approximately 134 cycles to complete. Clearing occurs regardless of the enable (DCE) value.
24	DCLOA	Data Cache Lock Overflow Allocate Set by software to allow a lock request to replace a locked line when a lock overflow situation exists. 0 Indicates a lock overflow condition will not replace an existing locked line with the requested line 1 Indicates a lock overflow condition will replace an existing locked line with the requested line
25:26	DCEA	Data Cache Error Action 00 Error Detection causes Machine Check exception. 01 Error Detection causes Correction/Auto-invalidation. No machine check is generated for uncorrectable errors unless the cache line was locked and invalidated or is dirty. Dirty lines are not auto-invalidated. In EDC mode, correction is performed for single-bit tag errors, single-bit lock errors, and single or multi-bit dirty errors. Correction is performed for data errors by reloading of the line. 1x Reserved
27	—	Reserved[1]
28	DCBZ32	Data Cache **dcba**, **dcbz** operation length 0 **dcba**, **dcbz** operations operate on an entire cache line 1 **dcba**, **dcbz** operations operate on 32bytes of a cache line **Note:** This bit is implemented for forward compatibility. Since cache lines are 32 bytes, this bit is ignored for **dcba**, **dcbz** operations
29	DCABT	Data Cache Operation Aborted Indicates a Cache Invalidate or a Cache Lock Bits Flash Clear operation was aborted prior to completion. This bit is set by hardware on an aborted condition, and will remain set until cleared by software writing 0 to this bit location.
30	DCINV	Data Cache Invalidate 0 No cache invalidate 1 Cache invalidation operation When written to a '1', a cache invalidation operation is initiated by hardware. Once complete, this bit is reset to '0'. Writing a '1' while an invalidation operation is in progress will result in an undefined operation. Writing a '0' to this bit while an invalidation operation is in progress will be ignored. Cache invalidation operations require approximately 134 cycles to complete. Invalidation occurs regardless of the enable (DCE) value. During cache invalidations, the parity check bits are written with a value dependent on the DCEDT selection. DCEDT should be written with the desired value for subsequent cache operation when DCINV is set to '1' for proper operation of the cache.

Table 11-1. L1CSR0 field descriptions (continued)

Bits	Name	Description
31	DCE	Data Cache Enable 0 Cache is disabled 1 Cache is enabled When disabled, cache lookups are not performed for normal load or store accesses, or for snoop requests. Other L1CSR0 cache control operations are still available. Also, operation of the store buffer is not affected by DCE.

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

11.4.2 L1 Cache Control and Status Register 1 (L1CSR1)

The L1 Cache Control and Status Register 1 (L1CSR1) is a 32-bit register used for general control of the instruction cache. The L1CSR1 register is accessed using a **mfspr** or **mtspr** instruction. The SPR number for L1CSR1 is 1011 in decimal. The L1CSR1 register is shown in Figure 11-5.

0	ICECE	ICEI	0	ICEDT	0	ICUL	ICLO	ICLFC	ICLOA	ICEA	0	ICABT	ICINV	ICE
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14	15	16	17	18 19	20	21	22	23	24	25 26	27 28	29	30	31

SPR - 1011; Read/Write; Reset - 0x0

Figure 11-5. L1 Cache Control and Status Register 1 (L1CSR1)

The L1CSR1 bits are described in Table 11-2.

Table 11-2. L1CSR1 field descriptions

Bits	Name	Description
0:14	—	Reserved[1]
15	ICECE	Instruction Cache Error Checking Enable 0 Error Checking is disabled 1 Error Checking is enabled
16	ICEI	Instruction Cache Error Injection Enable 0 Cache Error Injection is disabled 1 When ICEDT=01, a double-bit error will be injected into each doubleword written into the cache by inverting the two uppermost parity check bits (p_chk[0:1]). ICEI will cause injection of errors regardless of the setting of ICECE, although reporting of errors will be masked when ICECE=0.
17	—	Reserved[1]
17:24	—	Reserved[1]
18:19	ICEDT	Instruction Cache Error Detection Type 00 Reserved (defaults to ICEDT=01(EDC) actions) 01 EDC Error Detection is selected 1x - Reserved
20	—	Reserved[1]

Table 11-2. L1CSR1 field descriptions (continued)

Bits	Name	Description
21	ICUL	Instruction Cache Unable to Lock Indicates a lock set instruction was not effective in locking a cache line. This bit is set by hardware on an “unable to lock” condition (other than lock overflows), and will remain set until cleared by software writing 0 to this bit location.
22	ICLO	Instruction Cache Lock Overflow Indicates a lock overflow (overlocking) condition occurred. This bit is set by hardware on an “overlocking” condition, and will remain set until cleared by software writing 0 to this bit location.
23	ICLFC	Instruction Cache Lock Bits Flash Clear When written to a ‘1’, a cache lock bits flash clear operation is initiated by hardware. Once complete, this bit is reset to ‘0’. Writing a ‘1’ while a flash clear operation is in progress will result in an undefined operation. Writing a ‘0’ to this bit while a flash clear operation is in progress will be ignored. Cache Lock Bits Flash Clear operations require approximately 134 cycles to complete. Clearing occurs regardless of the enable (ICE) value.
24	ICLOA	Instruction Cache Lock Overflow Allocate Set by software to allow a lock request to replace a locked line when a lock overflow situation exists. 0 Indicates a lock overflow condition will not replace an existing locked line with the requested line 1 Indicates a lock overflow condition will replace an existing locked line with the requested line
25:26	ICEA	Instruction Cache Error Action 00 Error Detection causes Machine Check exception. 01 Error Detection causes Correction/Auto-invalidation. No machine check is generated unless a locked line is invalidated. Correction is performed for single-bit tag and lock errors, and lines with multi-bit tag or lock errors are invalidated. In parity mode, tag or lock errors will result in invalidation of lines. Correction is performed for single or multi-bit data errors by reloading of the line. 1x Reserved
27:28	—	Reserved[1]
29	ICABT	Instruction Cache Operation Aborted Indicates a Cache Invalidate or a Cache Lock Bits Flash Clear operation was aborted prior to completion. This bit is set by hardware on an aborted condition, and will remain set until cleared by software writing 0 to this bit location.

Table 11-2. L1CSR1 field descriptions (continued)

Bits	Name	Description
30	ICINV	Instruction Cache Invalidate 0 No cache invalidate 1 Cache invalidation operation When written to a ‘1’, a cache invalidation operation is initiated by hardware. Once complete, this bit is reset to ‘0’. Writing a ‘1’ while an invalidation operation is in progress will result in an undefined operation. Writing a ‘0’ to this bit while an invalidation operation is in progress will be ignored. Cache invalidation operations require approximately 134 cycles to complete. Invalidation occurs regardless of the enable (ICE) value. During cache invalidations, the parity check bits are written with a value dependent on the ICEDT selection. ICEDT should be written with the desired value for subsequent cache operation when ICINV is set to ‘1’ for proper operation of the cache.
31	ICE	Instruction Cache Enable 0 Cache is disabled 1 Cache is enabled When disabled, cache lookups are not performed for instruction accesses. Other L1CSR1 cache control operations are still available and are not affected by ICE.

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

11.4.3 L1 Cache Configuration Register 0 (L1CFG0)

The L1 Cache Configuration Register 0 (L1CFG0) is a 32-bit read-only register. L1CFG0 provides information about the configuration of the e200z759n3 L1 data cache design. The contents of the L1CFG0 register can be read using a **mfspr** instruction. The SPR number for L1CFG0 is 515 in decimal. The L1CFG0 register is shown in Figure 11-6.

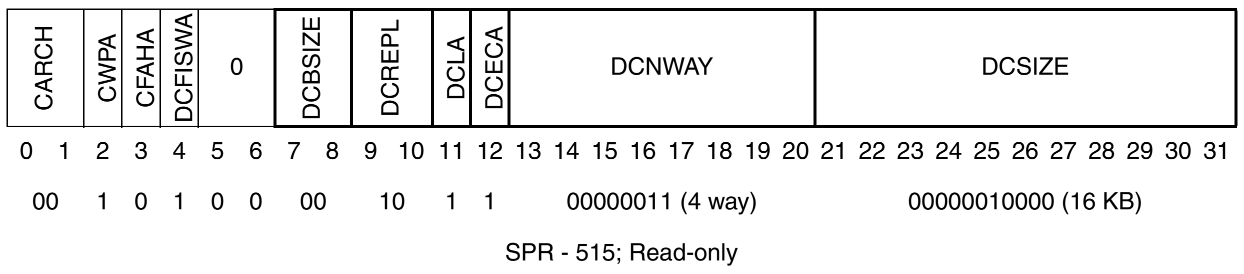


Figure 11-6. L1 Cache Configuration Register 0 (L1CFG0)

The L1CFG0 bits are described in Table 11-3.

Table 11-3. L1CFG0 field descriptions

Bits	Name	Description
0:1	CARCH	Cache Architecture 00 The cache architecture is Harvard
2	CWPA	Cache Way Partitioning Available 1 The caches support partitioning of way availability for I/D accesses

Table 11-3. L1CFG0 field descriptions (continued)

Bits	Name	Description
3	DCFAHA	Data Cache Flush All by Hardware Available 0 The data cache does not support Flush All in Hardware
4	DCFISWA	Data Cache Flush/Invalidate by Set and Way Available 1 The data cache supports flushing/invalidation by Set and Way via the L1FINV0 spr
5:6	—	Reserved - read as zeros
7:8	DCBSIZE	Data Cache Block Size 00 The data cache implements a block size of 32 bytes
9:10	DCREPL	Data Cache Replacement Policy 10 The data cache implements a pseudo-round-robin replacement policy
11	DCLA	Data Cache Locking APU Available 1 The data cache implements the line locking APU
12	DCECA	Data Cache Error Checking Available 1 The data cache implements error checking
13:20	DCNWAY	Data Cache Number of Ways 0x03 The data cache is 4-way set-associative
21:31	DCSIZE	Data Cache Size 0x010The size of the data cache is 16 KB.

11.4.4 L1 Cache Configuration Register 1 (L1CFG1)

The L1 Cache Configuration Register 1 (L1CFG1) is a 32-bit read-only register. L1CFG1 provides information about the configuration of the e200z759n3 L1 instruction cache design. The contents of the L1CFG1 register can be read using a **mfspr** instruction. The SPR number for L1CFG1 is 516 in decimal. The L1CFG1 register is shown in Figure 11-7.

0	ICFISWA	0	ICBSIZE	ICREPL	ICLA	ICECA	ICNWAY	ICSIZE
0 1 2 3	4	5 6	7 8	9 10	11	12	13 14 15 16 17 18 19 20	21 22 23 24 25 26 27 28 29 30 31
0000	1	0 0	00	10	1	1	00000011 (4 way)	00000010000 (16 Kbyte)

SPR - 516; Read-only

Figure 11-7. L1 Cache Configuration Register 1 (L1CFG1)

The L1CFG1 bits are described in Table 11-4.

Table 11-4. L1CFG1 field descriptions

Bits	Name	Description
0:3	—	Reserved - read as zeros
4	ICFISWA	Instruction Cache Flush/Invalidate by Set and Way Available 1 The instruction cache supports invalidation by Set and Way via the L1FINV1 spr
5:6	—	Reserved - read as zeros
7:8	ICBSIZE	Instruction Cache Block Size 00 The instruction cache implements a block size of 32 bytes
9:10	ICREPL	Instruction Cache Replacement Policy 10 The instruction cache implements a pseudo-round-robin replacement policy
11	ICLA	Instruction Cache Locking APU Available 1 The instruction cache implements the line locking APU
12	ICECA	Instruction Cache Error Checking Available 1 The instruction cache implements error checking
13:20	ICNWAY	Instruction Cache Number of Ways 0x03 The instruction cache is 4-way set-associative
21:31	ICSIZE	Instruction Cache Size 0x010The size of the instruction cache is 16 KB.

11.5 Data cache software coherency

Data cache coherency is supported through software operations to invalidate, flush dirty lines to memory or invalidate dirty lines. The data cache may operate in either writethrough or copyback modes, and in conjunction with a MMU, may designate certain accesses as writethrough or copyback. Data cache misses will force the push and store buffers to empty prior to performing the access to ensure coherency.

11.6 Address aliasing

Each cache is virtually indexed and physically tagged, thus the problems associated with potential cache synonyms due to effective address aliasing are eliminated, unless 1Kbyte or 2Kbyte pages are used. If 1Kbyte or 2Kbyte pages are used and multiple virtual addresses are mapped to the same physical address, the low order virtual address bits used to index the cache (A[20:21] for 1Kbyte pages, A20 for 2Kbyte pages) must be the same for each of the virtual pages, and these index bit(s) must match the corresponding physical address bit(s) value. For example, if logical pages X and Y map to physical page P, then X, Y, and P must have the same values of A[20:21] for 1Kbyte pages, and A20 for 2Kbyte pages. Note that this limitation should already met because of the requirements on 1Kbyte and 2Kbyte page usage mandated by Section 10.2.6, Restrictions on 1 KB and 2 KB page size usage.

11.7 Cache Operation

11.7.1 Cache enable/disable

The caches are enabled or disabled by using the respective Cache Enable bits, L1CSR0$_{DCE}$ and L1CSR1$_{ICE}$. Cache Enable bits are cleared by power-on reset or normal reset, disabling the caches.

When a cache is disabled, the cache tag status bits are ignored, and the cache is not accessed for snoops, normal loads, stores, or instruction fetches. All normal accesses are propagated to the system bus as single-beat (non-burst) transactions.

Note that the state of the Cache Inhibited access attribute (the I bit) remains independent of the state of L1CSR0$_{DCE}$ and L1CSR1$_{ICE}$. Disabling a cache does not affect the translation logic in the Memory Management Unit. Translation attributes will still be used when generating attribute information on the system buses.

The store buffer is still available for use even when the data cache is disabled.

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

All cache operations are affected by disabling the cache. Cache management instructions (except for **mtspr** L1FINV{0,1} and **mtspr** L1CSR{0,1}) do not affect a cache when it is disabled.

11.7.2 Cache fills

Cache line fills are requested when a cacheable load or instruction miss occurs. Cacheable store misses only allocate cache lines if data cache write allocation is enabled for the type of store being performed.

The cache line fill is performed critical doubleword first on the bus using a burst access. The critical doubleword is forwarded to the requesting unit before being written to the cache, thus minimizing stalls due to fill delays. Cache line fills load a four doubleword linefill buffer, and updates to the cache array are performed as half-lines are received.

Read accesses may hit in the line buffer and data supplied from the buffer to the CPU. On writes that hit to the buffer address, when write allocation is disabled, the writes will stall until the cache fill has been completed. When write allocation is enabled, these writes will update the linefill buffer if the buffer is being filled due to a store miss only, otherwise the write will also stall until the linefill completes.

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

The cache supports up to three outstanding misses, and will forward these miss requests to the BIU. Miss data is always returned from the BIU to the Cache in-order.

Cache fill operations are performed as wrapping bursts on the system bus. If an error response is received on any element of the burst, the burst will be terminated, and the cache line will be marked invalid.

If one or more store hit updates occur to the linefill buffer during allocation of a line for a store miss and a subsequent error response is received during the linefill, the original store miss access and each individual hitting store access will be performed on the system bus as if they were non-allocating. In this case, an async machine check exception will be signaled for the linefill.

11.7.3 Cache line replacement

On a cache miss, the cache controller uses a pseudo-round-robin replacement algorithm to determine which cache line will be selected to be replaced. There is a single replacement counter for each cache. The replacement algorithm acts as follows: On a miss, if the replacement pointer is pointing to a way that is not enabled for replacement (the selected line or way is locked), it is incremented until an available way is selected (if any). After a cache line is successfully filled without error, the replacement pointer increments to point to the next cache way. If no way is available for the replacement, the access is treated as a single beat access and no cache linefill occurs.

Lines selected for replacement that are dirty (modified) must be copied back to main memory. This is performed by first storing the replaced line in a 32-byte push buffer while the missed data is fetched. After filling the new line, the contents of the buffer are written to memory beginning with doubleword 0.

Each replacement counter is initialized to point to way 0 on a reset or on a respective cache invalidate all operation. A replacement counter may also be set to a specific value via a L1FINV0,1 command.

11.7.4 Cache miss access ordering

Cacheable cache misses may be processed out-of-order by e200z759n3. Load misses that are not cache-inhibited are allowed to bypass buffered stores and push buffer pushes as long as no address alias exists. Alias checking is performed by comparing the index of the load with the index of each buffered store and push. If no alias match exists, the load is allowed to bypass buffered stores and pushes, regardless of the attributes associated with those stores. Load misses will be performed in-order with respect to other load misses. Store accesses do not bypass loads. Stores are not necessarily performed in order from the point of view of the memory system, since a store miss may cause a linefill to satisfy the store prior to previously buffered stores being completed, as long as no aliasing occurs.

Memory access ordering must be enforced by software where required, using the **mbar** and/or **msync** instructions, per the PowerArch storage ordering rules.

11.7.5 Cache-inhibited accesses

When the Cache-Inhibited attribute is indicated by translation and a cache miss occurs, all accesses are performed as single beat transactions on the system bus. Cache Inhibited status is ignored on all cache hits. For cache-inhibited load access misses, the processor termination is withheld for the load until the store buffer has been flushed of all entries, the push buffer has been emptied, and the load has completed to memory. Cache-inhibited store accesses that are not marked as Guarded are placed in the store buffer (when enabled) and the processor termination occurs when the store buffer entry is allocated. (see Section 11.9, Push and store buffers).

11.7.6 Guarded accesses

When the Guarded attribute is indicated by translation and a cache miss occurs, the access will not proceed on the external bus until all previously initiated demand-accesses have been terminated to the processor without error. Buffered stores are considered terminated to the processor when they are placed into the store buffer. Guarded load misses that are not cache-inhibited are allowed to bypass buffered stores and push buffer pushes as long as no address alias exists, regardless of a buffered store being guarded. Guarded stores will not allocate cache lines on a miss, but are buffered in the store buffer if the access is not also cache-inhibited, regardless of being writethrough required or not (regardless of W bit or $L1CSR0_{DCWM}$ values), and will be performed as single-beat accesses on the bus.

11.7.7 Cache-inhibited guarded accesses

When the Cache-inhibited and Guarded attributes are indicated by translation and a cache miss occurs, accesses are performed as single beat transactions on the system bus. Cache-inhibited status is normally ignored on all cache hits. Cache-inhibited status for writethrough stores that are also guarded will not be ignored however. For cache-inhibited guarded access misses, or for cache-inhibited guarded writethrough store hits, the processor termination is withheld until the store buffer has been flushed of all entries, the push buffer has been emptied, and the access has completed to memory (see Section 11.9, Push and store buffers). Cache-inhibited guarded stores with W=0 or $L1CSR0_{DCWM}$=1 that hit ignore the Cache-inhibited and Guarded status.

11.7.8 Cache invalidation

e200z759n3 supports full invalidation of the caches under software control. The caches may be invalidated through the $L1CSR0_{DCINV}$ and $L1CSR1_{ICINV}$ cache invalidate control bits. This function is available even when a cache is disabled.

Reset does not invalidate a cache automatically. Software must use the {D,I}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 {D,I}CINV causes a flash invalidation to be initiated, which lasts for multiple (approx. 134) CPU cycles. Once set, the {D,I}CINV bit will be cleared by hardware after the operation is complete. It will remain set during the invalidation interval, and may be tested by software to determine when the operation has completed. A **mtspr** operation to L1CSR{0,1} that attempts to change the state of {D,I}CINV during invalidation will not affect the state of that bit.

In order to properly generate the tag parity/check bits during the invalidation process, the error detection type control located in the $L1CSR[0,1]_{[D,I]CEDT}$ field should be configured properly at the time the invalidation operation is initiated. A subsequent change to the error detection type control will require a new invalidation to avoid improper interpretation of previously stored tag parity/check bits.

During the process of performing the invalidation, a cache does not respond to accesses other than snoop accesses, and remains busy. Interrupts may still be recognized and processed, potentially aborting the invalidation operation. When this occurs, the $L1CSR\{0,1\}_{ABT}$ bit will be set to indicate unsuccessful completion of the operation. Software should read the L1CSR{0,1} register to determine that the operation has completed (L1CSR{0,1}$_{CINV}$ bit cleared), and then check the status of the L1CSR{0,1}$_{ABT}$ bit to determine completion status.

NOTE

Note that while most implementations of the e200z759n3 will stall further instruction execution during this invalidation interval, it is not guaranteed across all implementations, thus software should be written using these guidelines.

Individual cache lines may be invalidated using the **icbi**, **dcbi**, or **dcbf** instructions. These instructions require the respective cache to be enabled in order to operate normally.

11.7.9 Cache flush/invalidate by set and way

e200z759n3 supports cache flushing under software control. The caches may be flushed and/or invalidated by index and way through a **mtspr l1finv{0,1}** instruction.

The L1 Flush and Invalidate Control Registers (L1FINV{0,1}) are 32-bit SPRs used to select a cache set and way to be flushed/invalidated. No tag match is required. This function is available even when a cache is disabled. L1FINV0 is used for data cache operations, while L1FINV1 is used for instruction cache operations.

11.7.9.1 L1 Flush and Invalidate Control Register 0 (L1FINV0)

The SPR number for L1FINV0 is 1016 in decimal. The L1FINV0 register is shown in Figure 11-8.The L1FINV0 bits are described in Table 11-5.

0	CWAY	0	CSET	0	CCMD
0 1 2 3 4 5	6 7	8 9 10 11 12 13 14 15 16 17 18 19	20 21 22 23 24 25 26	27 28 29	30 31

SPR - 1016; Read/Write; Reset - 0x0

Figure 11-8. L1 Flush/Invalidate Register 0 (L1FINV0)

Table 11-5. L1FINV0 field descriptions

Bits	Name	Description
0:5	—	Reserved[1] for way extension
6:7	CWAY	Cache Way Specifies the data cache way to be selected
8:19	—	Reserved[1] for set extension
20:26	CSET	Cache Set Specifies the cache set to be selected

Table 11-5. L1FINV0 field descriptions (continued)

Bits	Name	Description
27:29	—	Reserved[1] for set/command extension
30:31	CCMD	Cache Command 00 The data contained in this entry is invalidated without flushing 01 The data contained in this entry is flushed if dirty and valid without invalidation 10 The data contained in this entry is flushed if dirty and valid and then is invalidated 11 Reset way replacement pointer to the way indicated by CWAY

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

For cache flush operations, if a transfer error occurs on a data cache line flush, the push of the remaining portion of the cache line is aborted, the line remains marked dirty and valid, and a machine check condition is signaled

For flush and flush with invalidation operations, data parity errors do not abort a flush to memory, but a machine check will be generated at the completion of the flush. In both cases the cache line is left unchanged. For flush with invalidation operations to clean lines, tag parity errors and data parity errors are ignored, and the line is invalidated. Note that only the line indicated by CSET and CWAY is checked for errors; lines in the other ways are ignored.

For invalidation without flushing operations, tag parity errors, data parity errors, and dirty-bit parity errors are ignored, and the line will be invalidated.

11.7.9.2 L1 Flush and Invalidate Control Register 1 (L1FINV1)

The SPR number for L1FINV1 is 959 in decimal. The L1FINV1 register is shown in Figure 11-9. The L1FINV1 bits are described in Table 11-6.

0						CWAY		0												CSET							0			CCMD	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

SPR - 959; Read/Write; Reset - 0x0

Figure 11-9. L1 Flush/Invalidate Register 1 (L1FINV1)

Table 11-6. L1FINV1 field descriptions

Bits	Name	Description
0:5	—	Reserved[1] for way extension
6:7	CWAY	Cache Way Specifies the instruction cache way to be selected
8:19	—	Reserved[1] for set extension
20:26	CSET	Cache Set Specifies the instruction cache set to be selected

Table 11-6. L1FINV1 field descriptions (continued)

Bits	Name	Description
27:29	—	Reserved[1] for set/command extension
30:31	CCMD	Cache Command 00 The data contained in this entry is invalidated 01 Reserved 10 Reserved 11 Reset way replacement pointer to the way indicated by CWAY

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

11.8 Cache parity and EDC protection

Cache parity is supported for both the tag and data arrays of each cache. Six parity check bits are provided for each tag entry for the tag arrays of both caches to support multi-bit error detection (EDC), and redundant dirty bits are provided in the data cache to provide dirty-bit parity checking without requiring a read-modify-write operation when the dirty bit is set. Redundant lock bits are provided as well for both the ICache and the DCache. Byte parity is supported for the data arrays of the data cache, and eight parity check bits are provided for each doubleword in the data arrays of the ICache, which are used for multi-bit error detection (EDC–DED, double error detection). Utilizing EDC protection, many multi-bit errors are also detected.

Parity and EDC checking is controlled by the L1CSR0$_{DCECE}$, L1CSR0$_{DCEDT}$, L1CSR1$_{ICECE}$, and L1CSR1$_{ICEDT}$ control fields. When error checking is enabled, checking is performed on each cache access, whether for lookup, snoop lookup, or for dirty line replacement. Parity or EDC errors are not signaled by the respective cache when cache error checking is disabled for that cache (L1CSR[0,1]$_{[I,D]CECE}$=0).

For normal cache lookups due to instruction fetching, loads, or stores, if an uncorrectable tag EDC error is detected on any portion of the accessed tags, a parity error is signaled, regardless of whether a cache hit or miss occurs. Otherwise, if a cache hit for a load occurs and a data parity error is detected on any portion of the accessed doubleword of data, a parity error is also signaled. Data parity errors are ignored for store hits, since the parity will be updated for the data being stored. Data parity errors are ignored for misses unless the replacement line is dirty or incurs a dirty bit parity error, since the parity will be updated for the new linefill data being stored.

Signaling of a parity error may not cause an exception to occur, depending on the error detection action to be taken. Instead, a correction/auto-invalidation cycle may be performed.

A dirty line push will not be generated for a dirty line replacement that incurs an uncorrectable tag EDC error. In this case, a machine check will be generated, but no push will have been requested to the external bus, and the cache line will be left unchanged. For dirty line pushes from the data cache, accessing the data arrays for the push data may occur after the burst write has been requested on the external bus, thus a push of dirty data may actually push data that contains a parity error. A machine check will be signaled, but the burst will not be aborted, and the line will be invalidated and replaced.

Dirty bit parity is checked when invalidation or replacement operations are required. If a dirty parity error is detected on a cache line replacement, in correction/autoinvalidation mode, it is ignored, and the line is pushed normally. In machine check mode, a machine check exception will be signaled indicating a tag parity error. Dirty status or dirty parity errors will prevent the auto-invalidation of cache lines with tag EDC errors. If a dirty parity error occurs, in correction/autoinvalidation mode the line is assumed to be dirty, and if correction/auto-invalidation is enabled, the error will be corrected by re-writing all three dirty bits to ‘1’. This implies that a single or multi-bit error that sets one or more dirty bits from an initially cleared state will cause the line to appear dirty. This should not cause a functional issue however, since the only result is that a clean but coherent line may be pushed on a flush or replacement in correction/autoinvalidation mode.

Regardless of the error action mode indicated by {D,I}CEA, lock bit parity errors will not signal an exception for normal hits without a tag parity error. If correction/auto-invalidation is enabled, on each cache lookup operation, if a single-bit lock error is detected in one or more ways, it will be corrected by re-writing all lock bits to the correct state. Uncorrectable lock errors will remain unchanged. For cache hits without a tag EDC error, all lock parity errors are ignored. Lock parity errors on a cacheable miss (after a correction attempt if correction/auto-invalidation is enabled) will result in the line(s) being invalidated if clean and a machine check to be generated. A new line will not be allocated, and the lock bits will not be updated on the invalidation. Lock bit parity errors are ignored for non-cacheable accesses.

Signaling of a parity error or EDC error may cause a Machine Check exception to occur, and one or more syndrome bits to be set in the Machine Check Syndrome register, or may instead result in a correction/auto-invalidation operation and not result in an exception being signaled, or both may occur, depending on the error action control setting in the appropriate cache control register. Refer to Section 11.8.1, Cache error action control, for details of the cache error action controls. Refer to Section 7.7.2, Machine Check interrupt (IVOR1), and to Section 2.4.7, Machine Check Syndrome Register (MCSR), for a description of Machine Check conditions.

11.8.1 Cache error action control

The $L1CSR0_{DCEA}$ and $L1CSR1_{ICEA}$ control fields allow for selection of several policies to apply when errors are detected during a cache lookup, and are described in the following subsections.

11.8.1.1 $L1CSR[0,1]_{[I,D]CEA}$ = 00, machine check generation on error

Selection of the machine check generation on error policy allows for all errors to be processed by software. Parity or EDC errors that could result in incorrect operation will cause a machine check condition. In order to be recoverable, the machine check handler must not incur another parity or EDC error during the initial portion of the machine check handler. Parity/EDC errors will not generate a machine check exception for cache-inhibited accesses.

If machine check generation on error is enabled ($L1CSR[0,1]_{[I,D]CEA}$=00) and an EDC error is detected on any portion of the accessed tags for a cacheable load or store access, a machine check is reported, regardless of whether a cache hit or miss occurs. Otherwise, if a cache hit occurs and a parity or EDC error is detected on any portion of the accessed doubleword of data for a load or an instruction access, a machine check is also reported. For store accesses, data parity errors are ignored. Lock or dirty parity errors on a cacheable miss will cause a machine check to be reported indicating a lock error and/or a tag parity error. Dirty parity errors on a cache hit for a reservation instruction (**lwarx**, **stwcx.**, etc.) will result in a machine check and will indicate a tag parity error. If a miss occurs and a tag EDC error is detected on a lookup for a cacheable reservation instruction (**lwarx**, **stwcx.**, etc.), it will be ignored if the line is clean, otherwise if the line is dirty or a dirty parity error occurs, a machine check will be generated and the reservation access will not be run externally. Cache inhibited reservation accesses will ignore all parity/EDC errors.

11.8.1.2 L1CSR[0,1]$_{[I,D]CEA}$ = 01, correction/auto-invalidation on error

The correction/auto-invalidation on error policy attempts to cause most parity and EDC errors to be transparently handled by correcting lines with single-bit tag errors, and invalidating lines with uncorrectable tag errors or with data errors and then causing cache refills to reload correct data from memory, without generation of exceptions. Exceptions are only generated when invalidations could cause or would cause a change in correct behavior, such as changing the locked status of a line, or invalidating potentially dirty data. Parity/EDC errors will not generate invalidations that could cause a machine check exception for cache-inhibited accesses however.

When using EDC protection for the cache tags (L1CSR[0,1]$_{[D,I]CEDT}$=01), single-bit tag errors are corrected by the cache hardware during a correction/auto-invalidation cycle. Clean unlocked lines with multi-bit errors are invalidated on cache hits, with no machine check signaled. Clean locked lines with uncorrectable tag errors are invalidated on cache misses, and a machine check is signaled.

Note that since the data arrays have a higher probability of incurring an error than the tag arrays, due to the relative storage capacities, most errors will be transparently corrected, even if they are double-bit or multi-bit errors. Using writethrough mode for critical data will ensure that invalidation or refills are able to recover from errors transparently in most cases.

11.8.1.2.1 Instruction cache errors

If correction/auto-invalidation on error is enabled (L1CSR1$_{ICEA}$=01) and an error is detected on any portion of the accessed tags or data for an access, a correction/auto-invalidation cycle is inserted, regardless of whether a cache hit or miss occurs. During this cycle, any tag entry with a single-bit tag or lock error is corrected and re-written to correct the stored error. Tag entries with uncorrectable errors are invalidated if unlocked or are invalidated if a cache miss will occur after a correction/auto-invalidation cycle regardless of locked status. If a locked line is invalidated, a machine check will occur, no replacement will occur, and the locked status will remain set for the invalidated line(s) to assist software in determining the location of the error(s).

Following the correction/auto-invalidation cycle, a re-lookup is performed for the access. If a cache hit occurs on a way without a tag EDC error, and an EDC error is detected on any portion of the accessed doubleword of data, a miss is forced, and the same line is refilled from system memory, retaining the existing lock status. The replacement pointer for the cache is not updated in these circumstances. If a cache hit occurs on a way without a tag EDC error, EDC errors on all other lines are ignored, and no invalidations for those lines will occur.

For all cases of invalidations, if any line that was locked or incurred a lock error was invalidated, a machine check will also occur, even though auto-invalidation is selected. Invalidation is not blocked for locked lines or lines with lock parity errors on cache misses. The lock bits will remain unmodified by the invalidation operation to allow for potential software recovery.

If a refill of a locked line due to a data EDC error encounters an external bus error during the linefill, a machine check will be generated, the line will be invalidated, and the lock bits will remain set.

11.8.1.2.2 Data cache errors

If correction/auto-invalidation on error is enabled ($L1CSR0_{DCEA}$=01) and an error is detected on any portion of the accessed tags, or if a lock or dirty parity error is detected, an invalidation/correction cycle is inserted, regardless of whether a cache hit or miss occurs. Following the invalidation/correction cycle, a re-lookup is performed for the access. During the correction/auto-invalidation cycle, any tag entry with a tag or lock error is corrected if possible, and re-written to correct the stored error. Tag entries with uncorrectable errors are invalidated if the line is clean and unlocked, or if the line is clean and a miss will occur after the re-lookup, regardless of lock status. Dirty parity errors are corrected by setting all dirty bits to '1'. Dirty lines and lines with a dirty parity error are not invalidated.

Following the correction/auto-invalidation cycle, a re-lookup is performed for the access. If a cache hit occurs on a way without a tag EDC error, and a parity error is detected on any portion of the accessed doubleword of data for a load, if the line is clean, a miss is forced and the line is refilled from system memory, retaining the existing lock status. The replacement pointer for the cache is not updated in these circumstances. All other clean unlocked lines with uncorrectable tag errors will have been invalidated during the correction/auto-invalidation cycle if one was initially needed. Tag EDC errors on lines that were not invalidated earlier due to lock or dirty status will be ignored since a cache hit occurs. For stores, parity errors on data are ignored, and no invalidation or refill of any lines will occur on a hit to a way without a tag EDC error.

Note that since the data arrays have a higher probability of incurring an error than the tag arrays, due to the relative storage capacities, most errors will be transparently corrected. Using writethrough mode for critical data will ensure that invalidation or refills are able to recover from errors transparently in most cases.

If a cache hit occurs on a way without a tag EDC error, and a parity error is detected on any portion of the accessed doubleword of data for a load, and the line is dirty or a dirty error occurs, no refill of the cache line will occur, the line will not be invalidated, and a machine check will also occur, even if auto-invalidation is selected. All other clean unlocked lines with uncorrectable tag errors will also have been invalidated during the correction/auto-invalidation cycle if one was initially needed. Tag EDC errors on lines that were not invalidated earlier due to lock or dirty status will be ignored

If a cache hit occurs only on a line(s) with an uncorrectable tag EDC error after a invalidation /correction cycle has been performed, since the line is dirty or has a dirty parity error (it would have been invalidated otherwise), a machine check is generated, and no linefill is performed.

If a cache miss occurs and any line with an uncorrectable tag EDC error is dirty or has a dirty parity error, the line is not invalidated, a machine check is generated, and no linefill is performed. All clean lines with tag errors will have been invalidated/corrected on a cache miss, regardless of locked status.

For all cases of invalidations, if any line that was locked or incurred a lock error was invalidated, a machine check will also occur, even though auto-invalidation is selected. Invalidation on a miss is not blocked for locked lines or lines with lock parity errors unless the access is cache-inhibited or is dirty. The lock bits will remain unmodified by the invalidation operation to allow for potential software recovery.

If a refill of a locked line due to a data parity error encounters an external bus error during the linefill, a machine check will be generated, the line will be invalidated, and the lock bits will remain set.

11.8.1.2.3 Data cache line flush or invalidation due to reservation instructions (l[b,h,w]arx, st[b,h,w]cx.)

Normally, when executing a load and reserve, or a store conditional instruction, a cache line hit results in the line being pushed (if dirty) and marked clean, and the reservation access performed as a single-beat access. Certain parity or EDC errors may cause other actions however.

If a cache hit to a line with no tag EDC error occurs when performing a lookup for a load or store reservation access, the line will be pushed if dirty, or if a dirty parity error occurs, and will be marked as clean. Locked status will not be changed. A push parity error may occur during the push if a data parity error is encountered, and a machine check will be generated. In this case the reservation access will not be performed. Otherwise, a load reservation access is then performed as a single-beat access, ignoring the cache data. A store reservation access is performed as a writethrough single-beat write access on the bus, regardless of whether it is marked as writethrough required. If the write access completes without error and succeeds (no ERROR or XFAIL response from the bus), then the cache is updated with the store data, but the line is left in a clean state. Uncorrectable tag errors on other clean unlocked lines will cause invalidation of those lines without signaling a machine check. Uncorrectable tag errors on other cache lines that are locked or are dirty will be ignored.

Otherwise, if any line has an uncorrectable tag EDC error and is dirty or has a dirty parity error, a machine check is generated, and the line(s) remains unchanged. Clean unlocked lines with tag EDC errors will be invalidated or corrected, but locked lines or lines with a lock error will not be invalidated on a cache miss, since no new cache line will be allocated.

11.8.2 Parity/EDC error handling for cache control operations and instructions

Parity/EDC errors are not signaled when the respective L1CSR0$_{\mathrm{DCECE}}$ and L1CSR1$_{\mathrm{ICECE}}$ cache error checking enable bits are cleared. When set, the following sections describe error handling for cache control operations and cache control instructions.

11.8.2.1 L1FINV[0,1] operations

For invalidation operations via the L1FINV[0,1] control registers, uncorrectable tag EDC errors will result in the specified line being invalidated, and no error will be reported, regardless of the setting of L1CSR[0,1]$_{\mathrm{[I,D]CEA}}$. Data parity or EDC errors and dirty errors are ignored. Parity or EDC errors on all other ways not specified by the CWAY value for the L1FINV[0,1] are ignored, regardless of the settings of L1CSR[0,1]$_{\mathrm{[D,I]CEA}}$.

For flush and flush with invalidate operations via the L1FINV0 control register, if no uncorrectable tag EDC error occurs on the specified line, it is flushed to memory if dirty or if a dirty parity error occurs, and then invalidated for flush with invalidate operations, and no machine check is signaled for dirty parity errors. If an uncorrectable tag EDC error occurs on the specified line, and the line is dirty or a dirty error is encountered, no flush or invalidation will be performed, the line will remain unchanged, and a machine check will be generated. For flush operations, an uncorrectable tag EDC error on a clean line will be ignored, and no error will be reported. For flush with invalidate operations, an uncorrectable tag EDC error on a clean line will result in the specified line being invalidated, and no error will be reported. Lock status is ignored for these operations. Data parity errors may result in a push parity error and a machine check generated, but the line will still be flushed to memory if not prevented due to an uncorrectable tag EDC error. If a push parity error occurs, the line will be left unaffected for flush with invalidate operations. Lock status will be cleared on an invalidation or flush with invalidation that does not result in a machine check.

11.8.2.2 Cache touch instructions (dcbt, dcbtst, icbt)

Parity errors are not signaled on a lookup for a **dcbt**, **dcbtst**, or **icbt** instruction. For those instructions, an uncorrectable tag EDC error results in a nop and no error is reported, regardless of error checking being enabled. No invalidations will occur.

11.8.2.3 icbi instructions

For **icbi** instructions, on a hit to any locked or unlocked line without an uncorrectable tag EDC error (with or without a lock parity error), or on a hit to an unlocked line with an uncorrectable tag EDC error, the line(s) is invalidated, regardless of the setting of $L1CSR1_{ICEA}$, and no machine check is generated. If $L1CSR1_{ICEA}$ = ‘01’, if any line has a tag EDC error, a correction/invalidation cycle is inserted to correct tags with single-bit errors, and to invalidate unlocked lines with multi-bit errors. Locked lines with uncorrectable tag errors that miss are unaffected. No machine check will be generated.

If a hit occurs to a line with a tag EDC error (after a correction for $L1CSR1_{ICEA}$ = ‘01’) that is locked or has a lock parity error, the line is left unaffected, and no machine check is generated, regardless of the setting of $L1CSR1_{ICEA}$.

If a miss occurs, all parity/EDC errors are ignored, the lines are left unaffected, and no machine check is generated, regardless of the setting of $L1CSR1_{ICEA}$.

All data EDC errors are ignored regardless of $L1CSR1_{ICEA}$.

11.8.2.4 dcbi instructions

For **dcbi** instructions, on a hit to a line without a tag EDC error, the line is invalidated, regardless of the setting of $L1CSR0_{DCEA}$. For this case, data, lock, and dirty parity errors are ignored. When $L1CSR0_{DCEA}$ = ‘00’, tag parity/DC errors on other lines are ignored. When $L1CSR0_{DCEA}$ = ‘01’, uncorrectable tag EDC errors on other lines will also cause clean unlocked lines to be invalidated, regardless of hit or miss. No machine check is generated regardless of the setting of $L1CSR0_{DCEA}$.

For **dcbi** instructions that hit to a line with a tag EDC error, the line(s) is invalidated if clean and unlocked and no machine check is generated, regardless of the setting of $L1CSR0_{DCEA}$. Uncorrectable tag EDC errors will cause other clean unlocked lines to be invalidated when $L1CSR0_{DCEA}$ = ‘01’, regardless of hit or miss. If a hit occurs to a line with an uncorrectable tag EDC error and the line is dirty, or is locked or has a lock parity error, the line is left unaffected, and no machine check is generated, regardless of the setting of $L1CSR0_{DCEA}$.

For **dcbi** instructions that miss in all ways, when L1CSR0$_{DCEA}$ = ‘00’, no invalidation is performed regardless of tag parity /EDC errors and no machine check is signaled. Uncorrectable tag EDC errors will cause clean unlocked lines to be invalidated when L1CSR0$_{DCEA}$ = ‘01’, and no machine check is signaled. All other lines are left unchanged.

11.8.2.5 dcbst instructions

For **dcbst** instructions, on a hit to any line without a tag EDC error, if the line is dirty, or has a dirty bit error, the line is flushed. Lock errors are ignored. When L1CSR0$_{DCEA}$ = ‘00’, tag EDC errors on other lines are ignored. When L1CSR0$_{DCEA}$ = ‘01’, uncorrectable tag EDC errors on other lines will also cause clean unlocked lines to be invalidated, regardless of hit or miss. No machine check is generated regardless of the setting of L1CSR0$_{DCEA}$. For **dcbst**, lock and dirty errors are ignored on a hit. Data parity errors will not prevent the line from being flushed, but will cause a machine check to be generated due to a push parity error.

For cacheable **dcbst** instructions that hit only to a line with a tag EDC error or that miss in all ways, a machine check will be generated if L1CSR0$_{DCEA}$ = ‘00’ and any line with a tag EDC error is dirty. Lock errors are ignored. If L1CSR0$_{DCEA}$ = ‘01’, clean unlocked lines with an uncorrectable tag EDC error are invalidated, and no errors are signaled unless any line with an uncorrectable tag EDC error is also dirty or has a dirty parity error. If any line with an uncorrectable tag EDC error is dirty, or has a dirty parity error, the line is not flushed and a machine check is generated, regardless of the settings of L1CSR0$_{DCEA}$.

11.8.2.6 dcbf instructions

For **dcbf** instructions, on a hit to any line without a tag EDC error, if the line is dirty, or has a dirty bit error, the line is flushed and invalidated. Lock errors are ignored. When L1CSR0$_{DCEA}$ = ‘00’, tag parity/EDC errors on other lines are ignored. When L1CSR0$_{DCEA}$ = ‘01’, uncorrectable tag EDC errors on other lines will also cause clean unlocked lines to be invalidated, regardless of hit or miss. No machine check is generated regardless of the setting of L1CSR0$_{DCEA}$. For **dcbf**, data parity errors will not prevent the line from being flushed, but will cause a machine check to be generated due to a push parity error.

For cacheable **dcbf** instructions that hit only to a line with a tag EDC error or that miss in all ways, a machine check will be generated if L1CSR0$_{DCEA}$ = ‘00’ and any line with a tag EDC error is dirty, locked, or has a dirty parity error or a lock parity error. If L1CSR0$_{DCEA}$ = ‘01’, clean unlocked lines with an uncorrectable tag EDC error are invalidated, and no errors are signaled unless any line with an uncorrectable tag EDC error is also dirty, locked, or has a dirty parity error or a lock parity error. If any line with an uncorrectable tag EDC error is dirty, or has a dirty parity error, the line is not flushed and a machine check is generated. If any line with an uncorrectable tag EDC error is locked, or has a lock parity error, the line is not invalidated, and a machine check is generated.

11.8.2.7 dcbz instructions

For **dcbz** instructions, on a hit to any line without a tag EDC error, the line is zeroed and set to dirty. Data errors, lock errors, and dirty errors are ignored. When L1CSR0$_{DCEA}$ = ‘00’, tag parity/EDC errors on other lines are ignored. When L1CSR0$_{DCEA}$ = ‘01’, uncorrectable tag EDC errors on other lines will also cause clean unlocked lines to be invalidated, regardless of hit or miss. No machine check is generated regardless of the setting of L1CSR0$_{DCEA}$. For **dcbz**, lock errors are ignored on a hit.

For cacheable **dcbz** instructions that hit only to a line with a tag EDC error or that miss in all ways, a machine check will be generated if L1CSR0$_{DCEA}$ = ‘00’ and any line has a tag parity/EDC or lock error. If L1CSR0$_{DCEA}$ = ‘01’ all line(s) with an uncorrectable tag EDC error are invalidated if clean. If a clean line that was locked or had a lock parity error was invalidated, a machine check is generated. If any line with an uncorrectable tag EDC error is dirty or has a dirty parity error, the line is not affected, and a machine check is generated, regardless of the settings of L1CSR0$_{DCEA}$. If a machine check is generated, no dcbz operation will be performed.

11.8.2.8 Cache locking instructions (dcbtls, dcbtstls, dcblc, icbtls, icblc)

For **dcbtls**, **dcbtstls**, **dcblc**, **icbtls**, and **icblc** instructions, on a hit to any line without a tag EDC error, the lock bits are set or cleared appropriately, and data, lock, and dirty bit parity or EDC errors are ignored. When L1CSR[0,1]$_{[D,I]CEA}$ = ‘00’, tag parity/EDC or lock errors on other lines are ignored. When L1CSR[0,1]$_{[D,I]CEA}$ = ‘01’, uncorrectable tag EDC errors on other lines will also cause clean unlocked lines to be invalidated, regardless of hit or miss. No machine check is generated regardless of the setting of L1CSR[0,1]$_{[D,I]CEA}$.

For cacheable **dcbtls**, **dcbtstls**, and **icbtls** instructions that hit only to a line with a tag EDC error or that miss in all ways, a machine check will be generated if L1CSR[0,1]$_{[D,I]CEA}$ = ‘00’ and any line has a tag parity/EDC error or a lock error. If L1CSR[0,1]$_{[D,I]CEA}$ = ‘01’, clean lines with an uncorrectable tag EDC error are invalidated and if a clean line that was locked or had a lock parity error was invalidated, a machine check is generated. If any line with an uncorrectable tag EDC error is dirty, or has a dirty parity error, the line is not affected and a machine check is generated, regardless of the settings of L1CSR[0,1]$_{[D,I]CEA}$.

For cacheable **dcblc** and **icblc** instructions that hit only to a line with a tag EDC error or that miss in all ways, a machine check will be generated if L1CSR[0,1]$_{[D,I]CEA}$ = ‘00’ and any line with a tag parity/EDC error is locked or has a lock parity error. If L1CSR[0,1]$_{[D,I]CEA}$ = ‘01’, lock and dirty parity errors will not cause a machine check on their own, but clean lines with an uncorrectable tag EDC error are invalidated, and if a clean line that was locked or had a lock parity error was invalidated, a machine check is generated. If any locked line with an uncorrectable tag EDC error is dirty, or has a dirty parity error, the line is not affected and a machine check is generated, regardless of the settings of L1CSR[0,1]$_{[D,I]CEA}$.

11.8.3 Cache inhibited accesses and parity/EDC errors

For non-cacheable access misses, no cache parity/EDC exceptions are signaled. When operating with correction/auto-invalidation disabled, tag EDC errors will cause misses for cache-inhibited accesses, and no machine check will be generated. When correction/auto-invalidation mode is enabled, a correction/auto-invalidation cycle will be run to correct/auto-invalidate tag, dirty, and lock errors, but invalidations will only be performed for uncorrectable tag errors on clean unlocked lines. If a cache-inhibited load or instruction fetch access hit occurs to a line with no tag EDC error, and the requested doubleword of data has no parity/EDC error, the access is treated as a cache hit and the CI status is ignored. Otherwise, if the requested doubleword of data has a parity/EDC error, the access is treated as a cache-inhibited cache miss and the cache data is ignored, even if dirty. No machine check will be generated in this case. A cache-inhibited store hit to a line with no tag EDC error will cause the data to be written to the cache, as well as to memory if the store is a writethrough store, and all data parity errors will be ignored. If a cache hit occurs to a line with an uncorrectable tag error, the hit is ignored, and the access is performed as a cache-inhibited cache miss and the cache data is ignored, even if dirty. No machine check will be generated in this case.

For cache control instructions such as **dcbf**, **dcbi**, **icbi**, and **dcbst** that are performed to addresses marked as cache-inhibited, no machine checks are generated, and the operations are only performed on/for lines that would not cause exceptions for the non-CI cases.

11.8.4 Snoop operations and parity/EDC errors

For snoop command lookups in which a hit occurs to a cache line with no tag EDC error, tag EDC errors in other lines are ignored, and no error condition is signaled.

Otherwise, for snoop command lookups in which a tag EDC error occurs and no hit occurs to a tag entry without a parity/EDC error, no correction attempt for the tags with errors will be made regardless of $L1CSR0_{DCEA}$, and the snoop response will indicate an error condition. When such a tag EDC error occurs on a snoop invalidate command, the invalidation will not occur, and the error will result in a machine check. The snoop queue will continue to be serviced, and the machine check will not necessarily be recoverable. A checkstop condition will not occur however. In this respect, it is treated similarly to a non-maskable interrupt, and the MSR[RI] bit should be used accordingly by software.

11.8.5 EDC checkbit/syndrome coding scheme generation — ICache

When operating with EDC enabled ($L1CSR1_{ICEDT}$ =01), double bit error detection codes are used to protect the tag and data portions of an instruction cache line. Each tag entry utilizes six check bits to cover the tag + valid bit, and each doubleword of data in the data arrays utilizes eight check bits. The specific coding schemes are shown in Table 11-7 and Table 11-8. The lock bits utilize bit-level redundancy, thus are independently protected.

Table 11-7 shows the checkbit coding for each tag entry. A '*' in the table indicates the bit is XOR'ed to form the final checkbit value.

Table 11-7. Tag checkbit generation

Checkbits p_tchk[0:5]	Tag bit																						
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	V
0	*	*	*	*	*	*			*	*	*		*	*		*							
1		*	*				*	*		*	*	*	*				*	*				*	*
2				*	*			*	*	*					*	*	*	*	*		*		*
3				*		*					*	*	*	*	*	*	*			*	*	*	*
4	*	*		*	*		*	*			*			*					*	*	*	*	*
5	*		*	*		*	*		*		*	*			*			*	*	*			*

Table 11-8 shows the checkbit coding for each doubleword data entry. A '*' in the table indicates the bit is XOR'ed to form the final checkbit value.

Table 11-8. Data checkbit generation

Checkbits p_dchk[0:7]	Data bit																															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	*	*	*	*	*	*	*	*			*			*	*			*			*			*	*			*				
1	*	*	*						*	*	*	*	*	*	*	*			*			*	*			*			*			*
2				*	*	*			*	*	*						*	*	*	*	*	*	*	*			*			*	*	
3				*			*	*				*	*	*			*	*	*						*	*	*	*	*	*	*	*
4				*								*			*	*				*	*	*			*	*	*					
5	*			*								*								*			*	*				*	*	*		
6		*			*			*	*			*								*								*			*	*
7			*			*	*			*			*			*	*			*								*				

Checkbit	Data bit																															
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
0				*								*			*	*				*	*	*			*	*	*					
1	*			*								*								*			*	*				*	*	*		
2		*			*			*	*			*								*								*			*	*
3			*			*	*			*			*			*	*			*								*				
4	*	*	*	*	*	*	*	*			*			*	*			*			*			*	*			*				
5	*	*	*						*	*	*	*	*	*	*	*			*			*	*			*			*			*
6				*	*	*			*	*	*						*	*	*	*	*	*	*	*			*			*	*	
7				*			*	*				*	*	*			*	*	*						*	*	*	*	*	*	*	*

11.8.6 EDC checkbit/syndrome coding scheme generation — DCache

When operating with EDC enabled (L1CSR0$_{DCEDT}$ =01), double bit error detection codes are used to protect the tag portion of a data cache line. The data array continues to utilize single-bit parity protection. Each data cache tag entry utilizes six check bits to cover the tag + valid bit. The specific coding scheme for the tag array is the same as is used for the ICache, and is shown in Table 11-7. The dirty and lock bits utilize bit-level redundancy, thus are independently protected. Three dirty bits are provided to support single-bit and double-bit error detection. Correction is performed by setting the dirty bits to '1' if a dirt parity error occurs and autoinvalidation/correction is enabled. Four lock bits are provided to support single-bit error correction and double-bit error detection.

11.8.7 Cache error injection

Cache error injection provides a way to test error recovery by intentionally injecting parity errors into the instruction and/or data cache.

Error injection into the instruction cache operates as follows:

- If L1CSR1$_{ICEI}$ is set and L1CSR1$_{ICEDT}$=01, any instruction cache line fill to the instruction cache data has the associated two most significant parity check bits inverted in the instruction cache data array for each doubleword loaded.

Error injection for the data cache operates as follows:

- If L1CSR0$_{DCEI}$ is set, any cache line fill to the data cache data array has all of the associated parity bits inverted in the data array for each doubleword loaded. Additionally, inverted parity bits are generated for any bytes stored into the data cache data array on a store hit.

Cache parity error injection is not performed for cache debug write accesses, since parity bit values written can be directly controlled (See Section 11.19.3, Cache Debug Access Control register (CDACNTL)).

In order to clear the parity errors, a cache invalidation or an invalidation of the lines that could have had an injected parity error may be performed. Line invalidation may be performed by an **icbi/dcbi** instruction, or an L1FINV[0,1] invalidation operation.

11.9 Push and store buffers

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

If a data cache miss displaces a dirty line, the linefill request is forwarded to the external bus. While waiting for the response, the current contents of the dirty cache line are placed into the push buffer. Once the linefill transaction (burst read) completes, the cache controller can generate the appropriate burst write bus transaction to write the contents of the push buffer into memory.

The store buffer contains a FIFO that can defer pending write misses or writes marked as write-through in order to maximize performance. The store buffer can buffer as many as eight words (32 bytes) for this purpose. The store buffer may be disabled for debug purposes. Operation of the store buffer is independent of the L1CSR0[DCE] bit. When the store buffer is enabled, non-allocating store operations that miss the cache or that are marked as writethrough are placed in the store buffer, and the CPU access is terminated. Each store buffer entry contains 32-bits of physical address, 32-bits of data, size information, and 3 bits of access attribute information (W, G, and S/U) in order to properly drive the **attribute** output signals on a buffered store access. Cache-inhibited guarded stores are not buffered however, and are delayed from being performed until the push and store buffers have been emptied.

Once the push or store buffer has valid data, the internal bus controller uses the next available external bus cycle to generate the appropriate write cycles. In the event that another data cache fill is required (e.g., cache load or store w/allocate miss to process) during the continued instruction execution by the processor pipeline, an alias check is performed between the linefill address and all valid entries in the store and push buffer using the index portion of the access address. If no match is found, the linefill may bypass pending stores in the store or push buffer. Otherwise, if an alias exists (index matches any valid store buffer entry), the data cache pipeline will stall until the aliased entries have been flushed from the store and push buffer before generating the required external bus transaction for the linefill.

Single-beat read transactions will not bypass pending stores in the push or store buffer.

The push buffer is always emptied prior to queued store buffer entries to avoid memory consistency issues. Once the push buffer has been loaded with dirty data to be written back to memory, a subsequent store may be buffered, but will not be written to memory until the push has completed.

For cache-inhibited load accesses or cache-inhibited guarded store accesses, the processor termination is withheld until the store buffer has been flushed of all entries, the push buffer has been emptied, and the access has completed to memory.

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

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

External termination errors that occur on any push of a dirty cache line will result in a machine check condition.

11.10 Cache management instructions

This section describes the implementation of Cache Management instructions in e200z759n3.

11.10.1 Instruction cache block invalidate (icbi) instruction

- **icbi** is described on page 280 of *Book E: Enhanced PowerPC^{tm} Architecture v0.99*
- If the cache line containing the byte addressed by the EA associated with this instruction is present in the instruction cache, it is invalidated, regardless of lock status. If an instruction cache linefill is in progress and the linefill data corresponds to the EA associated with a **icbi**, the instruction cache is not updated with linefill data.

11.10.2 Instruction cache block touch (icbt) instruction

- **icbt** is described on page 281 of *Book E: Enhanced PowerPC^{tm} Architecture v0.99*
- If HID0$_{NOPTI}$ is set, this instruction is treated as a no-op.

11.10.3 Data cache block allocate (dcba) instruction

- **dcba** is described on page 241 of *Book E: Enhanced PowerPC^{tm} Architecture v0.99*
- This instruction is treated as a no-op.

11.10.4 Data cache block flush (dcbf) instruction

- **dcbf** is described on page 242 of *Book E: Enhanced PowerPC^{tm} Architecture v0.99*
- If the cache line containing the byte addressed by the EA associated with this instruction is present in the data cache, it is copied back to memory if dirty. The line is subsequently invalidated regardless of whether it was copied back or locked. If a data cache linefill is in progress and the linefill data corresponds to the EA associated with a **dcbf**, the data cache is not updated with linefill data.
- This instruction is treated as a load for the purposes of access protection.
- If the data cache is disabled, this instruction is treated as a no-op.

11.10.5 Data cache block invalidate (dcbi) instruction

- **dcbi** is described on page 243 of *Book E: Enhanced PowerPC^{tm} Architecture v0.99*
- If the cache line containing the byte addressed by the EA associated with this instruction is present in the data cache, it is invalidated, regardless of lock status. No copyback occurs if the line is present in the data cache and dirty. If a data cache linefill is in progress and the linefill data corresponds to the EA associated with a **dcbi**, the data cache is not updated with linefill data.
- This instruction is privileged
- This instruction is treated as a store for the purposes of access protection.
- If the data cache is disabled, this instruction is treated as a no-op in supervisor mode.

11.10.6 Data cache block store (dcbst) instruction

- **dcbst** is described on page 245 of *Book E: Enhanced PowerPC^{tm} Architecture v0.99*
- If the cache line containing the byte addressed by the EA associated with this instruction is present in the data cache, it is copied back to memory if dirty. The line is subsequently marked clean, and the lock status is unchanged
- This instruction is treated as a load for the purposes of access protection.
- If the data cache is disabled, this instruction is treated as a no-op.

11.10.7 Data cache block touch (dcbt) instruction

- **dcbt** is described on page 246 of *Book E: Enhanced PowerPC^{tm} Architecture v0.99*
- If $\text{HID0}_{\text{NOPTI}}$ is set, this instruction is treated as a no-op.

11.10.8 Data cache block touch for store (dcbtst) instruction

- **dcbtst** is described on page 247 of *Book E: Enhanced PowerPC^{tm} Architecture v0.99*
- If $\text{HID0}_{\text{NOPTI}}$ is set, this instruction is treated as a no-op.

11.10.9 Data cache block set to zero (dcbz) instruction

- **dcbz** is described on page 248 of *Book E: Enhanced PowerPC^{tm} Architecture v0.99*

- If the cache line containing the byte addressed by the EA associated with this instruction is present in the data cache, all bytes in the line are zeroed, the line is marked as modified, and remains valid. Lock status remains unchanged. If the cache line is not present and the address is cacheable, it is established in the data cache (without fetching from memory), all bytes in the line are zeroed, and the line is marked as modified and valid.
- This instruction is treated as a store for the purposes of access protection.
- **dcbz** causes an Alignment exception if the EA is marked by the MMU as Cache-inhibited and a data cache miss occurs, or if the EA is marked by the MMU as Writethrough Required, or if the data cache is disabled or is operating in writethrough mode, or if an overlocking condition prevents the allocation of a line into the data cache.

11.11 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, e200z759n3 provides the $\mathrm{HID0}_{\mathrm{NOPTI}}$ control bit to cause these instructions to be treated as nops.

11.12 Cache line locking/unlocking APU

11.12.1 Overview

e200z759n3 supports the *Freescale EIS* 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**, **dcbtstls**) and two instructions are for instruction cache locking control (**icblc**, **icbtls**).

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

The user-mode cache lock enable MSR[UCLE] bit may be used to restrict user-mode cache line locking. If MSR[UCLE] is clear, any cache lock instruction executed in user-mode will take a Cache-locking DSI exception (unless nop'ed) and set either ESR[DLK] or ESR[ILK]. If MSR[UCLE] is set, cache-locking instructions can be executed in user-mode and they will not take a DSI for cache-locking. However, they may still cause a DSI for access violations or cause machine checks for external termination errors.

There are cases when attempting to set a lock will fail even when no DSI or DTLB exceptions occur. These are as follows:

- The target address is marked cache-inhibited and a cache miss occurs
- The cache is disabled or all ways of the cache are disabled for replacement
- The cache target indicated by the CT field (bits 7-10) of the instruction is not 0

In these cases, the lock set instruction is treated as a NOP, and the cache unable to lock L1CSR{0,1}[CUL] bit is set.

Assuming no exception conditions occur (DSI or DTLB error), for **dcbtls**, **dcbtstls**, and **icbtls** an attempt is made to lock the corresponding cache line. If a miss occurs, and all of the available ways (ways enabled for a particular access type) are already locked in a given cache set, an attempt to lock another line in the same set will result in an overlocking situation. In this case, the cache overlock bit L1CSR{0,1}[CLO] is set to indicate that an overlocking situation occurred. This does not cause an exception condition. The new line is conditionally placed in the cache, displacing a previously locked line depending on the setting of the appropriate L1CSR0,1[CLOA] bit.

The CUL conditions have priority over the CLO condition.

If multiple NOP or exception conditions arise on a cache lock instruction, the results are determined by the order of precedence described in Table 11-9.

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

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

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

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

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

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

11.12.2 dcbtls — data cache block touch and lock set

dcbtls **dcbtls**

Data Cache Block Touch and Lock Set

dcbtls CT, RA, RB (E=0) Form X

31	/	CT	RA	RB	0 0 1 0 1 0 0 1 1 0	/
0 5	6	10	11 15	16 20	21 30	31

Figure 11-10. dcbtls — data cache block touch and lock set

Description:

```
if RA=0 then a ← $^{64}0$ else a ← GPR(RA)
EA ← $^{32}0$ || (a + GPR(RB))$_{32:63}$
PrefetchDataCacheBlockLockSet(CT, EA)
```

If CT=0, the cache line corresponding to EA is loaded and locked into the level 1 data cache.

If CT=0 and the line already exists in the data cache, **dcbtls** locks the line without refetching it from external memory.

Exceptions:

If the MSR[UCLE] (user-mode cache lock enable) bit is set, **dcbtls** may be performed while in user mode (MSR[PR]=1). If the MSR[UCLE] bit is clear, an attempt to perform these instructions in user mode causes a data cache locking error DSI unless the CT field or other conditions otherwise NOP the instruction.

The e200z759n3 only supports CT=0. If CT is some value other than 0, the **dcbtls** is NOP'ed and the L1CSR0[DCUL] bit is set indicating an unable-to-lock condition occurred. No other exceptions are reported. If the data cache is disabled, the **dcbtls** is NOP'ed and the L1CSR0[DCUL] bit is set indicating an unable-to-lock condition occurred. No other exceptions are reported.

The **dcbtls** instruction is treated as a load with respect to translation and will cause a DSI interrupt for access violations, as well as causing a Data TLB error interrupt if the target address cannot be translated.

If the block corresponding to EA is cache-inhibited and a data cache miss occurs, the instruction is NOP'ed, (no DSI is taken due to the cache-inhibited status), and the L1CSR0[DCUL] bit is set indicating an unable-to-lock condition occurred.

Other registers altered:

- L1CSR0 (see below)

When a **dcbtls** is performed to an index, and a way can not be locked, the L1CSR0[DCUL] bit is set indicating an unable-to-lock condition occurred. This also occurs whenever the **dcbtls** must be NOP'ed.

When a **dcbtls** is performed to an index in the data cache that already has all the ways locked, this is referred to as an over-locking situation. There is no exception generated by an over-locking situation. Instead the L1CSR0[DCLO] bit is set, indicating an over-lock condition occurred. A line is allocated and locked in the cache depending on the setting of the L1CSR0[DCLOA] control bit. If system software wants to precisely determine if an overlock condition has happened, it must perform the following code sequence:

```
dcbtls
msync
mfspr (L1CSR0)
       (check L1CSR0[DCUL] bit for cache index unable-to-lock condition)
       (check L1CSR0[DCLO] bit for cache index over-lock condition)
```

11.12.3 dcbtstls — data cache block touch for store and lock set

dcbtstls dcbtstls

Data Cache Block Touch for Store and Lock Set

dcbtstls CT, RA, RB (E=0) Form X

31	/	CT	RA	RB	0 0 1 0 0 0 0 1 1 0	/
0 5	6	10 11	15 16	20 21	30	31

Figure 11-11. dcbtstls — data cache block touch for store and lock set

Description:

```
if RA=0 then a ← $^{64}$0 else a ← GPR(RA)
EA ← $^{32}$0 || (a + GPR(RB))$_{32:63}$
PrefetchDataCacheBlockLockSet(CT, EA)
```

e200z759n3 treats the **dcbtstls** instruction identically to the **dcbtls** instruction since no hardware coherency mechanisms are implemented for the cache.

11.12.4 dcblc — data cache block lock clear

dcblc dcblc

Data Cache Block Lock Clear

dcblc CT, RA, RB (E=0) Form X

31	/	CT	RA	RB	0 1 1 0 0 0 0 1 1 0	/
0 5	6	10 11	15 16	20 21	30	31

Figure 11-12. dcblc — data cache block lock clear

Description:

```
if RA=0 then a ← $^{64}$0 else a ← GPR(RA)
```

```
EA ← $^{32}0$ || (a + GPR(RB))$_{32:63}$
    DataCacheClearLockBit(CT, EA)
```

If CT=0, and the line is present in the L1 data cache, the lock bit for that line is cleared, making that line eligible for replacement.

Exceptions:

If the MSR[UCLE] (user-mode cache lock enable) bit is set, **dcblc** may be performed while in user mode (MSR[PR]=1). If the MSR[UCLE] bit is clear, an attempt to perform this instructions in user mode causes a DSI, unless the CT field or other conditions otherwise NOP the instruction.

The e200z759n3 only supports CT=0. If CT is some value other than 0, the **dcblc** is NOP'ed. No other exceptions are reported. If the data cache is disabled, the **dcblc** is NOP'ed. No other exceptions are reported.

The **dcblc** instruction is treated as a load with respect to translation and will cause a DSI interrupt for access violations, as well as causing a Data TLB error interrupt if the target address cannot be translated.

11.12.5 icbtls — instruction cache block touch and lock set

icbtls icbtls

Instruction Cache Block Touch and Lock Set

icbtls CT, RA, RB (E=0) Form X

31	/	CT	RA	RB	0 1 1 1 1 0 0 1 1 0	/
0 5	6	10 11	15 16	20 21	30	31

Figure 11-13. icbtls — instruction cache block touch and lock set

Description:

```
if RA=0 then a ← $^{64}0$ else a ← GPR(RA)
EA ← $^{32}0$ || (a + GPR(RB))$_{32:63}$
    PrefetchInstructionCacheBlockLockSet(CT, EA)
```

If CT=0, the cache line corresponding to EA is loaded and locked into the level 1 instruction cache.

If CT=0 and the line already exists in the instruction cache, **icbtls** locks the line without refetching it from external memory.

Exceptions:

If the MSR[UCLE] (user-mode cache lock enable) bit is set, **icbtls** may be performed while in user mode (MSR[PR]=1). If the MSR[UCLE] bit is clear, an attempt to perform these instructions in user mode causes an Instruction cache locking error DSI unless the CT field or other conditions otherwise NOP the instruction.

The e200z759n3 only supports CT=0. If CT is some value other than 0, the **icbtls** is NOP'ed and the L1CSR1[ICUL] bit is set indicating an unable-to-lock condition occurred. No other exceptions are reported. If the instruction cache is disabled, the **icbtls** is NOP'ed and the L1CSR1[ICUL] bit is set indicating an unable-to-lock condition occurred. No other exceptions are reported.

The **icbtls** instruction requires either execute or read (X or R) permissions with respect to translation and will cause a DSI interrupt for access violations, as well as causing a Data TLB error interrupt if the target address cannot be translated.

If the block corresponding to EA is cache-inhibited and an instruction cache miss occurs, the instruction is NOP'ed, (no DSI is taken due to the cache-inhibited status), and the L1CSR1[ICUL] bit is set indicating an unable-to-lock condition occurred.

Other registers altered:

- L1CSR1 (see below)

When **icbtls** is performed to an index and a way can not be locked, the L1CSR1[ICUL] bit is set indicating an unable-to-lock condition occurred. This also occurs whenever **icbtls** must be NOP'ed.

When **icbtls** is performed to an index in the instruction cache that already has all the ways locked, this is referred to as an over-locking situation. There is no exception generated by an over-locking situation. Instead the L1CSR1[ICLO] bit is set, indicating an over-lock condition occurred. A line is allocated and locked in the cache depending on the setting of the L1CSR1[ICLOA] control bit. If system software wants to precisely determine if an overlock condition has happened, it must perform the following code sequence:

```
icbtls
msync
mfspr (L1CSR1)
       (check L1CSR1[ICUL] bit for cache index unable-to-lock condition)
       (check L1CSR1[ICLO] bit for cache index over-lock condition)
```

11.12.6 icblc — instruction cache block lock clear

icblc

Instruction Cache Block Lock Clear

icblc CT, RA, RB (E=0) Form X

31	/	CT	RA	RB	0 0 1 1 1 0 0 1 1 0	/
0 5	6	10	11 15	16 20	21 30	31

Figure 11-14. icblc — instruction cache block lock clear

Description:

```
if RA=0 then a ← $^{64}0$ else a ← GPR(RA)
EA ← $^{32}0$ || (a + GPR(RB))$_{32:63}$
     InstCacheClearLockBit(CT, EA)
```

If CT=0, and the line is present in the instruction cache, the lock bit for that line is cleared, making that line eligible for replacement.

Exceptions:

If the MSR[UCLE] (user-mode cache lock enable) bit is set, **icblc** may be performed while in user mode (MSR[PR]=1). If the MSR[UCLE] bit is clear, an attempt to perform these instructions in user mode causes an Instruction cache locking error DSI unless the CT field or other conditions otherwise NOP the instruction.

The e200z759n3 only supports CT=0. If CT is some value other than 0, the **icblc** is NOP'ed. No other exceptions are reported. If the instruction cache is disabled, the **icblc** is NOP'ed. No other exceptions are reported.

The **icblc** instruction requires either execute or read (X or R) permissions with respect to translation and will cause a DSI interrupt for access violations, as well as causing a Data TLB error interrupt if the target address cannot be translated.

11.12.7 Effects of other cache instructions on locked lines

The following cache instructions have no effect on the state of a cache line's lock bit: **icbt**, **dcba**, **dcbz**, **dcbst**, **dcbt,** and **dcbtst**.

The following cache instructions flush/invalidate and unlock a cache line in the respective L1 caches: **dcbf, dcbi,** and **icbi**.

11.12.8 Flash clearing of lock bits

e200z759n3 supports flash clearing of cache lock bits under software control by using the CFCL (cache flash clear locks) control bit in the L1CSR{0,1} register.

Lock bits are not cleared automatically upon power-up (**m_por**) or normal reset (**p_reset_b**). Software must use the CLFC control bit to clear the lock bits after a reset. Proper use of this bit is to determine that it is clear and then set it with a pair of **mfspr mtspr** operations. A 0-to-1 transition on CLFC causes a flash clearing of the lock bits to be initiated, which lasts for multiple (approx. 134) CPU cycles. Once set, the CLFC bit will be cleared by hardware after the operation is complete. It will remain set during the clearing interval, and may be tested by software to determine when the operation has completed. A **mtspr** operation to L1CSR{0,1} that attempts to change the state of L1CSR{0,1}[CLFC] during invalidation will not affect the state of that bit.

During the process of performing the flash clearing, the cache does not respond to accesses, and remains busy. Interrupts may still be recognized and processed, potentially aborting the flash clearing operation. When this occurs, the L1CSR{0,1}[ABT] bit will be set to indicate unsuccessful completion of the operation. Software should read the L1CSR{0,1} register to determine that the operation has completed (L1CSR{0,1}[CLFC] bit cleared), and then check the status of the L1CSR{0,1}[ABT] bit to determine completion status.

NOTE

Note that while most implementations of the e200z759n3 will stall further instruction execution during this flash clearing interval, it is not guaranteed across all implementations, thus software should be written using these guidelines.

11.13 Cache instructions and exceptions

All cache management instructions (except **icbt**, **dcba**, **dcbt,** and **dcbtst**) can generate TLB miss exceptions if the effective address cannot be translated, or may generate DSI exceptions due to permission violations. In addition, **dcbz** may generate an Alignment interrupt as described in Section 11.10.9, Data cache block set to zero (dcbz) instruction.

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

11.13.1 Exception conditions for cache instructions

If multiple NOP or exception conditions arise on a cache instruction, the results are determined by the order of precedence described in Table 11-9.

Table 11-9. Special case handling

Operation	CT!=0	Cache disabled	TLB miss	User & UCLE=0	Protection Violation	WT or cache in write-through mode	Cache parity error	CI and miss in cache	All available ways locked	External termination error
icbt, dcbt, dcbtst	NOP	NOP	NOP	—	NOP	—	NOP	NOP	NOP	NOP
dcbtls dcbtstls dcblc	DCUL DCUL NOP	DCUL DCUL NOP	DTLB DTLB DTLB	DLK DLK DLK	DSI DSI DSI	— — —	MC MC MC	DCUL DCUL —	DCLO DCLO —	MC MC —
icbtls icblc	ICUL NOP	ICUL NOP	DTLB DTLB	ILK ILK	DSI DSI	— —	MC MC	ICUL —	ICLO —	MC —
dcbz	—	ALI	DTLB	—	DSI	ALI	MC	ALI	ALI	—
dcbf, dcbst	—	NOP	DTLB	—	DSI	—	MC	—	—	MC
icbi, dcbi	—	NOP	DTLB	—	DSI	—	—	—	—	—

Table 11-9. Special case handling (continued)

Operation	CT!= 0	Cache disabled	TLB miss	User & UCLE= 0	Protection Violation	WT or cache in write-through mode	Cache parity error	CI and miss in cache	All available ways locked	External termination error
Atomic load or store.	— —	— —	DTLB DTLB	— —	DSI DSI	— —	MC MC	— —	— —	MC MC
load store	— —	— —	DTLB DTLB	— —	DSI DSI	— —	MC MC	— —	— —	MC MC

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

11.13.2 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 11-10 shows the definitions of the **p_d_ttype[0:5]** encodings.

Table 11-10. Transfer type encoding

p_d_ttype[0:5][1]	Transfer type	Instruction
00000e	Normal	normal loads / stores
000010	Atomic	**lwarx**, **stwcx.**, **lharx**, **sthcx.**, **lbarx**, **stbcx.**
00010e	Flush Data Block	dcbst
00011e	Flush and Invalidate Data Block	dcbf
00100e	Allocate and Zero Data Block	dcbz
001010	Invalidate Data Block	dcbi
00110e	Invalidate Instruction Block	icbi
001110	multiple word load/store	lmw, stmw
010000	TLB Invalidate	tlbivax
010010	TLB Search	tlbsx
010100	TLB Read entry	tlbre
010110	TLB Write entry	tlbwe

Table 11-10. Transfer type encoding (continued)

p_d_ttype[0:5][1]	Transfer type	Instruction
011000	Touch for Instruction	icbt
011010	Lock Clear for Instruction	icblc
011100	Touch for Instruction and Lock Set	icbtls
011110	Lock Clear for Data	dcblc
10000e	Touch for Data	dcbt
10001e	Touch for Data Store	dcbtst
100100	Touch for Data and Lock Set	dcbtls
100110	Touch for Data Store and Lock Set	dcbtstls

1 p_ttype[5] ‘e’ is set to set to 0.

11.14 Sequential consistency

The Power Architecture 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 e200z759n3 CPU achieves this effect by operating a single pipeline to the Cache/MMU. All memory accesses are presented to the MMU in the exact order that they appear in the program and therefore exceptions are determined in order.

11.15 Self-modifying code requirements

The following sequence of instructions will synchronize the instruction stream.

1. dcbf
2. icbi
3. msync
4. isync

This sequence ensures that the operation is correct for *PowerISA 2.06* processors that implement separate instruction and data caches, as well as for multi-processor cache-coherent systems.

11.16 Page table control bits

The Power Architecture architecture allows certain memory characteristics to be set on a page and on a block basis. These characteristics include writethrough (using the W-bit), cacheability (using the I-bit), coherency (using the M-bit), guarded memory (using the G-bit), and endianness (using the E-bit). Incorrect use of these bits may create situations where coherency paradoxes are observed by the processor. In particular, this can happen when the state of these bits are changed without appropriate precautions being taken (that is, flushing the pages that correspond to the changed bits from the cache), or when the address translations of aliased real addresses specify different values for any of the WIMGE bits.

Generally, certain mixing of WIMG settings are allowed by the Book E Power Architecture architecture, however others may present cache coherence paradoxes and are considered programming errors.

11.16.1 Writethrough stores

A writethrough store (WIMGE = b'1xxxx') may normally hit to a valid cache line. In this case, the cache line remains in its current state, the store data is written into the cache, and the store goes out on the bus as a single beat write.

11.16.2 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.

11.16.3 Memory coherence required

For the e200z759n3, the "memory coherence required" storage attribute (WIMGE = b'xx1xx') is reflected on the **p_d_gbl** output during each external data access, to indicate to external coherency logic that memory coherence is required. This bit is ignored for instruction accesses.

11.16.4 Guarded storage

For the e200z759n3, the guarded storage attribute (WIMGE = b'xxx1x') is used to determine if a second outstanding data cache miss may proceed to the system interface prior to the termination of the first outstanding miss. If the second address is marked as guarded, it will not be presented to the external interface until the previous miss has been completed without error.

11.16.5 Misaligned accesses and the endian (E) bit

Misaligned load or store accesses that cross page boundaries can cause data corruption if the two pages do not have the same endianness (that is, one page is big endian while the other page is little endian). If this occurs, the processor would not get all the bytes, or would get some of them out of order, resulting in garbled data. To protect against data corruption, the e200z759n3 core takes a DSI exception and set the BO (byte ordering) bit in the Exception Syndrome register whenever this situation occurs.

11.17 Reservation instructions and cache interactions

The e200z759n3 core treats reservation instruction (***lbarx, lharx, lwarx, stbcx., sthcx.***, and **stwcx.**) accesses as though they were cache inhibited, regardless of page attributes. Additionally, a cache line corresponding to the address of a reservation instruction access will be flushed to memory if dirty, prior to the reservation access being issued to the bus. This is done to allow external reservation logic to be built that properly signals a reservation failure. The bus access will be treated as a single-beat transfer.

11.18 Effect of hardware debug on cache operation

Hardware debug facilities utilize normal CPU instructions to access register and memory contents during a *debug session*. This may have the unavoidable side-effect of causing the store and push buffers to be flushed. During hardware debug, the MMU page attributes are controllable by the debug firmware via settings of the OnCE Control register (OCR). Refer to Section 12.4.6.3, e200z759n3 OnCE Control Register (OCR).

Cache snoop operations continue to be serviced during debug sessions.

11.19 Cache memory access for debug / error handling

The cache memory provides resources needed to do foreground accesses via **mtdcr** instructions executed by the processor, or background accesses through the JTAG/OnCE port to read and write the cache SRAM arrays. Accesses are supported via a pair of device control registers (DCRs) that are also mapped into OnCE-accessible registers. These resources are intended for use by special debug tools and by debug or specialized error recovery exception software, not by general application code.

Access to the cache memory SRAM arrays using **mtdcr** instructions may be performed by supervisor-level software after appropriate synchronization has been performed with **msync**, **isync** instruction pairs. Access to the cache memory SRAM arrays using the JTAG port is conditional on the CPU being in debug mode. The CPU must be placed in debug state prior to initiation of a read or write access via OnCE.

This facility allows access only to the SRAM arrays used for cache tag and data storage. This function is available even when the cache is disabled. The cache linefill buffer, push buffer, store buffer, and late write buffer are all outside of the SRAM arrays and are not accessible. However, before a debug memory access request is serviced, the push and store buffers will be written to external memory, and the late write and linefill buffers will be written to the cache arrays.

11.19.1 Cache memory access via software

Cache debug access control and data information are accessed by executing **mfdcr** and **mtdcr** instructions to the Cache Debug Access control and data registers CDACNTL and CDADATA (see Table 11-11 and Table 11-12). Accesses are performed one word (32 bits) at a time.

For a Cache write access, software must first write the CDADATA register with the desired tag and status flags, or data values. The second step is to write the CDACNTL register with desired tag or data location and parity values, and assert the R/W and GO bits in CDACNTL.

Note that writing a 64-bit value for data requires two passes, one for the even word (A29=0) and one for the odd word (A29=1). Each 32-bit write will update all of the parity/check bits, so in general, if only a single 32-bit write is performed, it should be preceded by a read of the data that is not being modified, in order to properly compute or store all 8 parity/check bits when the modified 32-bit data is written. Tag writes are accomplished in a single pass.

For a Cache read access, software must first access and write the CDACNTL register with desired tag or data location, and assert the R/W and GO bits in CDACNTL. The second step is to read the CDADATA register for the tag or data and read the CDACNTL register for parity information.

Completion of any operation can be determined by reading the CDACNTL register. Operations are indicated as complete when CDACNTL[30:31] = ‘00’. Software should poll the CDACNTL register to determine when an access has been completed prior to assuming validity of any other information in the CDACNTL or CDADATA registers.

Note that no parity errors are generated as a result of **mtdcr**/**mfdcr** instructions involving the CDACNTL or CDADATA registers.

To ensure proper cache write operation, the following program sequence is recommended:

```
         msync
         isync
         mtdcr cdadata, rS1 // set up write data
         mtdcr cdacntl, rS2 // write control to initiate write
         msync
         isync
loop:    mfdcr rN, cdacntl // check for done
         andi. rT, rN, #3
         bne loop
         .
         .
```

To ensure proper cache read operation, the following program sequence is recommended:

```
         msync
         isync
         mtdcr cdacntl, rS2 // write control to initiate read
         msync
         isync
loop:    mfdcr rN, cdacntl // check for done
         andi. rT, rN, #3
         bne loop
         mfdcr rT, cdadata // return data
         .
         .
```

Conflict conditions with snoop accesses to the same cache line cannot be resolved in a manner that guarantees that a value read will not change state before a subsequent value written. No interlocking is performed, so a cache entry read as being valid or written to a valid state may become invalid at any time.

11.19.2 Cache memory access through JTAG/OnCE port

Cache debug access control and data information are serially accessed through the OnCE controller and access the Cache Debug Access control and data registers CDACNTL and CDADATA (see Table 11-11 and Table 11-12). Accesses are performed one word (32 bits) at a time.

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

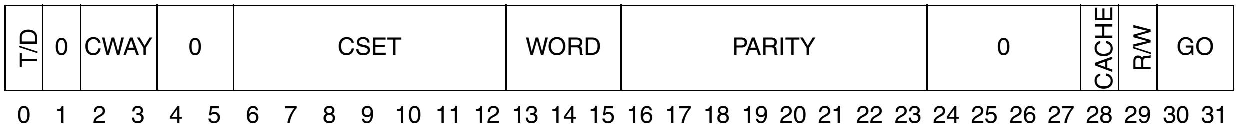
For a Cache read access, the user must first access and write the CDACNTL register with desired tag or data location, and assert the R/W and GO bits in CDACNTL. The second step is to access and read the CDADATA register for the tag or data and read the CDACNTL register for parity.

Completion of any operation can be determined by reading the CDACNTL register. Operations are indicated as complete when CDACNTL[30:31] = ‘00’. Debug firmware should poll the CDACNTL register to determine when an access has been completed prior to assuming validity of any other information in the CDACNTL or CDADATA registers.

Conflict conditions with snoop accesses to the same cache line cannot be resolved in a manner that guarantees that a value read will not change state before a subsequent value written. No interlocking is performed, so a cache entry read as being valid or written to a valid state may become invalid at any time.

11.19.3 Cache Debug Access Control register (CDACNTL)

The Cache Debug Access Control Register (CDACNTL) contains location information (T/D, CWAY, CSET, and WORD), and control (R/W and GO) needed to access the Cache Tag or Data SRAM arrays. Also included here are the SRAM parity bit values that must be supplied by the user for write accesses, and that will be supplied by the cache for read accesses. The CDACNTL register is shown in Figure 11-15.



DCR - 351; Read/Write; Reset - 0x0

Figure 11-15. Cache Debug Access Control register (CDACNTL)

Table 11-11 provides bit definitions for the Cache Debug Access Control Register.

Table 11-11. CDACNTL field descriptions

Bit	Name	Description
0	T/D	Tag / Data 0 Data array selected 1 Tag array selected
1	—	Reserved[1]
2:3	CWAY	Cache Way Specifies the cache way to be selected
4:5	—	Reserved[1]
6:12	CSET	Cache Set: Specifies the cache set to be selected
13:15	WORD	Word (Data array access only, I or D cache) Specifies one of eight words of selected set

Table 11-11. CDACNTL field descriptions (continued)

Bit	Name	Description
16:23	PARITY / EDC CHECK BITS	Parity check bits[2] (I or D cache) EDC Mode (L1CSR[0,1]$_{[D,I]}$CEDT = 01): DCache Data array: Byte parity bits. One bit per data byte. bit 16: Parity for byte 0, bit 17: Parity for byte 1.... bit 23: Parity for byte 7. ICache Data Array: parity check bits for data. Bits 16:23 correspond to p_dchk[0:7] (See Table 11-8). Tag Array: parity check bits for tag. Bits 16:21 correspond to p_tchk[0:5] (See Table 11-7). bits 22:23 reserved.
24:27	—	Reserved[1]
28	CACHE	Cache Select Specifies the cache to be selected 0 Selects the data cache for the operation. 1 Selects the instruction cache for the operation.
29	R/W	Read / Write: 0 Selects write operation. Write the data in the CDADATA register to the location specified by this CDACNTL register. 1 Selects read operation. Read the cache memory location specified by this CDACNTL register and store the resulting data in the CDADATA register and store the parity bits in this CDACNTL register.
30:31	GO	GO command bits 00 Inactive or complete (no action taken) hardware sets GO=00 when an operation is complete 01 Read or write cache memory location specified by this CDACNTL register. 1x Reserved

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

[2] Cache parity checkers assume odd parity when using parity protection. EDC coding is used otherwise.

11.19.3.1 Cache Debug Access Data register (CDADATA)

The Cache Debug Access Data Register (CDADATA) contains the SRAM data for a debug access. The same register is used for Tag and Data SRAM read and write operations for both caches. Note that a single 32-bit word is accessed. Accessing an entire 64-bit doubleword requires two passes. The CDADATA register is shown in Figure 11-16.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
TAG or DATA																															

DCR - 350; Read/Write; Reset - Undefined/Unaffected

Figure 11-16. Cache Debug Access Data register (CDADATA)

Table 11-12 provides bit definitions for the Cache Debug Access Data Register.

Table 11-12. CDADATA field descriptions

Bit(s)	Name	Description
0:31	TAG	TAG Array Access Data - when accessing the tag array of either cache: 0:21 Tag compare bits 22 Reserved 23 Valid bit 24:27 Lock bits. These four bits should have the same value, 1-Locked, 0-Unlocked. 28:30 Dirty bits - (data cache only). These three bits should have the same value, 1-Dirty, 0-Clean.
	DATA	DATA Array Access Data (Bytes 0:3 of the selected word) - when accessing the data array of either cache: 0:7 Byte 0 8:15 Byte 1 16:23 Byte 2 24:31 Byte 3

11.20 Hardware Debug (Cache) Control Register 0

Hardware debug control register 0 is used to disable certain cache features for hardware debug purposes. This register is not intended for normal user use. The HDBCR0 register is accessed using a **mfspr** or **mtspr** instruction. The SPR number for HDBCR0 is 976 in decimal. The HDBCR0 register is shown in Figure 11-17.

0–24	25	26	27	28	29	30	31
0	MBD	SNPDIS	0	DSB	DSTRM	0	ISTRM

SPR - 976; Read/Write; Reset - 0x0; Supervisor-only

Figure 11-17. Hardware Debug Control Register 0 (HDBCR0)

The HDBCR0 bits are described in Table 11-13.

Table 11-13. HDBCR0 field descriptions

Bits	Name	Description
0:24	—	Reserved[1]
25	MBD	Msync/Mbar Broadcast Disable 0 msync/mbar broadcasting is enabled. **p_sync_req_out** asserted normally and **p_sync_ack_in** is used to terminate msync and mbar MO=0,1 instruction execution 1 msync/mbar broadcasting is disabled. **p_sync_req_out** remains negated, and **p_sync_ack_in** is ignored and not used to terminate msync and mbar MO=0,1 instruction execution. **Note:** MBD settings have no effect on the operation of **p_sync_req_in** and **p_sync_ack_out**. Normal handshaking and completion of the synchronization request input will be performed.

Table 11-13. HDBCR0 field descriptions (continued)

Bits	Name	Description
26	SNPDIS	Snoop Disable 0 Snooping is not disabled. Snoops are processed normally according to the settings of $L1CSR0_{DCE}$. 1 Snoop lookups are disabled. Snoops are processed in the same manner as when the data cache is disabled, i.e null responses are generated and no snoop lookups are performed.
27	—	Reserved[1]
28	DSB	Disable Store Buffer 0 Store buffer enabled 1 Store buffer disabled
29	DSTRM	Disable Data Cache Streaming 0 DCache streaming is enabled 1 DCache streaming is disabled
30	—	Reserved[1]
31	ISTRM	Disable Instruction Cache Streaming 0 ICache streaming is enabled 1 ICache streaming is disabled

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

11.21 Hardware cache coherency

Hardware cache coherency is supported to allow for dual-core or CPU + I/O coherency. The cache must operate in writethrough mode for those pages of memory requiring coherency operations. Coherency is maintained by the use of snoop invalidation commands provided to the CPU through a dedicated snoop interface port. Snooping is only performed while the data cache is enabled ($L1CSR0_{DCE}$ =1). Figure 11-18 shows an abstract block diagram of the structure.

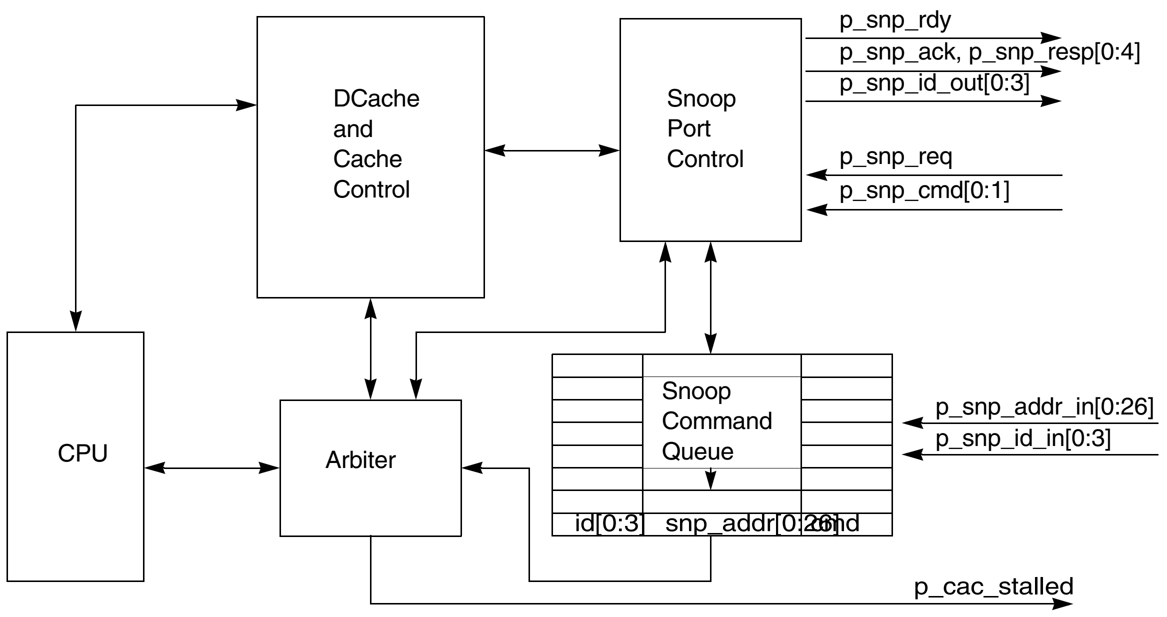


Figure 11-18. Snoop command port

11.21.1 Coherency protocol

The cache operates in a 2-state protocol for coherency purposes. The only state a coherent cache line should assume is Valid or Invalid. No Modified or Shared state is supported for coherent cache lines (although modified state is available for non-coherent lines), thus no snoop copyback or intervention operations are required. A snoop invalidation signaling port is provided to receive coherency requests. Snoop invalidation requests are received at the snoop invalidation port, and arbitrate with the CPU for access to the data cache tags for lookup and cache line invalidation. External coherency logic provides snoop invalidation requests to the snoop invalidation port based on the bus activity of other coherent bus masters, and these invalidation requests are later processed and a response provided. Memory regions that require coherency operations must be marked as “memory coherence required” (page’s M bit set) and as “writethrough” (page’s W bit set).

External data accesses by the CPU reflect the value of the M bit of the accessed page on the **p_d_gbl** output. Typically, external coherency logic will monitor external accesses by a CPU (or other agent), and will request invalidation operations to other coherent entities for write accesses that also have **p_d_gbl** asserted. Non-shared data should be placed into pages with the M bit cleared, thus avoiding unnecessary coherency operations.

11.21.2 Snoop command port

The snoop command port provides the signaling mechanism between external coherency logic and the snoop request queue. Command requests are received on the **p_snp_cmd[0:1]**, **p_snp_id_in[0:3]**, and **p_snp_addr_in[0:26]** inputs when the **p_snp_req** signal is properly asserted, and responses to snoop command requests are provided on the **p_snp_ack**, **p_snp_resp[0:4]**, and **p_snp_id_out[0:3]** outputs.

Snoop invalidation requests provide the physical address of the data to be invalidated (**p_snp_addr_in[0:26]**), along with a four-bit ID field (**p_snp_id_in[0:3]**), which flows through the command pipeline and is returned on the **p_snp_id_out[0:3]** output port along with the completion status provided on **p_snp_resp[0:4]** when **p_snp_ack** is asserted.

The **p_snp_rdy** output signal provides a handshaking mechanism for flow control of snoop requests to prevent overflow of the internal snoop queue, which buffers incoming snoop requests from the snoop command port prior to cache tag lookups and updates. Negation of **p_snp_rdy** indicates that another snoop command port request will not be accepted due to resource constraints in the snoop pipeline.

Refer to Section 14.2.9, Coherency control signals, for details on the operating protocol of the snoop command port.

The command value is stored in the snoop queue along with the snoop address and snoop ID value. Table 11-14 shows the definitions of the **p_snp_cmd[0:1]** encodings.

Table 11-14. p_snp_cmd[0:1] Snoop command encoding

p_snp_cmd[0:1]	Response type
00	Null - no status bit operation performed, lookup is performed
01	INV - invalidate matching cache entry
10	SYNC - synchronize snoop queue
11	Reserved

The NULL command is used for testing of interface handshaking and other status gathering purposes. The NULL command performs a snoop lookup operation, but performs no actual cache tag or status modifications (even in the presence of tag EDC errors). The INV command causes a snoop lookup and subsequent invalidation of a matching cache line. The SYNC command causes the snoop queue to be emptied with highest priority relative to CPU requests.

Table 11-14 shows the definitions of the **p_snp_resp[0:4]** encodings.

Table 11-15. p_snp_resp[0:4] Snoop response encoding

p_snp_resp[0:4][1]	Response type
000cc	NULL - no operation performed or no matching cache entry
001cc	AutoInv - AutoInvalidation performed on clean unlocked lines with tag parity errors
010cc	ERROR - Error in processing a snoop request due to TAG parity error. For NULL commands, a tag parity error occurred and no hit to a tag without error occurred. No modification of cache entries, no machine check generated internally. For INV commands, a) possible invalidation of locked line with tag parity error occurred, or b) dirty line left valid with tag parity error, or c) no true hit occurred, and one or more lines reported tag parity errors. Machine check generated internally.
01100	SYNC - Sync completed, snoop queue synchronized
100cc	HIT Clean- matching unlocked cache entry found

Table 11-15. p_snp_resp[0:4] Snoop response encoding (continued)

p_snp_resp[0:4][1]	Response type
101cc	HIT Dirty- matching unlocked dirty cache entry found
110cc	HIT Locked - matching clean locked cache entry found
111cc	HIT Dirty Locked - matching dirty locked cache entry found

[1] cc - # collapsed requests; 00-no collapsing, 01- two requests combined, 10- three requests combined, 11- four requests combined

The NULL response indicates there was no matching cache entry found for a null or invalidate command or the cache was disabled when the request was originally made. The HIT responses indicates that a matching cache entry was found. The SYNC response indicated all previous entries in the snoop queue were emptied. The ERROR response indicates that an error occurred in processing a snoop request due to a cache tag parity error. The AutoInv response indicates one or more cache lines with tag parity errors was invalidated.

Responses for a Null command are either NULL, HIT, or ERROR. Responses for an INV command are either Null (no hit occurred or cache is disabled), Hit (a matching entry was found and invalidated), or ERROR (a tag parity error was found and left valid, no guarantee of the command success). Responses for a Sync command are SYNC completed.

11.21.3 Snoop request queue

The snoop request queue provides a queueing mechanism between the snoop command port and the cache. As requests are accepted from the snoop invalidate port, they are queued into an 8-deep fifo queue for arbitration to the cache for tag and status lookup and conditional status clearing.

Snoops can be collapsed within the queue under certain circumstances to minimize the number of invalidation lookups performed. When two consecutive snoop requests refer to the same cache line, they are collapsed (timing permitting) into a single snoop invalidation cycle. Collapsed entries are indicated complete via an encoding of the **p_snp_resp[0:4]** status outputs.

Snoop invalidation requests have a lower priority than CPU data accesses or change of flow accesses when only a single queue entry is occupied. This allows for some optimization in cycle-stealing of the tag array from the CPU in an attempt to minimize CPU stalls. Snoop invalidation request priority is raised when a “snoop sync” command is received on the snoop command port or when a sync request is generated on the synchronization port (**p_sync_req_in**), regardless of the number of active queue entries.

11.21.4 Snoop lookup operation

Entries in the snoop request queue are processed in-order after arbitrating for the cache tag and status bit arrays. Once the CPU has been stalled from performing further tag accesses, the snoop request queue is processed by performing a tag lookup, and a subsequent status bit write to clear the valid bit of a matching valid entry. Invalidation hits require two tag array accesses to first read, and then to update the valid bit. A subsequent snoop lookup may be pipelined while the first lookup of a pair of lookups is being processed to determine a hit/miss condition. In this manner, a pair of hitting invalidation requests will block the CPU for a total of 5 cycles. A single snoop lookup requires 3 cycles of latency on a miss, and 4 cycles on a hit prior to allowing the CPU to resume cache accesses. If the snoop queue contains enough entries, snoop read and write accesses to the cache tag are pipelined, and the total blockage will be 3*number_of_hits + number_of_misses + 1. In certain cases where the CPU has pipelined one or more cache misses, initial snoop accesses will be interlaced with CPU tag accesses prior to assuming highest priority in order to allow for proper operation of linefill and copyback operations initiated by the CPU.

As entries are removed from the queue and the invalidation lookups are performed, the results of the lookups are provided on the response output signals, along with the original request ID.

11.21.5 Snoop errors

Errors can occur during snoop lookup operations and are signaled on the snoop response output port. Tag parity errors that prevent an accurate hit/miss determination on the snoop request address may result in an error response signaled via **p_snp_resp[0:4]**, as well as a machine check to the CPU for the INV command if a locked line was invalidated, if a line was dirty and not invalidated, or if a tag parity error occurred and no hit occurred to a line without error. When such a tag parity error occurs, the invalidation will not occur to the line(s) with error. The snoop queue will continue to be serviced, and the machine check will not necessarily be recoverable. A checkstop condition will not occur however. In this respect, it is treated similarly to a non-maskable interrupt, and the MSR[RI] bit should be used accordingly by software.

11.21.6 Snoop collisions

Snoop requests may collide with an outstanding or pending cache linefill.

Since there is no particular guarantee of the precise time an actual snoop invalidation lookup will occur relative to a cache linefill request, the CPU may in some instances be in the process of filling a line corresponding to a snoop invalidate request. In this case, the snoop will cause the linefills to be marked such that they are not loaded into the cache. Load miss operations that are in progress may use the data as it returns however. The responses for these collisions will be based on the state the cache line would have taken if the linefill completes successfully.

Snoop requests should not collide with dirty line copyback or flush operations, since the coherent pages must be marked as writethrough required. These snoop collisions will be ignored.

11.21.7 Snoop synchronization

Synchronization of the snoop queue will occur under two conditions; a synchronization port sync request, and a snoop command port sync request.

11.21.7.1 Synchronization port request

Assertion of the **p_sync_req_in** signal will cause the snoop queue to assume highest priority, and be flushed. It is assumed that the system will stop generating snoop requests during a synchronization of the queue to allow it to drain, but if snoop requests continue to be received, the acknowledgement of the synchronization request will be delayed until the queue finally drains to the point that all queue entries that were present prior to the recognition of the sync request have been serviced.

In general, the synchronization port is expected to be utilized to handshake execution of **msync** instructions from an alternate CPU, thus there will typically not be additional snoop requests occurring until the synchronization handshake is complete, since no further bus writes will be requested by the alternate CPU, but if additional coherency traffic occurs due to another alternate master, it will follow the normal queueing process and will not block the eventual assertion of the **p_sync_ack_out** signal.

11.21.7.2 Snoop command port request

Receiving a snoop command port “snoop sync” request encoded via the **p_snp_cmd[0:1]** inputs of will cause the snoop queue to assume highest priority, and be flushed to the point the command has reached the head of the queue and been acknowledged. After the command has been completed, snoop queue priority will revert to normal operation, unless another “snoop sync” command has been received and placed into the queue, in which case snoop queue priority will remain elevated until all “snoop sync” commands have been processed from the queue.

11.21.8 Starvation control

To avoid starvation of a higher priority CPU due to a continuous stream of snoop requests from a lower priority master that block CPU forward progress, some form of starvation control is desired. This is implemented with a forward progress counter, which tracks the number of contiguous cycles the CPU has been prevented from accessing the cache due to snoop command port access requests. Once the count has been exceeded, the CPU will regain highest priority for one access cycle. A similar counter exists for the snoop queue, to allow for periodic snoop request processing when the queue holds only a single entry. Each counter is 4 bits, and causes a priority inversion to occur for tag access upon timeout. The presence of one or more sync commands in the snoop queue when the counter expires will delay the priority inversion until the queue has been emptied up to the point that the sync(s) have been completed. Subsequent syncs received while the starvation timeout is being postponed may also prevent the priority inversion after the original sync(s) have been completed if additional snoops have been queued during the sync command processing. This is not normally expected to occur in a typical system however.

In addition, external logic may be used to implement additional safeguards by monitoring the **p_cac_stalled** output, which indicates that the CPU has a pending cache access request blocked due to snoop access activity.

11.21.9 Queue flow control

To avoid overflow of the snoop queue, the **p_snp_rdy** output is provided to indicate whether an additional snoop command port request will be accepted on the following clock cycle. When negated, no further command requests can be honored until a snoop queue entry becomes available.

To provide for flow control of CPU-generated snoop requests to another CPU’s queue, the **p_stall_bus_gwrite** input is provided, which will cause further bus activity that is requesting a global write cycle to be suspended. Other bus traffic is not affected.

11.21.10 Snooping in low power states

Snooping remains enabled while in the Waiting or Halted states if the clock is running. Snoops should only be issued to the core complex while the core is in the normal, Halted, or Waiting states and both the **p_stop** and **p_stopped** signals are negated. When a request is made to enter stop mode via the assertion of **p_stop**, the **p_snp_rdy** output will be negated. While the core complex is in the Stopped (power-down) state, bus snooping is disabled, and the **p_snp_rdy** output is held negated. Snoop requests will be processed around the assertion of the stop mode entry request (assertion of **p_stop**) per the normal protocol associated with **p_snp_rdy** negation, including acceptance of a snoop request during a small interval around **p_snp_rdy** negation, thus additional snoop operations may need to occur prior to entering the stopped state. All snoop queue entries will be processed prior to the assertion of **p_stopped**.

Chapter 12
Debug Support

This chapter describes the debug features of the e200z759n3 core.

12.1 Overview

Internal debug support in the e200z759n3 core allows for software and hardware debug by providing debug functions, such as instruction and data breakpoints and program trace modes. For software based debugging, debug facilities consisting of a set of software accessible debug registers and interrupt mechanisms are provided. These facilities are also available to a hardware based debugger, which communicates using a modified IEEE 1149.1 Test Access Port (TAP) controller and pin interface. When hardware debug is enabled, the debug facilities controlled by hardware are protected from software modification.

Software debug facilities are defined as part of *PowerISA 2.06*. e200z759n3 supports a subset of these defined facilities. In addition to the facilities defined in *PowerISA 2.06*, e200z759n3 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 e200z759n3 core also provides support for run-time integrity checking via a Parallel Signature unit, which is capable of monitoring the internal CPU data read and data write buses, and accumulating a pair of 32-bit MISR signatures of the data values transferred over these buses.

12.1.1 Software debug facilities

e200z759n3 provides debug facilities to enable hardware and software debug functions, such as instruction and data breakpoints and program single stepping. The debug facilities consist of a set of debug control registers (DBCR0-6, DBERC0), a set of address compare registers (IAC1-8, DAC1, and DAC2), a set of data value compare registers (DVC1, DVC2), a configurable Debug Counter, a Debug Status Register (DBSR) for enabling and recording various kinds of debug events, and a special Debug interrupt type built into the interrupt mechanism (see Section 7.7.16, Debug interrupt (IVOR15)). The debug facilities also provide a mechanism for software-controlled processor reset, and for controlling the operation of the timers in a debug environment.

Software debug facilities are enabled by setting the internal debug mode bit in Debug Control register 0 ($DBCR0_{IDM}$). When internal debug mode is enabled, debug events can occur, and can be enabled to record exceptions in the Debug Status register (DBSR). If enabled by MSR_{DE}, these recorded exceptions cause Debug interrupts to occur. When $DBCR0_{IDM}$ is cleared (and $DBCR0_{EDM}$ is cleared as well), no debug events occur, and no status flags are set in DBSR unless already set. In addition, when $DBCR0_{IDM}$ is cleared (or is overridden by $DBCR0_{EDM}$ being set and DBERC0 indicating no resource is “owned” by software) no Debug interrupts will occur, regardless of the contents of DBSR. A software Debug interrupt handler may access all system resources and perform necessary functions appropriate for system debug.

12.1.1.1 *PowerISA 2.06* compatibility

The e200z759n3 core implements a subset of the *PowerISA 2.06* 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.

12.1.2 Additional debug facilities

In addition to the debug functionality defined in *PowerISA 2.06*, e200z759n3 provides capability to link instruction and data breakpoints, provides a configurable debug event counter to allow debug exception generation capability, and also provides a sequential breakpoint control mechanism.

e200z759n3 also defines two new debug events (CIRPT, CRET) for debugging around critical interrupts.

In addition, e200z759n3 implements the Debug APU, which when enabled allows Debug Interrupts to utilize a dedicated set of save/restore registers (DSRR0, DSRR1) for saving state information when a Debug Interrupt occurs, and for restoring this state information at the end of a debug interrupt handler by means of the **rfdi** or **se_rfdi** instructions.

Zen also provides the capability of sharing resources between hardware and software debuggers. See Section 12.1.4, Sharing debug resources by software/hardware.

12.1.3 Hardware debug facilities

The e200z759n3 core contains facilities that allow for external test and debugging. A modified IEEE 1149.1 control interface is used to communicate with the core resources. This interface is implemented through a standard 1149.1 TAP (test access port) controller.

By using public instructions, the external debugger can freeze or halt the e200z759n3 core, read and write internal state and debug facilities, single-step instructions, and resume normal execution.

Hardware Debug is enabled by setting the External Debug Mode enable bit in Debug Control register 0 ($\text{DBCR0}_{\text{EDM}}$), which is also aliased to $\text{EDBCR0}_{\text{EDM}}$. Setting $\text{DBCR0}_{\text{EDM}}$ overrides the Internal Debug Mode enable bit $\text{DBCR0}_{\text{IDM}}$ unless resources are provided back to software via the settings in DBERC0. When the Hardware Debug facility is enabled, software is blocked from modifying the “hardware-owned” debug facilities. In addition, since resources are “owned” by the hardware debugger, inconsistent values may be present if software attempts to read “hardware-owned” debug-related resources.

When hardware debug is enabled by setting $\text{[E]DBCR0}_{\text{EDM}}$=1, the control registers and resources described in Section 12.3, Debug registers are reserved for use by the external debugger. The same events described in Section 12.2, Software debug events and exceptions are also used for external debugging, but exceptions are not generated to running software. Hardware-owned debug events enabled in the respective DBCR0-6 registers are recorded in the EDBSR0 register (not the DBSR) regardless of MSR_{DE}, and no debug interrupts are generated unless the resource is granted back to software via DBERC0 settings, and debug mode entry is not masked by the corresponding event bit in EDBSRMSK0. Instead, the CPU will enter debug mode when an enabled event causes a EDBSR0 bit to become set. $\text{DBCR0}_{\text{EDM}}$, EDBSR0, EDBSRMSK0, and DBERC0 may only be written through the OnCE port.

A program trace PC FIFO is also provided to support program change of flow capture.

Access to most debug resources (registers) requires that the CPU clock (**m_clk**) be running in order to perform write accesses from the external hardware debugger.

12.1.4 Sharing debug resources by software/hardware

Debug resources may be shared by a hardware debugger and software debug based on the settings of debug control register DBERC0. When $DBCR0_{EDM}$ is set, DBERC0 settings determine which debug resources are allocated to software and which resources remain under exclusive hardware control. Software-owned resources that set DBSR bits when $DBCR0_{IDM}$=1 will cause a debug interrupt to occur when enabled with MSR_{DE}. Hardware-owned resources that set EDBSR0 bits when $[E]DBCR0_{EDM}$=1 will cause an entry into debug mode if the event is not masked in EDBSRMSK0. DBERC0 is read-only by software. When resource sharing is enabled, ($DBCR0_{EDM}$=1 and $DBERC0_{IDM}$=1), only software-owned resources may be modified by software. Hardware always has full access to all registers and all register fields through the OnCE register access mechanism, and it is up to the debug firmware to properly implement modifications to these registers with read-modify-write operations to implement any control sharing with software. Hardware-owned resources will set status bits in the EDBSR0 register instead of in DBSR. Settings in DBERC0 should be considered by the debug firmware in order to preserve software settings of control and status registers as appropriate when hardware modifications to the debug registers is performed.

12.1.4.1 Simultaneous hardware and software debug event handing

Since it is possible that a “hardware-owned” resource can produce a debug event in conjunction with a software-owned resource producing a different debug event simultaneously, a priority ordering mechanism is implemented that guarantees that the hardware event is handled as soon as possible, while preserving the recognition of the software event. The CPU will give highest priority to the software event initially in order to reach a recoverable boundary, and then will give highest priority to the hardware event in order to enter debug mode as near the point of event occurrence as possible. This is implemented by allowing software exception handing to begin internal to the CPU and to reach the point where the current program counter and MSR values have been saved into DSRR0/1, and the new PC pointing to the debug interrupt handler, along with the new MSR updates. At this point, hardware priority takes over, and the CPU enters debug mode.

Figure 12-1 shows the e200z759n3 debug resources.

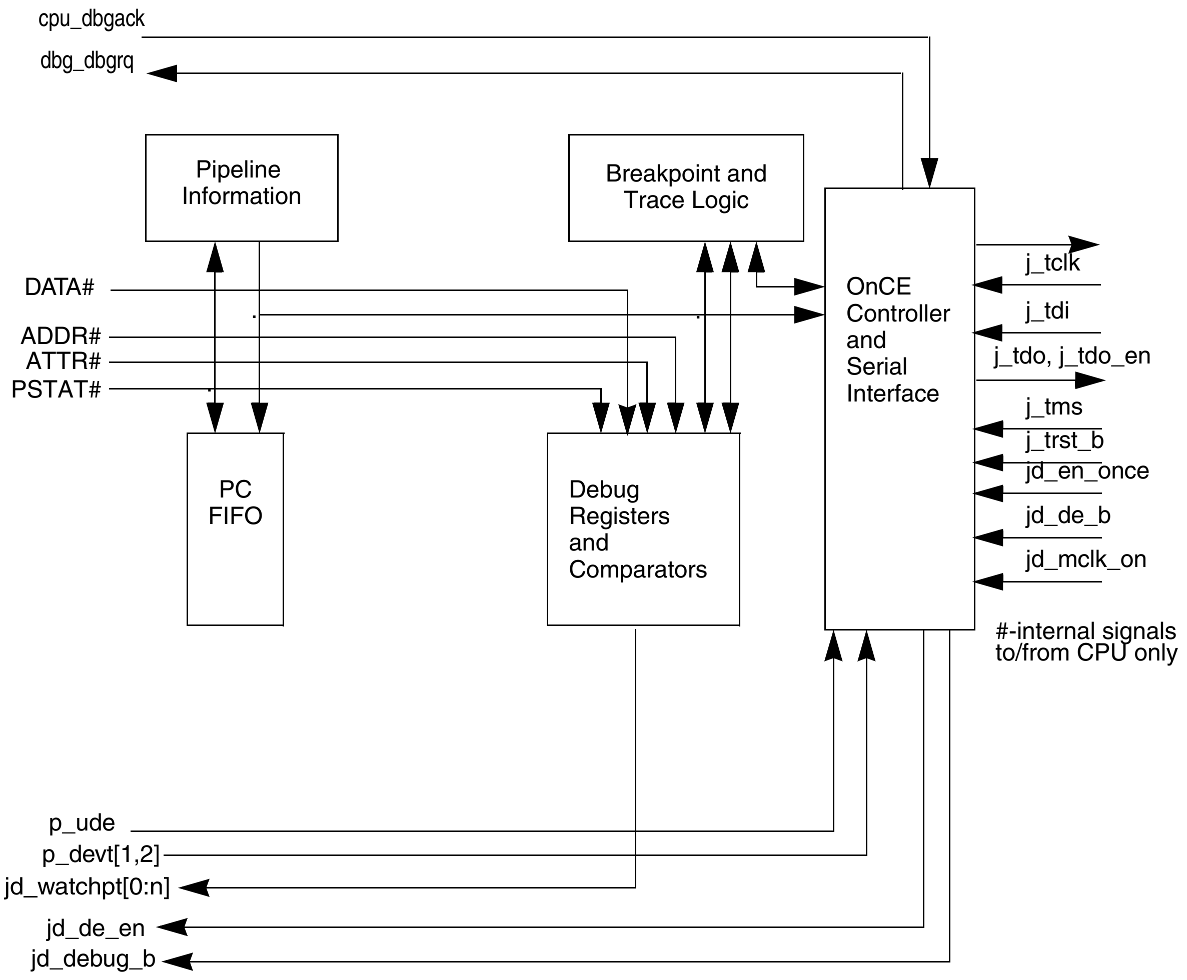


Figure 12-1. e200z759n3 debug resources

12.2 Software debug events and exceptions

Software debug events and exceptions are available when internal debug mode is enabled (DBCR0$_{IDM}$=1) and not overridden by external debug mode (DBCR0$_{EDM}$ must either be cleared or corresponding resources must be allocated to software debug by the settings in DBERC0). When enabled, debug events cause debug exceptions to be recorded in the Debug Status Register. Specific event types are enabled by the Debug Control Registers (DBCR0–6). The Unconditional Debug Event (UDE) is an exception to this rule; it is always enabled. Once a Debug Status Register (DBSR) bit is set by a debug resource that is “owned” by software (other than MRR and CNT1TRG), if Debug interrupts are enabled by MSR$_{DE}$, a Debug interrupt will be 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$_{IDM}$=1. In such situations, no debug exception occurs and thus no DBSR bit is set. Other debug events may cause debug exceptions and set DBSR bits regardless of the state of MSR$_{DE}$. A Debug interrupt will be delayed until MSR$_{DE}$ is later set to ‘1’.

When a Debug Status Register bit is set while MSR_{DE}=0, an Imprecise Debug Event flag ($DBSR_{IDE}$) will also be set to indicate that an exception bit in the Debug Status Register was set while Debug interrupts were disabled. Debug interrupt handler software can use this bit to determine whether the address recorded in Debug Save/Restore Register 0 is an address associated with the instruction causing the debug exception, or the address of the instruction that enabled a delayed Debug interrupt by setting the MSR_{DE} bit. A **mtmsr** or **mtdbcr0** that causes both MSR_{DE} and $DBCR0_{IDM}$ to become set, enabling precise debug mode, may cause an Imprecise (Delayed) Debug exception to be generated due to an earlier recorded event in the Debug Status register.

There are eight types of debug events defined by *PowerISA 2.06*:

1. Instruction Address Compare debug events
2. Data Address Compare debug events
3. Trap debug events
4. Branch Taken debug events
5. Instruction Complete debug events
6. Interrupt Taken debug events
7. Return debug events
8. Unconditional debug events

These events are described in detail beginning on page 198 of *Book E: Enhanced PowerPC^{tm} Architecture v0.99*.

In addition, e200z759n3 defines additional debug events:

- The Debug Counter debug events DCNT1 and DCNT2, which are described in Section 12.2.11, Debug Counter debug event.
- The External debug events DEVT1 and DEVT2, which are described in Section 12.2.12, External debug event.
- The Critical Interrupt Taken debug event CIRPT, which is described in Section 12.2.8, Critical Interrupt Taken debug event.
- The Critical Return debug event CRET, which is described in Section 12.2.10, Critical Return debug event.

The e200z759n3 debug configuration supports most of these event types. Unsupported *PowerISA 2.06* defined functionality is as follows:

- Instruction Address Compare and Data Address Compare *Real address* mode is not supported

A brief description of each of the event types follows. In these descriptions, DSRR0 and DSRR1 are used, assuming that the Debug APU is enabled. If it is disabled, use CSRR0 and CSRR1 respectively.

12.2.1 Instruction Address Compare event

Instruction Address Compare debug events occur when enabled and execution is attempted of an instruction at an address that meets the criteria specified in the DBCR0, DBCR1, DBCR5, DBCR6, and IAC1-8 Registers. Instruction Address compares may specify user/supervisor mode and instruction space (MSR_{IS}), along with an effective address, masked effective address, or range of effective addresses for comparison (range compares are not supported for IAC5-8). This event can occur and be recorded in DBSR regardless of the setting of MSR_{DE}. IAC events will not occur when an instruction would not have normally begun execution due to a higher priority exception at an instruction boundary.

IAC compares perform a 31-bit compare for VLE instruction pages, and 30-bit compares for BookE instruction pages. Each halfword fetched by the instruction fetch unit will be marked with a set of bits indicating whether an Instruction Address Compare occurred on that halfword. Debug exceptions will occur if enabled and a 16-bit instruction, or the first halfword of a 32-bit instruction, is tagged with an IAC hit. For instruction fetches that miss in the TLB, BookE pages are assumed, and a 30-bit compare is performed.

12.2.2 Data Address Compare event

Data Address Compare debug events occur when enabled and execution of a load or store class instruction or a cache maintenance instruction results in a data access that meets the criteria specified in the DBCR0, DBCR2, DBCR4, DAC1, DAC2, DVC1, and DVC2 Registers. Data address compares may specify user/supervisor mode and data space (MSR_{DS}), along with an effective address, masked effective address, or range of effective addresses for comparison. This event can occur and be recorded in DBSR regardless of the setting of MSR_{DE}. Two address compare values (DAC1, DAC2) are provided.

NOTE

In contrast to the *PowerISA 2.06* definition, Data Address Compare events on e200z759n3 do not prevent the load or store class instruction from completing. If a load or store class instruction completes successfully without a Data TLB or Data Storage interrupt, Data Address Compare exceptions are reported at the completion of the instruction. If the exception results in a precise Debug interrupt, the address value saved in DSRR0 (or CSRR0 if the Debug APU is disabled) is the address of the instruction following the load or store class instruction. For DVC DAC events, the exception can be imprecisely reported even further past the load or store class instruction generating the event (without necessarily affecting $DBSR_{IDE}$) and the saved address value can point to a subsequent instruction past the next instruction. This occurrence is indicated in the $DBSR_{DAC\_OFST}$ field.

If a load or store class instruction does not complete successfully due to a Data TLB or Data Storage exception or a machine check condition for the load or store, and a Data Address Compare debug exception also occurs, or a Debug Counter event based on a counted DAC occurs, the result is an imprecise Debug interrupt, the address value saved in DSRR0 (or CSRR0 if the Debug APU is disabled) is the address of the load or store class instruction, and the $DBSR_{IDE}$ bit will be set. In addition to occurring when $DBCR0_{IDM}$=1, this circumstance can also occur when $DBCR0_{EDM}$=1.

NOTE

DAC events will not be recorded or counted if a load multiple word or store multiple word instruction is interrupted prior to completion by a critical input or external input interrupt.

NOTE

DAC events are not signaled on the second portion of a misaligned load or store that is broken up into two separate accesses.

NOTE

DAC events are not signaled on the **tlbre**, **tlbwe**, **tlbsx**, or **tlbivax** instructions.

NOTE

DAC[1,2] events are not signaled if DVC[1,2]M is non-zero and a DSI or DTLB exception occurs on the load or store, since the load or store access is not performed. For a **lmw** or **stmw** transfer however, if a DVC successfully occurs on a transfer and a later transfer encounters a DSI or DTLB exception, the DAC event will be reported, since a successful data value compare took place.

12.2.2.1 Data Address Compare event status updates

Data Address Compare debug events with Data Value compares can be reported ambiguously in several circumstances involving issuing a sequence of load or store class instructions. Due to the CPU pipeline and the delay in performing the data value compare following completion of the access, if the first load or store class instruction generates a DVC DAC, a second and possibly third load or store class instruction may also generate a DAC or DVC DAC event, or may generate a DTLB or DSI exception with or without a simultaneous DAC event.

Also, since non-load/store instructions may be dual-issued in combination with a load/store instruction, the actual number of additional instructions that are completed following a recognized DVC DAC on a load/store instruction may vary from 0 to 5. This value will be reported in the $\text{DBSR}_{\text{DAC\_OFST}}$ field when the DVC DAC status is recorded.

Table 12-1 outlines the settings of the DBSR, DSRR0 saved value, and potential updating of the ESR and MMU MASx registers for various exception cases on sequences of load/store class instructions. Not all exception combinations are covered in the table, such as IAC, ITLB, ISI, or Alignment exceptions on subsequent instructions. In general these exceptions will cause further instruction issue to be halted, execution of the excepting instruction to be aborted, and reporting of these exceptions will be masked. The saved DSRR0 value will point to this excepting instruction, and the exception(s) may be regenerated after returning from the debug interrupt handler and attempting to re-execute the instruction pointed to by DSRR0. In addition, in the examples in Table 12-1, the DAC_OFST and DSRR0 values assume no dual issue occurs. If dual-issue occurs with the first, second, or third column, then the DAC_OFST and DSRR0 values will point beyond the values shown.

Table 12-1. DAC events and resultant updates

1st load/store class instruction	2nd instruction (load/store class unless otherwise specified)	3rd instruction (load/store class unless otherwise specified)	Result
DTLB Error, no DAC	—	—	Take DTLB exception, no DBSR update, update MASx registers for 1st load/store class instruction. Update ESR.
DSI, no DAC	—	—	Take DSI exception, no DBSR update, no MASx register update. Update ESR.
DTLB Error, with DACx	—	—	Take Debug exception, DBSR update setting DACx and IDE, DAC_OFST not set. No MASx register update for 1st load/store class instruction. DSRR0 points to 1st load/store class instruction. No ESR update.
DSI, with DACx	—	—	Take Debug exception, DBSR update setting DACx and IDE, DAC_OFST not set. DSRR0 points to 1st load/store class instruction. No MASx register update. No ESR update.
DACx	—	—	Take Debug exception, DBSR update setting DACx, DAC_OFST not set. DSRR0 points to 2nd load/store class instruction. No MASx register update. No ESR update.
DVC DACx	No exceptions, any instruction	No exceptions, Non-ldst instruction	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 3'b001. DSRR0 points to 3rd instruction. No MASx register update. No ESR update.
DVC DACx	No exceptions	No exceptions, Ldst instruction	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 3'b010. DSRR0 points to instruction after 3rd instruction. No MASx register update. No ESR update.
DVC DACx	DTLB Error, no DAC	—	Take Debug exception, DBSR update setting DACx, DAC_OFST not set. DSRR0 points to 2nd load/store class instruction. no MASx register update. No ESR update. No debug counter updates for 2nd ld/st instruction. **Note:** In this case the 2nd ld/st exception is masked. This behavior is implementation dependent and may differ on other CPUs.
DVC DACx	DSI, no DAC	—	Take Debug exception, DBSR update setting DACx, DAC_OFST not set. DSRR0 points to 2nd load/store class instruction. No MASx register update. No ESR update. No debug counter updates for 2nd ld/st instruction. **Note:** In this case the 2nd ld/st exception is masked. This behavior is implementation dependent and may differ on other CPUs.
DVC DACx	DTLB Error, with DACy	—	Take Debug exception, DBSR update setting DACx. DAC_OFST not set. DSRR0 points to 2nd load/store class instruction. No MASx register update. No ESR update. No debug counter update occurs for the 2nd ld/st. **Note:** In this case the 2nd ld/st exception is masked. This behavior is implementation dependent and may differ on other CPUs.

Table 12-1. DAC events and resultant updates (continued)

1st load/store class instruction	2nd instruction (load/store class unless otherwise specified)	3rd instruction (load/store class unless otherwise specified)	Result
DVC DACx	DSI, with DACy	—	Take Debug exception, DBSR update setting DACx. DAC_OFST not set. DSRR0 points to 2nd load/store class instruction. No MASx register update. No ESR update. No debug counter update occurs for the 2nd ld/st. **Note:** In this case the 2nd ld/st exception is masked. This behavior is implementation dependent and may differ on other CPUs.
DVC DACx	DACy	—	Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 3'b001. DSRR0 points to 3rd instruction. Debug counter update occurs for the 2nd ld/st as appropriate. **Note:** In this case debug counter updates can occur for the 2nd ld/st even though the 1st ld/st has a DVC DAC exception[1]. **Note:** In this case if x==y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the "no DACy" case.
DVC DACx	DVC DACy, Normal Ldst	Non-Ldst instruction	Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 3'b001. DSRR0 points to the 3rd instruction. Debug counter update occurs for the 2nd ld/st as appropriate. **Note:** In this case debug counter updates occur for the 2nd ld/st even though the 1st ld/st has a DVC DAC exception[1]. **Note:** In this case if x==y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the "no DACy" case.
DVC DACx	DVC DACy, Normal Ldst	Ldst instruction, no exception	Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 3'b010. DSRR0 points to instruction after the 3rd load/store class instruction. Debug counter update occurs for the 2nd and 3rd ld/st as appropriate. **Note:** In this case debug counter updates occur for the 2nd and 3rd ld/st even though the 1st ld/st has a DVC DAC exception[2]. **Note:** In this case if x==y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the "no DACy" case.
DVC DACx	DVC DACy, Normal Ldst	DSI Error, with or without DAC	Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 3'b001. No ESR update. DSRR0 points to 3rd instruction. Debug counter update occurs for the 2nd ld/st as appropriate. **Note:** In this case debug counter updates occur for the 2nd ld/st even though the 1st ld/st has a DVC DAC exception[1]. **Note:** In this case if x==y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the "no DACy" case. **Note:** In this case the 3rd ld/st exception is masked. This behavior is implementation dependent and may differ on other CPUs.

Table 12-1. DAC events and resultant updates (continued)

1st load/store class instruction	2nd instruction (load/store class unless otherwise specified)	3rd instruction (load/store class unless otherwise specified)	Result
DVC DACx	DVC DACy, Normal Ldst	DTLB, with or without DAC	Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 3'b001. No ESR update. No MASx register updates. DSRR0 points to 3rd instruction. Debug counter update occurs for the 2nd ld/st as appropriate. **Note:** In this case debug counter updates occur for the 2nd ld/st even though the 1st ld/st has a DVC DAC exception[1]. **Note:** In this case if x==y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the "no DACy" case. **Note:** In this case the 3rd ld/st exception is masked. This behavior is implementation dependent and may differ on other CPUs.
DVC DACx	DVC DACy, Normal Ldst	DACy, or DVC DAC_y Normal Ldst or multiple word Ldst	Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 3'b010. DSRR0 points to instruction after the 3rd load/store class instruction. Debug counter update occurs for the 2nd and 3rd ld/st as appropriate. **Note:** In this case debug counter updates occur for the 2nd and 3rd ld/st even though the 1st ld/st has a DVC DAC exception[2]. **Note:** In this case if x==y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the "no DACy" case.
DVC DACx	DVC DACy, Ldst multiple (lmw, stmw)	Any instruction including ld/st	Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 3'b001. DSRR0 points to the 3rd instruction. Debug counter update occurs for the 2nd ld/st multiple as appropriate. **Note:** In this case debug counter updates occur for the 2nd ld/st multiple even though the 1st ld/st has a DVC DAC exception[1]. **Note:** In this case if x==y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the "no DACy" case.
DVC DACx	Any instruction (no exception)	DSI, with or without DAC, Normal Ldst or multiple word Ldst	Take Debug exception, DBSR update setting DACx. DAC_OFST set to 3'b001. DSRR0 points to the 3rd instruction. No MASx register update. No ESR update. No debug counter update occurs for the 3rd instruction. Debug counter update occurs for the 2nd instruction as appropriate. **Note:** In this case debug counter updates occur for the 2nd instruction even though the 1st ld/st has a DVC DAC exception[1]. **Note:** In this case the 3rd ld/st exception is masked. This behavior is implementation dependent and may differ on other CPUs.
DVC DACx	Any instruction (no exception)	DACy, or DVC DAC_y Normal Ldst or multiple word Ldst	Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 3'b010. DSRR0 points to instruction after the 3rd class instruction. Debug counter update occurs for the 2nd and 3rd instruction as appropriate. **Note:** In this case debug counter updates occur for the 2nd and 3rd instructions even though the 1st ld/st has a DVC DAC exception[2]. **Note:** In this case if x==y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the "no DACy" case.

[1] The 2nd instruction may cause DAC, ICMP or IAC events to be counted.
[2] The 2nd and 3rd instructions may cause DAC, ICMP or IAC events to be counted.

Table 12-2 – Table 12-5 show some example updates for specific code sequences of dual issuing of load/store class instructions with non-load/store class instructions and the results of DAC and DVC events on selected ones of the load/store instructions.

Table 12-2. DAC events and resultant updates, dual-issue case 1

Instruction Sequence: The following pairs dual-issue: (1) load/store (2) alu (3) load/store (4) alu (5) load/store (6) alu	**Event(s)**	**Result**
	Instruction (1): DTLB Error, no DAC	Take DTLB exception, no DBSR update, update MASx registers for 1st load/store instruction. Update ESR.
	Instruction (1): DSI, no DAC	Take DSI exception, no DBSR update, no MASx register update. Update ESR.
	Instruction (1): DTLB Error, with DACx	Take Debug exception, DBSR update setting DACx and IDE, DAC_OFST set to 3’b000. DSRR0 points to instruction (1). No MASx register update. No ESR update.
	Instruction (1): DSI, with DACx	
	Instruction (1): DACx	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 3’b000. DSRR0 points to instruction (2). No MASx register update. No ESR update.
	Instruction (1): DVC DACx No other exceptions	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 3’b100. DSRR0 points to instruction (6). No MASx register update. No ESR update. Debug counter update occurs for instructions (1)-(5) as appropriate. No debug counter or event updates for instruction (6)

	Instruction (1):
	DVC DACx
	Instruction (3):
	DTLB Error, with or without DAC
		Take Debug exception, DBSR update setting DACx, DAC_OFST set to 3’b001. DSRR0 points to instruction (3). no MASx register update. No ESR update. Debug counter update occurs for instructions (1)-(2) as appropriate. No debug counter or event updates for instructions (3)-(6).
		Note: In this case the 2nd ld/st exception is masked. This behavior is implementation dependent and may differ on other CPUs.

	Instruction (1):
	DVC DACx
	Instruction (3):
	DSI, with or without DAC	

Table 12-2. DAC events and resultant updates, dual-issue case 1 (continued)

Instruction Sequence: The following pairs dual-issue: (1) load/store (2) alu (3) load/store (4) alu (5) load/store (6) alu	**Event(s)**	**Result**
	Instruction (1): DVC DACx Instruction (3): DACy	Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 3’b010. DSRR0 points to instruction (4). Debug counter update occurs for instructions (1)-(3) as appropriate. No debug counter or event updates for instructions (4)-(6). **Note:** In this case if x==y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the “no DACy” case.
	Instruction (1): DVC DACx Instruction (3): DVC DACy	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 3’b100. DSRR0 points to instruction (6). No MASx register update. No ESR update. Debug counter update occurs for instructions (1)-(5) as appropriate. No debug counter or event updates for instruction (6). **Note:** In this case debug counter updates can occur for instructions (2)-(5) even though the 1st ld/st has a DVC DAC exception. **Note:** In this case if x==y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the “no DACy” case.
	Instruction (1): DVC DACx Instruction (3): DVC DACy Instruction (5): DSI, with or without DAC	Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 3’b010. No ESR update. DSRR0 points to instruction (4). Debug counter update occurs for instructions (1)-(3) as appropriate. No debug counter or event updates for instructions (4)-(6). **Note:** In this case if x==y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the “no DACy” case. **Note:** In this case the 3rd ld/st exception is masked. This behavior is implementation dependent and may differ on other CPUs.
	Instruction (1): DVC DACx Instruction (3): DVC DACy Instruction (5): DTLB Error, with or without DAC	
	Instruction (1): DVC DACx Instruction (3): DVC DACy Instruction (5): DACy or DVC DACy	Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 3’b100. No ESR update. DSRR0 points to instruction (6). Debug counter update occurs for instructions (1)-(5) as appropriate. **Note:** In this case if x==y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the “no DACy” case.

Table 12-3. DAC events and resultant updates, dual-issue case 2

Instruction Sequence: The following pairs dual-issue: (1) load/store (2) alu (3) load/store (4) alu (5) alu (6) load/store	**Event(s)**	**Result**
	Instruction (1): DTLB Error, no DAC	Take DTLB exception, no DBSR update, update MASx registers for 1st load/store instruction. Update ESR.
	Instruction (1): DSI, no DAC	Take DSI exception, no DBSR update, no MASx register update. Update ESR.
	Instruction (1): DTLB Error, with DACx	Take Debug exception, DBSR update setting DACx and IDE, DAC_OFST set to 3’b000. DSRR0 points to instruction (1). No MASx register update. No ESR update.
	Instruction (1): DSI, with DACx	
	Instruction (1): DACx	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 3’b000. DSRR0 points to instruction (2). No MASx register update. No ESR update.
	Instruction (1): DVC DACx No other exceptions	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 3’b101. DSRR0 points to instruction after instruction (6). No MASx register update. No ESR update.Debug counter update occurs for instructions (1)-(6) as appropriate.
	Instruction (1): DVC DACx Instruction (3): DTLB Error, with or without DAC	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 3’b001. DSRR0 points to instruction (3). no MASx register update. No ESR update. Debug counter update occurs for instructions (1)-(2) as appropriate. No debug counter or event updates for instructions (3)-(6). **Note:** In this case the 2nd ld/st exception is masked. This behavior is implementation dependent and may differ on other CPUs.
	Instruction (1): DVC DACx Instruction (3): DSI, with or without DAC	
	Instruction (1): DVC DACx Instruction (3): DACy	Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 3’b010. DSRR0 points to instruction (4). Debug counter update occurs for instructions (1)-(3) as appropriate. No debug counter or event updates for instructions (4)-(6). **Note:** In this case if x==y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the “no DACy” case.
	Instruction (1): DVC DACx Instruction (3): DVC DACy	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 3’b101. DSRR0 points to instruction (7). No MASx register update. No ESR update.Debug counter update occurs for instructions (1)-(6) as appropriate. **Note:** In this case if x==y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the “no DACy” case.

Table 12-3. DAC events and resultant updates, dual-issue case 2 (continued)

Instruction Sequence: The following pairs dual-issue: (1) load/store (2) alu (3) load/store (4) alu (5) alu (6) load/store	**Event(s)**	**Result**
	Instruction (1): DVC DACx Instruction (3): DVC DACy Instruction (6): DSI, with or without DAC	Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 3’b010. No ESR update. DSRR0 points to instruction (4). Debug counter update occurs for instructions (1)-(3) as appropriate. No debug counter or event updates for instruction (4). **Note:** In this case if x==y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the “no DACy” case. **Note:** In this case the 3rd ld/st exception is masked. This behavior is implementation dependent and may differ on other CPUs.
	Instruction (1): DVC DACx Instruction (3): DVC DACy Instruction (6): DTLB Error, with or without DAC	
	Instruction (1): DVC DACx Instruction (3): DVC DACy Instruction (6): DACy or DVC DACy	Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 3’b101. No ESR update. DSRR0 points to instruction (7). Debug counter update occurs for instructions (1)-(6) as appropriate. No debug counter or event updates for instruction (7). **Note:** In this case if x==y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the “no DACy” case.

Table 12-4. DAC events and Resultant Updates, Dual-issue case 3

Instruction Sequence: The following pairs dual-issue: (1) load/store (2) alu (3) alu (4) alu (5) load/store (6) alu	**Event(s)**	**Result**
	Instruction (1): DTLB Error, no DAC	Take DTLB exception, no DBSR update, update MASx registers for 1st load/store instruction. Update ESR.
	Instruction (1): DSI, no DAC	Take DSI exception, no DBSR update, no MASx register update. Update ESR.
	Instruction (1): DTLB Error, with DACx	Take Debug exception, DBSR update setting DACx and IDE, DAC_OFST set to 3'b000. DSRR0 points to instruction (1). No MASx register update. No ESR update.
	Instruction (1): DSI, with DACx	
	Instruction (1): DACx	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 3'b000. DSRR0 points to instruction (2). No MASx register update. No ESR update.
	Instruction (1): DVC DACx No other exceptions	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 3'b100. DSRR0 points to instruction (6). No MASx register update. No ESR update.Debug counter update occurs for instructions (1)-(5) as appropriate. No debug counter or event updates for instruction (6).
	Instruction (1): DVC DACx Instruction (5): DTLB Error, with or without DAC	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 3'b011. DSRR0 points to instruction (5). no MASx register update. No ESR update. Debug counter update occurs for instructions (1)-(4) as appropriate. No debug counter or event updates for instructions (5)-(6). **Note:** Note: in this case the 2nd ld/st exception is masked. This behavior is implementation dependent and may differ on other CPUs.
	Instruction (1): DVC DACx Instruction (5): DSI, with or without DAC	

Table 12-4. DAC events and Resultant Updates, Dual-issue case 3 (continued)

Instruction Sequence: The following pairs dual-issue: (1) load/store (2) alu (3) alu (4) alu (5) load/store (6) alu	**Event(s)**	**Result**
	Instruction (1): DVC DACx Instruction (5): DACy	Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 3'b100. DSRR0 points to instruction (6). Debug counter update occurs for instructions (1)-(5) as appropriate. No debug counter or event updates for instruction (6). **Note:** Note: in this case if x==y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the "no DACy" case.
	Instruction (1): DVC DACx Instruction (5): DVC DACy	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 3'b100. DSRR0 points to instruction (6). No MASx register update. No ESR update.Debug counter update occurs for instructions (1)-(5) as appropriate. No debug counter or event updates for instruction (6) **Note:** Note: in this case if x==y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the "no DACy" case.

Table 12-5. DAC events and resultant updates, dual-issue case 4

Instruction Sequence: The following pairs dual-issue: (1) load/store (2) alu (3) load/store (4) alu (5) alu (6) alu	**Event(s)**	**Result**
	Instruction (1): DTLB Error, no DAC	Take DTLB exception, no DBSR update, update MASx registers for 1st load/store instruction. Update ESR.
	Instruction (1): DSI, no DAC	Take DSI exception, no DBSR update, no MASx register update. Update ESR.
	Instruction (1): DTLB Error, with DACx	Take Debug exception, DBSR update setting DACx and IDE, DAC_OFST set to 3'b000. DSRR0 points to instruction (1). No MASx register update. No ESR update.
	Instruction (1): DSI, with DACx	

Table 12-5. DAC events and resultant updates, dual-issue case 4 (continued)

Instruction Sequence: The following pairs dual-issue: (1) load/store (2) alu (3) load/store (4) alu (5) alu (6) alu	**Event(s)**	**Result**
	Instruction (1): DACx	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 3'b000. DSRR0 points to instruction (2). No MASx register update. No ESR update.
	Instruction (1): DVC DACx No other exceptions	Take Debug exception, DBSR update setting DACx, DAC_OFST set to 3'b011. DSRR0 points to instruction (5). No MASx register update. No ESR update. Debug counter update occurs for instructions (1)-(4) as appropriate. No debug counter or event updates for instructions (5)-(6)

	Instruction (1):
	DVC DACx
	Instruction (3):
	DTLB Error, with or without DAC
		Take Debug exception, DBSR update setting DACx, DAC_OFST set to 3'b001. DSRR0 points to instruction (3). no MASx register update. No ESR update. Debug counter update occurs for instructions (1)-(2) as appropriate. No debug counter or event updates for instructions (3)-(6).
		Note: In this case the 2nd ld/st exception is masked. This behavior is implementation dependent and may differ on other CPUs.

	Instruction (1):
	DVC DACx
	Instruction (3):
	DSI, with or without DAC	

	Instruction (1):
	DVC DACx
	Instruction (3):
	DACy
		Take Debug exception, DBSR update setting DACx, DACy. DAC_OFST set to 3'b010. DSRR0 points to instruction (4). Debug counter update occurs for instructions (1)-(3) as appropriate. No debug counter or event updates for instructions (4)-(6).
		Note: In this case if x==y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the "no DACy" case.

	Instruction (1):
	DVC DACx
	Instruction (3):
	DVC DACy
		Take Debug exception, DBSR update setting DACx, DAC_OFST set to 3'b011. DSRR0 points to instruction (5). No MASx register update. No ESR update. Debug counter update occurs for instructions (1)-(4) as appropriate. No debug counter or event updates for instructions (5)-(6).
		Note: In this case if x==y, then the resultant state of DBSR and DSRR0 may be indistinguishable from the "no DACy" case.

12.2.3 Linked Instruction Address and Data Address Compare event

Data Address Compare debug events may be 'linked' with an Instruction Address Compare event by setting the DAC1LNK and/or DAC2LNK control bits in DBCR2 to further refine when a Data Address Compare debug event is generated. DAC1 may be linked with IAC1, and DAC2 (when not used as a mask or range bounds register) may be linked with IAC3. When linked, a DAC1 (or DAC2) debug event occurs when the same instruction that generates the DAC1 (or DAC2) ‘hit’ also generates an IAC1 (or IAC3) ‘hit’. When linked, the IAC1 (or IAC3) event is not recorded in the Debug Status register, regardless of whether a corresponding DAC1 (or DAC2) event occurs, or whether the IAC1 (or IAC3) event enable is set.

When enabled and execution of a load or store class instruction results in a data access with an address that meets the criteria specified in the DBCR0, DBCR2, DBCR4, DAC1, DAC2, DVC1, and DVC2 Registers, and the instruction also meets the criteria for generating an Instruction Address Compare event, a Linked Data Address Compare debug event occurs. This event can occur and be recorded in DBSR regardless of the setting of MSR_{DE}. The normal DAC1 and DAC2 status bits in the DBSR are used for recording these events. The IAC1 and IAC3 status bits are not set if the corresponding Instruction Address Compare register is linked.

Linking is enabled using control bits in DBCR2. If Data Address Compare debug events are used to control or modify operation of the Debug Counter, linking is also available, even though DBCR0 may not have enabled IAC or DAC events. Also, Instruction Address Compare events that are linked may still affect the Debug Counter (if enabled to), thus may be used to either trigger a counter, or be counted, in contrast to being blocked from affecting the DBSR.

NOTE

Linked DAC events will not be recorded or counted if a load multiple word or store multiple word type instruction is interrupted prior to completion by a critical input or external input interrupt.

12.2.4 Trap debug event

A Trap debug event (TRAP) occurs if Trap debug events are enabled ($DBCR0_{TRAP}$=1), a Trap instruction (**tw**, **twi**) is executed, and the conditions specified by the instruction for the trap are met. This event can occur and be recorded in DBSR regardless of the setting of MSR_{DE}. When a Trap debug event occurs, the $DBSR_{TRAP}$ bit is set to 1 to record the debug exception.

12.2.5 Branch Taken debug event

A Branch Taken debug event (BRT) 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, the $DBSR_{BRT}$ bit is set to 1 to record the debug exception, and the address of the branch instruction will be recorded in DSRR0.

12.2.6 Instruction Complete debug event

An Instruction Complete debug event (ICMP) occurs if Instruction Complete debug events are enabled ($DBCR0_{ICMP}$=1), execution of any instruction is completed, and MSR_{DE}=1 or $DBCR0_{EDM}$=1. If execution of an instruction is suppressed due to the instruction causing some other exception that is enabled to generate an interrupt, then the attempted execution of that instruction does not cause an

Instruction Complete debug event. The ***sc*** instruction does not fall into the category of an instruction whose execution is suppressed, since the instruction actually executes and then generates a System Call interrupt. In this case, the Instruction Complete debug exception will also be set. When an Instruction Complete debug event is recognized, $DBSR_{ICMP}$ is set to 1 to record the debug exception and the address of the next instruction to be executed will be recorded in DSRR0.

Instruction Complete debug events are not recognized if MSR_{DE}=0 and $DBCR0_{EDM}$=0 at the time of execution of the instruction, thus $DBSR_{IDE}$ is not generally set by an ICMP debug event.

One circumstance may cause the $DBSR_{ICMP}$ and $DBSR_{IDE}$ bits to be set. This occurs when a EFPU Round exception occurs. Since the instruction is by definition completed (SRR0 points to the following instruction), this interrupt takes higher priority than the Debug interrupt so as not to be lost, and $DBSR_{IDE}$ is set to indicate the imprecise recognition of a Debug interrupt. In this case, the Debug interrupt will be taken with SRR0 pointing to the instruction following the instruction that generated the EFPU Round exception, and DSRR0 will point to the Round exception handler. In addition to occurring when $DBCR0_{IDM}$=1, this circumstance can also occur when $DBCR0_{EDM}$=1.

NOTE

Instruction complete debug events are not generated by the execution of an instruction that sets MSR_{DE} to '1' while $DBCR0_{ICMP}$=1, nor by the execution of an instruction that sets $DBCR0_{ICMP}$ to '1' while MSR_{DE}=1 or $DBCR0_{EDM}$=1.

12.2.7 Interrupt Taken debug event

An Interrupt Taken debug event (IRPT) occurs if Interrupt Taken debug events are enabled ($DBCR0_{IRPT}$=1) and a non-critical interrupt occurs. Only non-critical class interrupts cause an Interrupt Taken debug event. This event can occur and be recorded in DBSR regardless of the setting of MSR_{DE}. When an Interrupt Taken debug event occurs, the $DBSR_{IRPT}$ bit is set to 1 to record the debug exception. The value saved in DSRR0 will be the address of the non-critical interrupt handler.

12.2.8 Critical Interrupt Taken debug event

A Critical Interrupt Taken debug event (CIRPT) occurs if Critical Interrupt Taken debug events are enabled ($DBCR0_{CIRPT}$=1) and a critical interrupt (other than a Debug interrupt when the Debug 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, the $DBSR_{CIRPT}$ bit is set to 1 to record the debug exception. The value saved in DSRR0 will be the address of the critical interrupt handler. Note that this debug event should not normally be enabled unless the Debug APU is also enabled to avoid corruption of CSRR0/1.

12.2.9 Return debug event

A Return debug event (RET) occurs if Return debug events are enabled ($DBCR0_{RET}$=1) and an attempt is made to execute an **rfi** or **se_rfi** instruction. This event can occur and be recorded in DBSR regardless of the setting of MSR_{DE}. When a Return debug event occurs, the $DBSR_{RET}$ bit is set to 1 to record the debug exception.

If MSR_{DE}=0 and $DBCR0_{EDM}$=0 at the time of the execution of the **rfi** or **se_rfi** (i.e. before the MSR is updated by the **rfi** or **se_rfi**), then $DBSR_{IDE}$ is also set to 1 to record the imprecise debug event.

If MSR_{DE}=1 at the time of the execution of the **rfi** or **se_rfi**, a Debug interrupt will occur provided there exists no higher priority exception that is enabled to cause an interrupt. Debug Save/Restore Register 0 will be set to the address of the **rfi** or **se_rfi** instruction.

12.2.10 Critical Return debug event

A Critical Return debug event (CRET) occurs if Critical Return debug events are enabled ($DBCR0_{CRET}$=1) and an attempt is made to execute an **rfci** or **se_rfci** instruction. This event can occur and be recorded in DBSR regardless of the setting of MSR_{DE}. When a Critical Return debug event occurs, the $DBSR_{CRET}$ bit is set to 1 to record the debug exception.

If MSR_{DE}=0 and $DBCR0_{EDM}$=0 at the time of the execution of the **rfci** or **se_rfci** (i.e. before the MSR is updated by the **rfci** or **se_rfci**), then $DBSR_{IDE}$ is also set to 1 to record the imprecise debug event.

If MSR_{DE}=1 at the time of the execution of the **rfci** or **se_rfci**, a Debug interrupt will occur provided there exists no higher priority exception that is enabled to cause an interrupt. Debug Save/Restore Register 0 will be set to the address of the **rfci** or **se_rfci** instruction. Note that this debug event should not normally be enabled unless the Debug APU is also enabled to avoid corruption of CSRR0/1.

12.2.11 Debug Counter debug event

A Debug Counter debug event (DCNT1, DCNT2) occurs if Debug Counter debug events are enabled ($DBCR0_{DCNT1}$=1 or $DBCR0_{DCNT2}$=1), a Debug Counter is enabled, and a Counter decrements to zero. This event can occur and be recorded in DBSR regardless of the setting of MSR_{DE}. When a Debug Counter debug event occurs, $DBSR_{DCNT\{1,2\}}$ is set to ‘1’ to record the debug exception.

12.2.12 External debug event

An External debug event (DEVT1, DEVT2) occurs if External debug events are enabled ($DBCR0_{DEVT1}$=1 or $DBCR0_{DEVT2}$=1), and the respective **p_devt1** or **p_devt2** input signal transitions to the asserted state. This event can occur and be recorded in DBSR regardless of the setting of MSR_{DE}. When an External debug event occurs, $DBSR_{DEVT\{1,2\}}$ is set to ‘1’ to record the debug exception.

12.2.13 Unconditional debug event

An Unconditional debug event (UDE) occurs when the Unconditional Debug Event (**p_ude**) input transitions to the asserted state, and either $DBCR0_{IDM}$=1 or $DBCR0_{EDM}$=1. The Unconditional debug event is the only debug event that does not have a corresponding enable bit for the event in DBCR0. This event can occur and be recorded in DBSR regardless of the setting of MSR_{DE}. When an Unconditional debug event occurs, the $DBSR_{UDE}$ bit is set to ‘1’ to record the debug exception.

12.3 Debug registers

This section describes debug-related registers that are software accessible. These registers are intended for use by special debug tools and debug software, not by general application code.

Access to these registers (other than DBSR) by software is conditioned by the External Debug Mode control bit ($DBCR0_{EDM}$/$EDBCR0_{EDM}$) and the settings of debug control register DBERC0, which can be set by the hardware debug port. If $DBCR0_{EDM}$ is set and if the bit in DBERC0 corresponding to the resource is cleared, software is prevented from modifying debug register values other than in DBSR, since the resource is not “owned” by software. Software always has ownership of DBSR. Execution of a **mtspr** instruction targeting a debug register or register field not “owned” by software will not cause modifications to occur, and no exception will be signaled. In addition, since the external debugger hardware may be manipulating debug register values, the state of these registers or register fields not “owned” by software is not guaranteed to be consistent if accessed (read) by software with a **mfspr** instruction, other than the $DBCR0_{EDM}$ bit itself and the DBERC0 register. Hardware always has full access to all registers and all register fields through the OnCE register access mechanism, and it is up to the debug firmware to properly implement modifications to these registers with read-modify-write operations to implement any control sharing with software. Settings in DBERC0 should be considered by the debug firmware in order to preserve software settings of control registers as appropriate when hardware modifications to the debug registers is performed.

12.3.1 Debug address and value registers

Instruction Address Compare registers IAC1-8 are used to hold instruction addresses for address comparison purposes. In addition, IAC2 and IAC4 hold mask information for IAC1 and IAC3 respectively and IAC6 and IAC8 hold mask information for IAC5 and IAC7 respectively, when *Address Bit Match* compare modes are selected. Note that when performing instruction address compares, the low order two address bits of the instruction address and the corresponding IAC register are ignored for BookE instruction pages, and the low order bit of the instruction address and the corresponding IAC register is ignored for VLE instruction pages.

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

Data Value Compare registers DVC1 and DVC2 are used to hold data values for data comparison purposes. DVC1 and DVC2 are 64-bit registers. Data value comparisons are used to qualify Data Address compare debug events. DVC1 is associated with DAC1, and DVC2 is associated with DAC2. The most significant byte of the DVC1(2) register (labeled B0 in Figure 12-2) corresponds to the byte data value transferred to/from memory byte offset 0, 8, ..., and the least significant byte of the register (labeled B7 in Figure 12-2) corresponds to byte offset 7, F, ... When enabled for performing data value comparisons, each enabled byte in DVC1(2) is compared with the memory value transferred on the corresponding active byte lane of the data memory interface to determine if a match occurs. Inactive byte lanes do not participate in the comparison, they are implicitly masked. Table 14-11 shows active byte lanes for data transfers. Software must also program the DVC1(2) register byte positions based on the endian mode and alignment of the access. Misaligned accesses are not fully supported, since the data address and data value comparisons are only performed on the initial access in the case of a misaligned access; thus, accesses that cross a 64-bit boundary cannot be fully matched. For address and size combinations that involve two transfers, only the initial transfer is used for data address and value matching. DVC1 and DVC2 may be read or written using **mtspr** and **mfspr** instructions. All 64-bits of the GPR will be accessed, regardless of the value of the MSR_{SPE} bit.

B0	B1	B2	B3
0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31
B4	B5	B6	B7
32 33 34 35 36 37 38 39	40 41 42 43 44 45 46 47	48 49 50 51 52 53 54 55	56 57 58 59 60 61 62 63

SPR - 318 (DVC1), 319 (DVC2); Read/Write; Reset - Unaffected

Figure 12-2. DVC1, DVC2 registers

12.3.2 Debug Counter register (DBCNT)

The Debug Counter register (DBCNT) contains two 16-bit counters (CNT1 and CNT2) that can be configured to operate independently, or 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 will operate regardless of whether counters are enabled to generate debug exceptions. When a count value reaches zero, a Debug Count event is signaled, and a Debug event can be generated (if enabled). Upon reaching zero, the counter(s) are frozen. A debug counter signals an event on the transition from a value of one to a final value of zero. Loading a value of zero into the counter prevents the counter from counting. The Debug Counter is configured by the contents of Debug Control Register 3. The DBCNT register is shown in Figure 12-3.

CNT1	CNT2
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

SPR - 562; Read/Write; Reset - Unaffected

Figure 12-3. DBCNT register

Refer to Section 12.3.3.4, Debug Control Register 3 (DBCR3), for more information about updates to the DBCNT register. Certain caveats exist on how the DBCNT and DBCR3 register are modified when one or more counters are enabled.

12.3.3 Debug Control and Status registers

Debug Control Registers (DBCR0-6 and DBERC0) are used to enable debug events, reset the processor, control timer operation during debug events, and set the debug mode of the processor. The Debug Status register (DBSR) records debug exceptions while Internal Debug Mode is enabled.

e200z759n3 requires that a context synchronizing instruction follow a **mtspr** DBCR0-6 or DBSR to ensure that any alterations enabling/disabling debug events are effective. The context synchronizing instruction may or may not be affected by the alteration. Typically, an **isync** instruction is used to create a synchronization boundary beyond which it can be guaranteed that the newly written control values are in effect.

For watchpoint generation and counter operation, configuration settings contained in DBCR1-5 are used, even though the corresponding event(s) may be disabled (via DBCR0) from setting DBSR flags.

12.3.3.1 Debug Control Register 0 (DBCR0)

Debug Control Register 0 is used to enable debug modes and controls which debug events are allowed to set DBSR or EDBSR0 flags. e200z759n3 adds some implementation specific bits to this register, as seen in Figure 12-4.

EDM	IDM	RST		ICMP	BRT	IRPT	TRAP	IAC1	IAC2	IAC3	IAC4	DAC1		DAC2		RET	IAC5	IAC6	IAC7	IAC8	DEVT1	DEVT2	DCNT1	DCNT2	CIRPT	CRET	0				FT
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

SPR - 308; Read/Write; Reset[1] - 0x0

Figure 12-4. DBCR0 register

[1] DBCR0$_{EDM}$ is affected by **j_trst_b** or **m_por** assertion, and remains reset while in the Test_Logic_Reset state, but is not affected by **p_reset_b**. All other bits are reset by processor reset **p_reset_b** if DBCR0$_{EDM}$=0, as well as unconditionally by **m_por**. If DBCR0$_{EDM}$=1, DBERC0 masks off hardware-owned resources (other than RST) from reset by **p_reset_b,** and only software-owned resources indicated by DBERC0 and the DBCR0$_{RST}$ field will be reset by **p_reset_b**. The DBCR0$_{RST}$ field will always be reset by **p_reset_b** regardless of the value of DBCR0$_{EDM}$.

Table 12-6 provides field descriptions for Debug Control Register 0.

Table 12-6. DBCR0 field descriptions

Bit(s)	Name	Description
0	EDM	External Debug Mode. This bit is read-only by software. 0 External debug mode disabled. Internal debug events not mapped into external debug events. 1 External debug mode enabled. Events will not cause the CPU to vector to interrupt code. Software is not permitted to write to debug registers {DBCR0-6, DBCNT, IAC1-8, DAC1-2} unless permitted by settings in EDBCR0. Hardware-owned events will set status bits in EDBSR0. When external debug mode is enabled, hardware-owned resources in debug registers are not affected by processor reset **p_reset_b**. This allows the debugger to set up hardware debug events that remain active across a processor reset.
		Programming Notes: It is recommended that debug status bits in the Debug Status Registers be cleared before disabling external debug mode to avoid any internal imprecise debug interrupts. Software may use this bit to determine if external debug has control over the debug registers. The hardware debugger must set the EDM bit to ‘1’ before other bits in this register (and other debug registers) may be altered. On the initial setting of this bit to ‘1’, all other bits are unchanged. This bit is only writable through the OnCE port.
1	IDM	Internal Debug Mode 0 Debug exceptions are disabled. Debug events do not affect DBSR. 1 Debug exceptions are enabled. Enabled debug events owned by software will update the DBSR. If MSR$_{DE}$=1, the occurrence of a debug event, or the recording of an earlier debug event in the Debug Status Register when MSR$_{DE}$ was cleared, will cause a Debug interrupt.

Table 12-6. DBCR0 field descriptions (continued)

Bit(s)	Name	Description
2:3	RST	Reset Control 00 No function 01 **p_dbrstc[1]** pin asserted by Debug Reset Control. Allows external device to initiate processor or system reset 10 **p_dbrstc[0]** pin asserted by Debug Reset Control. Allows external device to initiate processor or system reset. 11 Reserved
4	ICMP	Instruction Complete Debug Event Enable 0 ICMP debug events are disabled 1 ICMP debug events are enabled
5	BRT	Branch Taken Debug Event Enable 0 BRT debug events are disabled 1 BRT debug events are enabled
6	IRPT	Interrupt Taken Debug Event Enable 0 IRPT debug events are disabled 1 IRPT debug events are enabled
7	TRAP	Trap Taken Debug Event Enable 0 TRAP debug events are disabled 1 TRAP debug events are enabled
8	IAC1	Instruction Address Compare 1 Debug Event Enable 0 IAC1 debug events are disabled 1 IAC1 debug events are enabled
9	IAC2	Instruction Address Compare 2 Debug Event Enable 0 IAC2 debug events are disabled 1 IAC2 debug events are enabled
10	IAC3	Instruction Address Compare 3 Debug Event Enable 0 IAC3 debug events are disabled 1 IAC3 debug events are enabled
11	IAC4	Instruction Address Compare 4 Debug Event Enable 0 IAC4 debug events are disabled 1 IAC4 debug events are enabled
12:13	DAC1	Data Address Compare 1 Debug Event Enable 00 DAC1 debug events are disabled 01 DAC1 debug events are enabled only for store-type data storage accesses 10 DAC1 debug events are enabled only for load-type data storage accesses 11 DAC1 debug events are enabled for load-type or store-type data storage accesses
14:15	DAC2	Data Address Compare 2 Debug Event Enable 00 DAC2 debug events are disabled 01 DAC2 debug events are enabled only for store-type data storage accesses 10 DAC2 debug events are enabled only for load-type data storage accesses 11 DAC2 debug events are enabled for load-type or store-type data storage accesses
16	RET	Return Debug Event Enable 0 RET debug events are disabled 1 RET debug events are enabled

Table 12-6. DBCR0 field descriptions (continued)

Bit(s)	Name	Description
17	IAC5	Instruction Address Compare 5 Debug Event Enable 0 IAC5 debug events are disabled 1 IAC5 debug events are enabled
18	IAC6	Instruction Address Compare 6 Debug Event Enable 0 IAC6 debug events are disabled 1 IAC6 debug events are enabled
19	IAC7	Instruction Address Compare 7 Debug Event Enable 0 IAC7 debug events are disabled 1 IAC7 debug events are enabled
20	IAC8	Instruction Address Compare 8 Debug Event Enable 0 IAC8 debug events are disabled 1 IAC8 debug events are enabled
21	DEVT1	External Debug Event 1 Enable 0 DEVT1 debug events are disabled 1 DEVT1 debug events are enabled
22	DEVT2	External Debug Event 2 Enable 0 DEVT2 debug events are disabled 1 DEVT2 debug events are enabled
23	DCNT1	Debug Counter 1 Debug Event Enable 0 Counter 1 debug events are disabled 1 Counter 1 debug events are enabled
24	DCNT2	Debug Counter 2 Debug Event Enable 0 Counter 2 debug events are disabled 1 Counter 2 debug events are enabled
25	CIRPT	Critical Interrupt Taken Debug Event Enable 0 CIRPT debug events are disabled 1 CIRPT debug events are enabled
26	CRET	Critical Return Debug Event Enable 0 CRET debug events are disabled 1 CRET debug events are enabled
27:30	—	Reserved
31	FT	Freeze Timers on Debug Event 0 TimeBase Timers are unaffected by set DBSR/EDBSR0 bits 1 Disable clocking of TimeBase timers if any DBSR bit is set (any EDBSR0 bit set if DBCR0$_{FT}$ owned by hardware) except MRR or CNT1TRG

12.3.3.2 Debug Control Register 1 (DBCR1)

Debug Control Register 1 is used to configure Instruction Address Compare operation. The DBCR1 register is shown in Figure 12-5.

0 1	2 3	4 5	6 7	8 9	10 11 12 13 14 15	16 17	18 19	20 21	22 23	24 25	26 27 28 29 30 31
IAC1US	IAC1ER	IAC2US	IAC2ER	IAC12M	0	IAC3US	IAC3ER	IAC4US	IAC4ER	IAC34M	0

SPR - 309; Read/Write; Reset[1] - 0x0

Figure 12-5. DBCR1 register

[1] Reset by processor reset **p_reset_b** if DBCR0$_{EDM}$=0, as well as unconditionally by **m_por**. If DBCR0$_{EDM}$=1, DBERC0 masks off hardware-owned resources from reset by **p_reset_b** and only software-owned resources indicated by DBERC0 will be reset by **p_reset_b.**

Table 12-7 provides field descriptions for Debug Control Register 1.

Table 12-7. DBCR1 field descriptions

Bits	Name	Description
0:1	IAC1US	Instruction Address Compare 1 User/Supervisor Mode 00 IAC1 debug events not affected by MSR$_{PR}$ 01 Reserved 10 IAC1 debug events can only occur if MSR$_{PR}$=0 (Supervisor mode) 11 IAC1 debug events can only occur if MSR$_{PR}$=1. (User mode)
2:3	IAC1ER	Instruction Address Compare 1 Effective/Real Mode 00 IAC1 debug events are based on effective address 01 Unimplemented in e200z759n3 (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
4:5	IAC2US	Instruction Address Compare 2 User/Supervisor Mode 00 IAC2 debug events not affected by MSR$_{PR}$ 01 Reserved 10 IAC2 debug events can only occur if MSR$_{PR}$=0 (Supervisor mode) 11 IAC2 debug events can only occur if MSR$_{PR}$=1. (User mode)
6:7	IAC2ER	Instruction Address Compare 2 Effective/Real Mode 00 IAC2 debug events are based on effective address 01 Unimplemented in e200z759n3 (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
8:9	IAC12M	Instruction Address Compare 1/2 Mode 00 Exact address compare. IAC1 debug events can only occur if the address of the instruction fetch is equal to the value specified in IAC1. IAC2 debug events can only occur if the address of the instruction fetch is equal to the value specified in IAC2. 01 Address bit match. IAC1 debug events can occur only if the address of the instruction fetch, ANDed with the contents of IAC2 are equal to the contents of IAC1, also ANDed with the contents of IAC2. IAC2 debug events do not occur. IAC1US and IAC1ER settings are used. 10 Inclusive address range compare. IAC1 debug events can occur only if the address of the instruction fetch is greater than or equal to the value specified in IAC1 and less than the value specified in IAC2. IAC2 debug events do not occur. IAC1US and IAC1ER settings are used. 11 Exclusive address range compare. IAC1 debug events can occur only if the address of the instruction fetch is less than the value specified in IAC1 or is greater than or equal to the value specified in IAC2. IAC2 debug events do not occur. IAC1US and IAC1ER settings are used.
10:15	—	Reserved

Table 12-7. DBCR1 field descriptions (continued)

Bits	Name	Description
16:17	IAC3US	Instruction Address Compare 3 User/Supervisor Mode 00 IAC3 debug events not affected by MSR_{PR} 01 Reserved 10 IAC3 debug events can only occur if MSR_{PR}=0 (Supervisor mode) 11 IAC3 debug events can only occur if MSR_{PR}=1 (User mode)
18:19	IAC3ER	Instruction Address Compare 3 Effective/Real Mode 00 IAC3 debug events are based on effective address 01 Unimplemented in e200z759n3 (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
20:21	IAC4US	Instruction Address Compare 4 User/Supervisor Mode 00 IAC4 debug events not affected by MSR_{PR} 01 Reserved 10 IAC4 debug events can only occur if MSR_{PR}=0 (Supervisor mode). 11 IAC4 debug events can only occur if MSR_{PR}=1. (User mode)
22:23	IAC4ER	Instruction Address Compare 4 Effective/Real Mode 00 IAC4 debug events are based on effective address 01 Unimplemented in e200z759n3 (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
24:25	IAC34M	Instruction Address Compare 3/4 Mode 00 Exact address compare. IAC3 debug events can only occur if the address of the instruction fetch is equal to the value specified in IAC3. IAC4 debug events can only occur if the address of the instruction fetch is equal to the value specified in IAC4. 01 Address bit match. IAC3 debug events can occur only if the address of the instruction fetch, ANDed with the contents of IAC4 are equal to the contents of IAC3, also ANDed with the contents of IAC4. IAC4 debug events do not occur. IAC3US and IAC3ER settings are used. 10 Inclusive address range compare. IAC3 debug events can occur only if the address of the instruction fetch is greater than or equal to the value specified in IAC3 and less than the value specified in IAC4. IAC4 debug events do not occur. IAC3US and IAC3ER settings are used. 11 Exclusive address range compare. IAC3 debug events can occur only if the address of the instruction fetch is less than the value specified in IAC3 or is greater than or equal to the value specified in IAC4. IAC4 debug events do not occur. IAC3US and IAC3ER settings are used.
26:31	—	Reserved

12.3.3.3 Debug Control Register 2 (DBCR2)

Debug Control Register 2 is used to configure Data Address Compare and Data Value Compare operation.The DBCR2 register is shown in Figure 12-6.

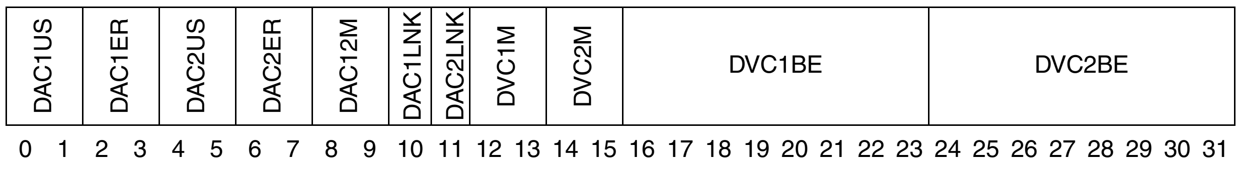


Figure 12-6. DBCR2 Register

SPR - 310; Read/Write; Reset[1] - 0x0

Figure 12-6. DBCR2 Register

[1] Reset by processor reset **p_reset_b** if DBCR0$_{EDM}$=0, as well as unconditionally by **m_por**. If DBCR0$_{EDM}$=1, DBERC0 masks off hardware-owned resources from reset by **p_reset_b** and only software-owned resources indicated by DBERC0 will be reset by **p_reset_b**.

Table 12-8 provides field descriptions for Debug Control Register 2.

Table 12-8. DBCR2 field descriptions

Bit(s)	Name	Description
0:1	DAC1US	Data Address Compare 1 User/Supervisor Mode 00 DAC1 debug events not affected by MSR$_{PR}$ 01 Reserved 10 DAC1 debug events can only occur if MSR$_{PR}$=0 (Supervisor mode) 11 DAC1 debug events can only occur if MSR$_{PR}$=1. (User mode)
2:3	DAC1ER	Data Address Compare 1 Effective/Real Mode 00 DAC1 debug events are based on effective address 01 Unimplemented in e200z759n3 (Book E real address compare), no match can occur 10 DAC1 debug events are based on effective address and can only occur if MSR$_{DS}$=0 11 DAC1 debug events are based on effective address and can only occur if MSR$_{DS}$=1
4:5	DAC2US	Data Address Compare 2 User/Supervisor Mode. 00 DAC2 debug events not affected by MSR$_{PR}$ 01 Reserved 10 DAC2 debug events can only occur if MSR$_{PR}$=0 (Supervisor mode) 11 DAC2 debug events can only occur if MSR$_{PR}$=1. (User mode)
6:7	DAC2ER	Data Address Compare 2 Effective/Real Mode 00 DAC2 debug events are based on effective address 01 Unimplemented in e200z759n3 (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
8:9	DAC12M	Data Address Compare 1/2 Mode 00 Exact address compare. DAC1 debug events can only occur if the address of the data access is equal to the value specified in DAC1. DAC2 debug events can only occur if the address of the data access is equal to the value specified in DAC2. 01 Address bit match. DAC1 debug events can occur only if the address of the data access ANDed with the contents of DAC2, are equal to the contents of DAC1 also ANDed with the contents of DAC2. DAC2 debug events do not occur. DAC1US and DAC1ER settings are used. 10 Inclusive address range compare. DAC1 debug events can occur only if the address of the data access is greater than or equal to the value specified in DAC1 and less than the value specified in DAC2. DAC2 debug events do not occur. DAC1US and DAC1ER settings are used. 11 Exclusive address range compare. DAC1 debug events can occur only if the address of the data access is less than the value specified in DAC1 or is greater than or equal to the value specified in DAC2. DAC2 debug events do not occur. DAC1US and DAC1ER settings are used.

Table 12-8. DBCR2 field descriptions (continued)

Bit(s)	Name	Description
10	DAC1LNK	Data Address Compare 1 Linked 0 No effect 1 DAC1 debug events are linked to IAC1 debug events. IAC1 debug events do not affect DBSR When linked to IAC1, DAC1 debug events are conditioned based on whether the instruction also generated an IAC1 debug event
11	DAC2LNK	Data Address Compare 2 Linked 0 No effect eopg events are linked to IAC3 debug events. IAC3 debug events do not affect DBSR When linked to IAC3, DAC2 debug events are conditioned based on whether the instruction also generated an IAC3 debug event. DAC2 can only be linked if DAC12M specifies *Exact Address Compare* since DAC2 debug events are not generated in the other compare modes.
12:13	DVC1M	Data Value Compare 1 Mode When DBCR4$_{DVC1C}$=0: 00 DAC1 debug events not affected by data value compares. 01 DAC1 debug events can only occur when all bytes specified in the DVC1BE field match the corresponding data byte values for active byte lanes of the memory access. 10 DAC1 debug events can only occur when any byte specified in the DVC1BE field matches the corresponding data byte value for active byte lanes of the memory access. 11 DAC1 debug events can only occur when all bytes specified in the DVC1BE field within at least one of the halfwords of the data value of the memory access matches the corresponding DVC1 value. **Note:** Inactive byte lanes of the memory access are automatically masked. When DBCR4$_{DVC1C}$=1: 00 Reserved 01 DAC1 debug events can only occur when any byte specified in the DVC1BE field does not match the corresponding data byte value for active byte lanes of the memory access. If all active bytes match, then no event will be generated. 10 DAC1 debug events can only occur when all bytes specified in the DVC1BE field do not match the corresponding data byte values for active byte lanes of the memory access. If any active byte match occurs, no event will be generated. 11 Reserved **Note:** Inactive byte lanes of the memory access are automatically masked.

Table 12-8. DBCR2 field descriptions (continued)

Bit(s)	Name	Description
14:15	DVC2M	Data Value Compare 2 Mode When DBCR4$_{DVC2C}$=0: 00 DAC2 debug events not affected by data value compares. 01 DAC2 debug events can only occur when all bytes specified in the DVC2BE field match the corresponding data byte values for active byte lanes of the memory access. 10 DAC2 debug events can only occur when any byte specified in the DVC2BE field matches the corresponding data byte value for active byte lanes of the memory access. 11 DAC2 debug events can only occur when all bytes specified in the DVC2BE field within at least one of the halfwords of the data value of the memory access matches the corresponding DVC2 value. **Note:** Inactive byte lanes of the memory access are automatically masked. When DBCR4$_{DVC2C}$=1: 00 Reserved 01 DAC2 debug events can only occur when any byte specified in the DVC2BE field does not match the corresponding data byte value for active byte lanes of the memory access. If all active bytes match, then no event will be generated. 10 DAC2 debug events can only occur when all bytes specified in the DVC2BE field do not match the corresponding data byte values for active byte lanes of the memory access. If any active byte match occurs, no event will be generated. 11 Reserved **Note:** Inactive byte lanes of the memory access are automatically masked.
16:23	DVC1BE	Data Value Compare 1 Byte Enables Specifies which bytes in the aligned doubleword value associated with the memory access are compared to the corresponding bytes in DVC1. Inactive byte lanes of a memory access smaller than 64-bits are automatically masked by hardware. If all bits in the DVC1BE field are clear, then a match will occur regardless of the data. Misaligned accesses that cross a doubleword boundary are not fully supported. 1*xxxxxxx* - Byte lane 0 is enabled for comparison with the value in bits 0:7 of DVC1. *x*1*xxxxxx* - Byte lane 1 is enabled for comparison with the value in bits 8:15 of DVC1. *xx*1*xxxxx* - Byte lane 2 is enabled for comparison with the value in bits 16:23 of DVC1. *xxx*1*xxxx* - Byte lane 3 is enabled for comparison with the value in bits 24:31 of DVC1. *xxxx*1*xxx* - Byte lane 4 is enabled for comparison with the value in bits 32:39 of DVC1. *xxxxx*1*xx* - Byte lane 5 is enabled for comparison with the value in bits 40:47 of DVC1. *xxxxxx*1*x* - Byte lane 6 is enabled for comparison with the value in bits 48:55 of DVC1. *xxxxxxx*1 - Byte lane 7 is enabled for comparison with the value in bits 56:63 of DVC1.
24:31	DVC2BE	Data Value Compare2 Byte Enables Specifies which bytes in the aligned doubleword value associated with the memory access are compared to the corresponding bytes in DVC2. Inactive byte lanes of a memory access smaller than 64-bits are automatically masked by hardware. If all bits in the DVC1BE field are clear, then a match will occur regardless of the data. Misaligned accesses that cross a doubleword boundary are not fully supported. 1*xxxxxxx* - Byte lane 0 is enabled for comparison with the value in bits 0:7 of DVC2. *x*1*xxxxxx* - Byte lane 1 is enabled for comparison with the value in bits 8:15 of DVC2. *xx*1*xxxxx* - Byte lane 2 is enabled for comparison with the value in bits 16:23 of DVC2. *xxx*1*xxxx* - Byte lane 3 is enabled for comparison with the value in bits 24:31 of DVC2. *xxxx*1*xxx* - Byte lane 4 is enabled for comparison with the value in bits 32:39 of DVC2. *xxxxx*1*xx* - Byte lane 5 is enabled for comparison with the value in bits 40:47 of DVC2. *xxxxxx*1*x* - Byte lane 6 is enabled for comparison with the value in bits 48:55 of DVC2. *xxxxxxx*1 - Byte lane 7 is enabled for comparison with the value in bits 56:63 of DVC2.

12.3.3.4 Debug Control Register 3 (DBCR3)

Debug Control Register 3 (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 that the corresponding events do not need to be (and probably should not be) enabled in DBCR0.* The IAC1–IAC4 and DAC1–DAC2 control fields in DBCR0 are ignored for counter operations, and the control fields in DBCR3 determine when counting is enabled. DBCR1 and DBCR2 control fields are also used to determine the configuration of IAC1–4 and DAC1–2 operation for counting, even though corresponding events may be disabled via DBCR0. Multiple count-enabled events that occur during execution of an instruction will typically cause only a single decrement of a counter. As an example, if more than one IAC or DAC register hits and is enabled for counting, only a single count will occur per counter. During **lmw** and **stmw** instructions, multiple DACx hits could occur. If the instruction is not interrupted prior to completion, a single decrement of a counter will occur. Note that if the counters are operating independently, both may count for the same instruction.

The Debug Counter Register (DBCNT) is configured by $DBCR3_{CONFIG}$ to operate either as separate 16-bit Counter 1 and Counter 2, or as a combined 32-bit counter (using control bits in DBCR3 for Counter 1). Counters are enabled whenever any of their respective Count Enable event control bits are set to ‘1’ and either $DBCR0_{IDM}$ or $DBCR0_{EDM}$ is set to ‘1’. Counters are frozen during a hardware “debug session” (see Section 12.4.2, OnCE introduction). Counter 1 may be configured to count down on a number of different debug events. Counter 2 is also configurable to count down on instruction complete, instruction or data address compare events, and external events.

Special capability is provided for Counter 1 to be triggered to begin counting down by a subset of events (IAC1, IAC3, DAC1R, DAC1W, DEVT1, DEVT2, and Counter 2). When one or more of the Counter 1 trigger bits is set (IAC1T1, IAC3T1, DAC1RT1, DAC1WT1, DEVT1T1, DEVT2T1, CNT2T1), Counter 1 is frozen until at least one of the triggering events occurs, and is then enabled to begin operation. Depending on the trigger source, if it is enabled for counting, the trigger event may be counted. Triggering status for Counter 1 is provided in the Debug Status Register or External Debug Status Register 0. Triggering mode is enabled by a **mtspr** *DBCR3*, which sets one or more of the trigger enable bits and also enables Counter 1. Once set, the trigger can be re-armed by clearing the $DBSR_{CNT1TRG}$ or $EDBSR0_{CNT1TRG}$ status bit.

Most combinations of enables do not make sense and should be avoided. As an example, if $DBCR3_{ICMP}$ is set for Counter 1, no other count enable should be set for Counter 1. Conversely, multiple Instruction Address Compare count enables are allowed to be set and may be useful.

Due to instruction pipelining issues and other constraints, most combinations of events are not supported for event counting. Only the following combinations are intended to be used, and other combinations are not supported:

- Any combination of IAC[1–4]
- Any combination of DAC[1–2] including linking
- Any combination of DEVT[1–2]
- Any combination of IRPT, RET

Limited support is provided for the following combinations:

- Any combination of IAC[1–4] with DAC[1–2] (linked or unlinked). Note that these combinations may be reported in an imprecise fashion, with $DBSR_{IDE}$ set in such cases.

Due to pipelining and detection of IAC events early in the pipeline and DAC events late in the pipeline, no guarantee is made on the exact instruction boundary that a debug exception will be generated when IAC and DAC events are combined for counting. This also applies to the case where Counter 1 is being triggered by Counter 2, and a combination of IAC and DAC events are being enabled for the counters, even if only one of these types is enabled for a particular counter. In general, when an IAC event logically follows closely behind a DAC event (within several instructions), it cannot be recognized immediately since the DAC event has not necessarily been generated in the pipeline at the time the IAC is seen, and thus the counter may not decrement to zero for the IAC event until after the instruction with the IAC (and perhaps several additional instructions) has proceeded down the execution pipeline. The instruction boundary where the debug exception is actually generated in this case will typically follow the IAC by up to several instructions.

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

If Counter 2 is being used to trigger Counter 1, Counter 2 events should not normally be enabled in DBCR, and will not be blocked.

NOTE

Multiple IAC or DAC events will not be counted during a load multiple word or store multiple word type instruction, and no count will occur if either is interrupted by a critical input or external input interrupt prior to completion.

DBCR3 is a e200z759n3 implementation specific register and is shown in Figure 12-7.

DEVT1C1	DEVT2C1	ICMPC1	IAC1C1	IAC2C1	IAC3C1	IAC4C1	DAC1RC1	DAC1WC1	DAC2RC1	DAC2WC1	IRPTC1	RETC1	DEVT1C2	DEVT2C2	ICMPC2	IAC1C2	IAC2C2	IAC3C2	IAC4C2	DAC1RC2	DAC1WC2	DAC2RC2	DAC2WC2	DEVT1T1	DEVT2T1	IAC1T1	IAC3T1	DAC1RT1	DAC1WT1	CNT2T1	CONFIG
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

SPR - 561; Read/Write; Reset[1] - 0x0

Figure 12-7. DBCR3 register

[1] Reset by processor reset **p_reset_b** if $DBCR0_{EDM}$=0, as well as unconditionally by **m_por**. If $DBCR0_{EDM}$=1, DBERC0 masks off hardware-owned resources from reset by **p_reset_b** and only software-owned resources indicated by DBERC0 will be reset by **p_reset_b.**

Table 12-9 provides field descriptions for Debug Control Register 3.

Table 12-9. DBCR3 field descriptions

Bit(s)	Name	Description
0	DEVT1C1	External Debug Event 1 Count 1 Enable 0 Counting DEVT1 debug events by Counter 1 is disabled 1 Counting DEVT1 debug events by Counter 1 is enabled

Table 12-9. DBCR3 field descriptions (continued)

Bit(s)	Name	Description
1	DEVT2C1	External Debug Event 2 Count 1 Enable 0 Counting DEVT2 debug events by Counter 1 is disabled 1 Counting DEVT2 debug events by Counter 1 is enabled
2	ICMPC1	Instruction Complete Debug Event Count 1 Enable 0 Counting ICMP debug events by Counter 1 is disabled 1 Counting ICMP debug events by Counter 1 is enabled **Note:** ICMP events are masked by MSR_{DE}=0 when operating in Internal Debug Mode.
3	IAC1C1	Instruction Address Compare 1 Debug Event Count 1 Enable 0 Counting IAC1 debug events by Counter 1 is disabled 1 Counting IAC1 debug events by Counter 1 is enabled
4	IAC2C1	Instruction Address Compare2 Debug Event Count 1 Enable 0 Counting IAC2 debug events by Counter 1 is disabled 1 Counting IAC2 debug events by Counter 1 is enabled
5	IAC3C1	Instruction Address Compare 3 Debug Event Count 1 Enable 0 Counting IAC3 debug events by Counter 1 is disabled 1 Counting IAC3 debug events by Counter 1 is enabled
6	IAC4C1	Instruction Address Compare 4 Debug Event Count 1 Enable 0 Counting IAC4 debug events by Counter 1 is disabled 1 Counting IAC4 debug events by Counter 1 is enabled
7	DAC1RC1	Data Address Compare 1 Read Debug Event Count 1 Enable[1] 0 Counting DAC1R debug events by Counter 1 is disabled 1 Counting DAC1R debug events by Counter 1 is enabled
8	DAC1WC1	Data Address Compare 1 Write Debug Event Count 1 Enable[1] 0 Counting DAC1W debug events by Counter 1 is disabled 1 Counting DAC1W debug events by Counter 1 is enabled
9	DAC2RC1	Data Address Compare 2 Read Debug Event Count 1 Enable[1] 0 Counting DAC2R debug events by Counter 1 is disabled 1 Counting DAC2R debug events by Counter 1 is enabled
10	DAC2WC1	Data Address Compare 2 Write Debug Event Count 1 Enable[1] 0 Counting DAC2W debug events by Counter 1 is disabled 1 Counting DAC2W debug events by Counter 1 is enabled
11	IRPTC1	Interrupt Taken Debug Event Count 1 Enable 0 Counting IRPT debug events by Counter 1 is disabled 1 Counting IRPT debug events by Counter 1 is enabled
12	RETC1	Return Debug Event Count 1 Enable 0 Counting RET debug events by Counter 1 is disabled 1 Counting RET debug events by Counter 1 is enabled
13	DEVT1C2	External Debug Event 1 Count 2 Enable 0 Counting DEVT1 debug events by Counter 2 is disabled 1 Counting DEVT1 debug events by Counter 2 is enabled
14	DEVT2C2	External Debug Event 2 Count 2 Enable 0 Counting DEVT2 debug events by Counter 2 is disabled 1 Counting DEVT2 debug events by Counter 2 is enabled

Table 12-9. DBCR3 field descriptions (continued)

Bit(s)	Name	Description
15	ICMPC2	Instruction Complete Debug Event Count 2 Enable 0 Counting ICMP debug events by Counter 2 is disabled 1 Counting ICMP debug events by Counter 2 is enabled **Note:** ICMP events are masked by MSR_{DE}=0 when operating in Internal Debug Mode.
16	IAC1C2	Instruction Address Compare 1 Debug Event Count 2 Enable 0 Counting IAC1 debug events by Counter 2 is disabled 1 Counting IAC1 debug events by Counter 2 is enabled
17	IAC2C2	Instruction Address Compare2 Debug Event Count 2 Enable 0 Counting IAC2 debug events by Counter 2 is disabled 1 Counting IAC2 debug events by Counter 2 is enabled
18	IAC3C2	Instruction Address Compare 3 Debug Event Count 2 Enable 0 Counting IAC3 debug events by Counter 2 is disabled 1 Counting IAC3 debug events by Counter 2 is enabled
19	IAC4C2	Instruction Address Compare 4 Debug Event Count 2 Enable 0 Counting IAC4 debug events by Counter 2 is disabled 1 Counting IAC4 debug events by Counter 2 is enabled
20	DAC1RC2	Data Address Compare 1 Read Debug Event Count 2 Enable[1] 0 Counting DAC1R debug events by Counter 2 is disabled 1 Counting DAC1R debug events by Counter 2 is enabled
21	DAC1WC2	Data Address Compare 1 Write Debug Event Count 2 Enable[1] 0 Counting DAC1W debug events by Counter 2 is disabled 1 Counting DAC1W debug events by Counter 2 is enabled
22	DAC2RC2	Data Address Compare 2 Read Debug Event Count 2 Enable[1] 0 Counting DAC2R debug events by Counter 2 is disabled 1 Counting DAC2R debug events by Counter 2 is enabled
23	DAC2WC2	Data Address Compare 2 Write Debug Event Count 2 Enable[1] 0 Counting DAC2W debug events by Counter 2 is disabled 1 Counting DAC2W debug events by Counter 2 is enabled
24	DEVT1T1	External Debug Event 1 Trigger Counter 1 Enable 0 No effect 1 A DEVT1 debug event will trigger Counter 1 operation
25	DEVT2T1	External Debug Event 2 Trigger Counter 1 Enable 0 No effect 1 A DEVT2 debug event will trigger Counter 1 operation
26	IAC1T1	Instruction Address Compare 1 Trigger Counter 1 Enable 0 No effect 1 An IAC1 debug event will trigger Counter 1 operation
27	IAC3T1	Instruction Address Compare 3 Trigger Counter 1 Enable 0 No effect 1 An IAC3 debug event will trigger Counter 1 operation
28	DAC1RT1	Data Address Compare 1 Read Trigger Counter 1 Enable 0 No effect 1 A DAC1R debug event will trigger Counter 1 operation

Table 12-9. DBCR3 field descriptions (continued)

Bit(s)	Name	Description
29	DAC1WT1	Data Address Compare 1 Write Trigger Counter 1 Enable 0 No effect 1 A DAC1W debug event will trigger Counter 1 operation
30	CNT2T1	Debug Counter 2 Trigger Counter 1 Enable 0 No effect 1 Counter 2 decrementing to a value of '0' will trigger Counter 1 operation
31	CONFIG	Debug Counter Configuration 0 Counter 1 and Counter 2 are independent counters 1 Counter 1 and Counter 2 are concatenated into a single 32-bit counter. The event count control bits for Counter 1 are used and the event count control bits for Counter 2 are ignored.

[1] If the DACx field in DBCR0 is set to restrict events to only reads or only writes, only those events will be counted if enabled in DBCR3. In general, DAC events should be disabled in DBCR0.

NOTE

Updates to the DBCR0, DBSR, DBCR3, and DBCNT registers should be performed carefully if the counters are currently enabled for counting events. For these cases, it is possible that the instruction that updates the counters or control over the counters will cause one or more counter events to occur (DCNT1, DCNT2, CNT1TRG), even if the result of the instruction is to modify the counter value or control value to a state where counter events would not be expected to occur. This is due to the pipelined nature of the counter and control operation. As an example, if a counter was enabled to count ICMP events, and MSR_{DE} = '1', and the value of the counter is '1' prior to execution of a **mtspr** instruction that is loading the counter with a different value, a counter event will be generated following completion of the **mtspr**, even though the counter ends up being loaded with a new value. At the end of the **mtspr** instruction, a debug event will be posted, but the counter value will be that of the newly written count value. In addition, no decrement of the new counter value is performed at the completion of a **mtspr** instruction that modifies a counter, regardless of whether a debug event is generated based on the old counter value. To avoid this, it is recommended that the DBCNT and DBCR3 values be modified only when no possibility of a counter related debug event on the **mtspr** instruction is possible. Modifying DBCR0 to affect counter event enabling/disabling may have similar issues, as may modifying the $DBSR_{CNT1TRG}$ bit.

As another example, if a counter was enabled to count ICMP events, and MSR_{DE} = ‘1’, and the value of the counter is ‘1’ prior to execution of a **mtspr** instruction that is loading DBCR3 with a different value, a counter event may be generated following completion of the **mtspr**, even though DBCR3 ends up being loaded with a new value that is disabling the particular event from being counted. At the end of the **mtspr** instruction, a debug event will be posted, but the DBCR3 value will reflect 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 the $DBSR_{CNT1TRG}$ bit.

12.3.3.5 Debug Control Register 4 (DBCR4)

Debug Control Register 4 is used to extend data address and value compare matching functionality. DBCR4 is shown in Figure 12-8.

0	1	2	3	4–15	16–19	20–23	24–31
0	DVC1C	0	DVC2C	0	DAC1XM	DAC2XM	0

SPR - 563; Read/Write; Reset[1] - 0x0

Figure 12-8. DBCR4 register

[1] DBCR4 is reset by processor reset **p_reset_b** if $DBCR0_{EDM}$=0, as well as unconditionally by **m_por**. If $DBCR0_{EDM}$=1, DBERC0 masks off hardware-owned resources from reset by **p_reset_b** and only software-owned resources indicated by DBERC0 will be reset by **p_reset_b**.

Table 12-10 provides field descriptions for Debug Control Register 4.

Table 12-10. DBCR4 field descriptions

Bit(s)	Name	Description
0	—	Reserved
1	DVC1C	Data Value Compare 1 Control 0 Normal DVC1 operation. 1 Inverted polarity DVC1 operation DVC1C controls whether DVC1 data value comparisons utilize the normal BookE operation, or an alternate “inverted compare” operation. In inverted polarity mode, data value compares perform a not-equal comparison. See details in the DBCR2 register definition
2	—	Reserved
3	DVC2C	Data Value Compare 2 Control 0 Normal DVC2 operation. 1 Inverted polarity DVC2 operation DVC2C controls whether DVC2 data value comparisons utilize the normal BookE operation, or an alternate “inverted compare” operation. In inverted polarity mode, data value compares perform a not-equal comparison. See details in the DBCR2 register definition

Table 12-10. DBCR4 field descriptions (continued)

Bit(s)	Name	Description
4:15	—	Reserved
16:19	DAC1XM	Data Address Compare 1 Extended Mask Control 0000 No additional masking when DBCR2[DAC12M] = 00 0001 - 1100 Exact Match Bit Mask. Number of low order bits masked in DAC1 when comparing the storage address with the value in DAC1 for exact address compare (DBCR2[DAC12M] = 00). Ranges up to 4 KB are supported. 1101 - 1111 Reserved DAC1XM allows for binary power of 2 address range compares for DAC1 without requiring the use of DAC2.
20:23	DAC2XM	Data Address Compare 2 Extended Mask Control 0000 No additional masking when DBCR2[DAC12M] = 00 0001 - 1100 Exact Match Bit Mask. Number of low order bits masked in DAC2 when comparing the storage address with the value in DAC2 for exact address compare (DBCR2[DAC12M] = 00). Ranges up to 4 KB are supported. 1101 - 1111 Reserved DAC2XM allows for binary power of 2 address range compares for DAC2 without requiring the use of DAC1.
24:31	—	Reserved

12.3.3.6 Debug Control Register 5 (DBCR5)

Debug Control Register 5 is used to configure Instruction Address Compare operation for IAC5-8. The DBCR5 register is shown in Figure 12-9.

IAC5US	IAC5ER	IAC6US	IAC6ER	IAC56M	0	IAC7US	IAC7ER	IAC8US	IAC8ER	IAC78M	0
0 1	2 3	4 5	6 7	8 9	10 11 12 13 14 15	16 17	18 19	20 21	22 23	24 25	26 27 28 29 30 31

SPR - 564; Read/Write; Reset[1] - 0x0

Figure 12-9. DBCR5 Register

[1] Reset by processor reset **p_reset_b** if DBCR0$_{EDM}$=0, as well as unconditionally by **m_por**. If DBCR0$_{EDM}$=1, DBERC0 masks off hardware-owned resources from reset by **p_reset_b** and only software-owned resources indicated by DBERC0 will be reset by **p_reset_b**.

Table 12-7 provides field descriptions for Debug Control Register 5.

Table 12-11. DBCR5 field descriptions

Bit(s)	Name	Description
0:1	IAC5US	Instruction Address Compare 5 User/Supervisor Mode 00 IAC5 debug events not affected by MSR_{PR} 01 Reserved 10 IAC5 debug events can only occur if MSR_{PR}=0 (Supervisor mode) 11 IAC5 debug events can only occur if MSR_{PR}=1. (User mode)
2:3	IAC5ER	Instruction Address Compare 5 Effective/Real Mode 00 IAC5 debug events are based on effective address 01 Unimplemented in e200z759n3 (Book E real address compare), no match can occur 10 IAC5 debug events are based on effective address and can only occur if MSR_{IS}=0 11 IAC5 debug events are based on effective address and can only occur if MSR_{IS}=1
4:5	IAC6US	Instruction Address Compare 6 User/Supervisor Mode 00 IAC6 debug events not affected by MSR_{PR} 01 Reserved 10 IAC6 debug events can only occur if MSR_{PR}=0 (Supervisor mode) 11 IAC6 debug events can only occur if MSR_{PR}=1. (User mode)
6:7	IAC6ER	Instruction Address Compare 6 Effective/Real Mode 00 IAC6 debug events are based on effective address 01 Unimplemented in e200z759n3 (Book E real address compare), no match can occur 10 IAC6 debug events are based on effective address and can only occur if MSR_{IS}=0 11 IAC6 debug events are based on effective address and can only occur if MSR_{IS}=1
8:9	IAC56M	Instruction Address Compare 5/6 Mode 00 Exact address compare. IAC5 debug events can only occur if the address of the instruction fetch is equal to the value specified in IAC5. IAC6 debug events can only occur if the address of the instruction fetch is equal to the value specified in IAC6. 01 Address bit match. IAC5 debug events can occur only if the address of the instruction fetch, ANDed with the contents of IAC6 are equal to the contents of IAC5, also ANDed with the contents of IAC6. IAC6 debug events do not occur. IAC5US and IAC5ER settings are used. 10 Reserved 11 Reserved
10:15	—	Reserved
16:17	IAC7US	Instruction Address Compare 7 User/Supervisor Mode 00 IAC7 debug events not affected by MSR_{PR} 01 Reserved 10 IAC7 debug events can only occur if MSR_{PR}=0 (Supervisor mode) 11 IAC7 debug events can only occur if MSR_{PR}=1 (User mode)
18:19	IAC7ER	Instruction Address Compare 7 Effective/Real Mode 00 IAC7 debug events are based on effective address 01 Unimplemented in e200z759n3 (Book E real address compare), no match can occur 10 IAC7 debug events are based on effective address and can only occur if MSR_{IS}=0 11 IAC7 debug events are based on effective address and can only occur if MSR_{IS}=1
20:21	IAC8US	Instruction Address Compare 8 User/Supervisor Mode 00 IAC8 debug events not affected by MSR_{PR} 01 Reserved 10 IAC8 debug events can only occur if MSR_{PR}=0 (Supervisor mode). 11 IAC8 debug events can only occur if MSR_{PR}=1. (User mode)

Table 12-11. DBCR5 field descriptions (continued)

Bit(s)	Name	Description
22:23	IAC8ER	Instruction Address Compare 8 Effective/Real Mode 00 IAC8 debug events are based on effective address 01 Unimplemented in e200z759n3 (Book E real address compare), no match can occur 10 IAC8 debug events are based on effective address and can only occur if MSR_{IS}=0 11 IAC8 debug events are based on effective address and can only occur if MSR_{IS}=1
24:25	IAC78M	Instruction Address Compare 7/8 Mode 00 Exact address compare. IAC7 debug events can only occur if the address of the instruction fetch is equal to the value specified in IAC7. IAC8 debug events can only occur if the address of the instruction fetch is equal to the value specified in IAC8. 01 Address bit match. IAC7 debug events can occur only if the address of the instruction fetch, ANDed with the contents of IAC8 are equal to the contents of IAC7, also ANDed with the contents of IAC8. IAC8 debug events do not occur. IAC7US and IAC7ER settings are used. 10 Reserved 11 Reserved
26:31	—	Reserved

12.3.3.7 Debug Control Register 6 (DBCR6)

Debug Control Register 6 is used to extend instruction address compare matching functionality. DBCR6 is shown in Figure 12-10.

IAC1XM				IAC2XM				IAC3XM				IAC4XM				IAC5XM				IAC6XM				IAC7XM				IAC8XM			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

SPR - 603; Read/Write; Reset[1] - 0x0

Figure 12-10. Debug Control Register 6 (DBCR6)

[1] DBCR6 is reset by processor reset **p_reset_b** if $DBCR0_{EDM}$=0, as well as unconditionally by **m_por**. If $DBCR0_{EDM}$=1, DBERC0 masks off hardware-owned resources from reset by **p_reset_b** and only software-owned resources indicated by DBERC0 will be reset by **p_reset_b**.

Table 12-12 provides field descriptions for Debug Control Register 6.

Table 12-12. DBCR6 field descriptions

Bit(s)	Name	Description
0:3	IAC1XM	Instruction Address Compare 1 Extended Mask Control 0000 No additional masking when DBCR1[IAC12M] = 00 0001 - 1100 Exact Match Bit Mask. Number of low order bits masked in IAC1 when comparing the storage address with the value in IAC1 for exact address compare (DBCR1[IAC12M] = 00). Ranges up to 4 KB are supported. 1101 - 1111 Reserved IAC1XM allows for binary power of 2 address range compares for IAC1 without requiring the use of IAC2.

Table 12-12. DBCR6 field descriptions (continued)

Bit(s)	Name	Description
4:7	IAC2XM	Instruction Address Compare 2 Extended Mask Control 0000 No additional masking when DBCR1[IAC12M] = 00 0001 - 1100 Exact Match Bit Mask. Number of low order bits masked in IAC2 when comparing the storage address with the value in IAC2 for exact address compare (DBCR1[IAC12M] = 00). Ranges up to 4 KB are supported. 1101 - 1111 Reserved IAC2XM allows for binary power of 2 address range compares for IAC2 without requiring the use of IAC1.
8:11	IAC3XM	Instruction Address Compare 3 Extended Mask Control 0000 No additional masking when DBCR1[IAC34M] = 00 0001 - 1100 Exact Match Bit Mask. Number of low order bits masked in IAC3 when comparing the storage address with the value in IAC3 for exact address compare (DBCR1[IAC34M] = 00). Ranges up to 4 KB are supported. 1101 - 1111 Reserved IAC3XM allows for binary power of 2 address range compares for IAC1 without requiring the use of IAC2.
12:15	IAC4XM	Instruction Address Compare 4 Extended Mask Control 0000 No additional masking when DBCR1[IAC34M] = 00 0001 - 1100 Exact Match Bit Mask. Number of low order bits masked in IAC4 when comparing the storage address with the value in IAC4 for exact address compare (DBCR1[IAC34M] = 00). Ranges up to 4 KB are supported. 1101 - 1111 Reserved IAC4XM allows for binary power of 2 address range compares for IAC4 without requiring the use of IAC3.
16:19	IAC5XM	Instruction Address Compare 5 Extended Mask Control 0000 No additional masking when DBCR5[IAC56M] = 00 0001 - 1100 Exact Match Bit Mask. Number of low order bits masked in IAC5 when comparing the storage address with the value in IAC5 for exact address compare (DBCR5[IAC56M] = 00). Ranges up to 4 KB are supported. 1101 - 1111 Reserved IAC5XM allows for binary power of 2 address range compares for IAC5 without requiring the use of IAC6.
20:23	IAC6XM	Instruction Address Compare 6 Extended Mask Control 0000 No additional masking when DBCR5[IAC56M] = 00 0001 - 1100 Exact Match Bit Mask. Number of low order bits masked in IAC6 when comparing the storage address with the value in IAC6 for exact address compare (DBCR5[IAC56M] = 00). Ranges up to 4 KB are supported. 1101 - 1111 Reserved IAC6XM allows for binary power of 2 address range compares for IAC6 without requiring the use of IAC5.
24:27	IAC7XM	Instruction Address Compare 7 Extended Mask Control 0000 No additional masking when DBCR5[IAC78M] = 00 0001 - 1100 Exact Match Bit Mask. Number of low order bits masked in IAC7 when comparing the storage address with the value in IAC7 for exact address compare (DBCR5[IAC78M] = 00). Ranges up to 4 KB are supported. 1101 - 1111 Reserved IAC7XM allows for binary power of 2 address range compares for IAC7 without requiring the use of IAC8.

Table 12-12. DBCR6 field descriptions (continued)

Bit(s)	Name	Description
28:31	IAC8XM	Instruction Address Compare 8 Extended Mask Control 0000 No additional masking when DBCR5[IAC78M] = 00 0001 - 1100 Exact Match Bit Mask. Number of low order bits masked in IAC8 when comparing the storage address with the value in IAC8 for exact address compare (DBCR5[IAC78M] = 00). Ranges up to 4 KB are supported. 1101 - 1111 Reserved IAC8XM allows for binary power of 2 address range compares for IAC8 without requiring the use of IAC7.

12.3.3.8 Debug Status register (DBSR)

The Debug Status Register (DBSR) contains status on debug events and the most recent processor reset. The Debug Status Register is set via hardware, and read and cleared via software. Bits in the Debug Status Register can be cleared using **mtspr** *DBSR*,*RS*. Clearing is done by writing to the Debug Status Register with a 1 in any bit position that is to be cleared and 0 in all other bit positions. The write data to the Debug Status Register is not direct data, but a mask. A '1' causes the bit to be cleared, and a '0' has no effect. Debug Status bits are set by Debug events only while Internal Debug Mode is enabled ($DBCR0_{IDM}$=1). When debug interrupts are enabled (MSR_{DE}=1 $DBCR0_{IDM}$=1 and $DBCR0_{EDM}$=0, or MSR_{DE}=1, $DBCR0_{IDM}$=1 and $DBCR0_{EDM}$=1 and software is allocated resource(s) via DBERC0), a set bit in DBSR other than MRR, VLES, or CNT1TRG will cause a debug interrupt to be generated. The debug interrupt handler is responsible for clearing DBSR bits prior to returning to normal execution. The PowerISA VLE APU adds the $DBSR_{VLES}$ status bit to indicate debug events occurring due to a PowerISA VLE instruction. When resource sharing is enabled, ($DBCR0_{EDM}$=1 and $DBERC0_{IDM}$=1), only software-owned resources may be modified by software, and status bits associated with hardware-owned resources will not be set by hardware in DBSR. The DBSR register is shown in Figure 12-11.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
IDE	UDE	MRR		ICMP	BRT	IRPT	TRAP	IAC1	IAC2	IAC3	IAC4-8	DAC1R	DAC1W	DAC2R	DAC2W	RET	0				DEVT1	DEVT2	DCNT1	DCNT2	CIRPT	CRET	VLES	DAC_OFST			CNT1TRG

SPR - 304; Read/Write; Reset[1] - 0x1000_0000

Figure 12-11. Debug Status Register (DBSR)

[1] Reset by processor reset **p_reset_b** if $DBCR0_{EDM}$=0, as well as unconditionally by **m_por**. If $DBCR0_{EDM}$=1, DBERC0 masks off hardware-owned resources from reset by **p_reset_b** and only software-owned resources indicated by DBERC0 will be reset by **p_reset_b.** $DBSR_{MRR}$ is always updated by **p_reset_b** however.

Table 12-13 provides field descriptions for the Debug Status register.

Table 12-13. DBSR field descriptions

Bit(s)	Name	Description
0	IDE	Imprecise Debug Event Set to 1 if MSR$_{DE}$=0 and DBCR0$_{IDM}$=1 and a debug event causes its respective Debug Status Register bit to be set to 1. It may also be set to ‘1’ if an imprecise debug event occurs due to a DAC event on a load or store that is terminated with error, or if an ICMP event occurs in conjunction with a EFPU FP round exception.
1	UDE	Unconditional Debug Event Set to 1 if an Unconditional debug event occurred.
2:3	MRR	Most Recent Reset. 00 No reset occurred since these bits were last cleared by software 01 A hard reset occurred since these bits were last cleared by software 10 Reserved 11 Reserved
4	ICMP	Instruction Complete Debug Event Set to 1 if an Instruction Complete debug event occurred.
5	BRT	Branch Taken Debug Event Set to 1 if an Branch Taken debug event occurred.
6	IRPT	Interrupt Taken Debug Event Set to 1 if an Interrupt Taken debug event occurred.
7	TRAP	Trap Taken Debug Event Set to 1 if a Trap Taken debug event occurred.
8	IAC1	Instruction Address Compare 1 Debug Event Set to 1 if an IAC1 debug event occurred.
9	IAC2	Instruction Address Compare 2 Debug Event Set to 1 if an IAC2 debug event occurred.
10	IAC3	Instruction Address Compare 3 Debug Event Set to 1 if an IAC3 debug event occurred.
11	IAC4-8	Instruction Address Compare 4-8 Debug Event Set to 1 if an IAC4, IAC5, IAC6, IAC7, or IAC8 debug event occurred.
12	DAC1R	Data Address Compare 1 Read Debug Event Set to 1 if a read-type DAC1 debug event occurred while DBCR0$_{DAC1}$=0b10 or DBCR0$_{DAC1}$=0b11
13	DAC1W	Data Address Compare 1 Write Debug Event Set to 1 if a write-type DAC1 debug event occurred while DBCR0$_{DAC1}$=0b01 or DBCR0$_{DAC1}$=0b11
14	DAC2R	Data Address Compare 2 Read Debug Event Set to 1 if a read-type DAC2 debug event occurred while DBCR0$_{DAC2}$=0b10 or DBCR0$_{DAC2}$=0b11
15	DAC2W	Data Address Compare 2 Write Debug Event Set to 1 if a write-type DAC2 debug event occurred while DBCR0$_{DAC2}$=0b01 or DBCR0$_{DAC2}$=0b11
16	RET	Return Debug Event Set to 1 if a Return debug event occurred
17:20	—	Reserved

Table 12-13. DBSR field descriptions (continued)

Bit(s)	Name	Description
21	DEVT1	External Debug Event 1 Debug Event Set to 1 if a DEVT1 debug event occurred
22	DEVT2	External Debug Event 2 Debug Event Set to 1 if a DEVT2 debug event occurred
23	DCNT1	Debug Counter 1 Debug Event Set to 1 if a DCNT1 debug event occurred
24	DCNT2	Debug Counter 2 Debug Event Set to 1 if a DCNT2 debug event occurred
25	CIRPT	Critical Interrupt Taken Debug Event Set to 1 if a Critical Interrupt Taken debug event occurred.
26	CRET	Critical Return Debug Event Set to 1 if a Critical Return debug event occurred
27	VLES	VLE Status Set to 1 if an ICMP, BRT, TRAP, RET, CRET, IAC, or DAC debug event occurred on a PowerISA VLE Instruction. Undefined for IRPT, CIRPT, DEVT[1,2], DCNT[1,2], and UDE events
28:30	DAC_OFST	Data Address Compare Offset Indicates offset-1 of saved DSRR0 value from the address of the load or store instruction that took a DAC Debug exception, unless a simultaneous DTLB or DSI error occurs, in which case this field is set to 3‘b000 and DBSR$_{IDE}$ is set to 1. Normally set to 3‘b000 by a non-DVC DAC. A DVC DAC may set this field to any value.
31	CNT1TRG	Counter 1 Triggered Set to 1 if Debug Counter 1 is triggered by a trigger event.

12.3.4 Debug External Resource Control register (DBERC0)

The Debug External Resource Control register (DBERC0) controls resource allocation when DBCR0$_{EDM}$ is set to ‘1’. DBERC0 provides a mechanism for the hardware debugger to share certain debug resources with software. Individual resources are allocated based on the settings of DBERC0 when DBCR0$_{EDM}$=1. DBERC0 settings are ignored when DBCR0$_{EDM}$=0.

Hardware-owned resources that generate debug events update EDBSR0 instead of DBSR and cause entry into debug mode if the event is not masked in EDBSRMSK0, while software-owned resources that generate debug events if DBCR0$_{IDM}$=1 update DBSR, causing debug interrupts to occur if MSR$_{DE}$=1. DBERC0 is controlled via the OnCE port hardware, and is read-only to software.

The DBSR status register is always owned by software. Debug status bits in DBSR are set by software-owned debug events only while Internal Debug Mode is enabled. When debug interrupts are enabled (MSR$_{DE}$=1 DBCR0$_{IDM}$=1 and DBCR0$_{EDM}$=0, or MSR$_{DE}$=1, DBCR0$_{IDM}$=1 and DBCR0$_{EDM}$=1 and software is allocated resource(s) via DBERC0), a set bit in DBSR by an event that is software-owned (other than MRR, DAC_OFST, CNT1TRG, or VLES) will cause a debug interrupt to be generated.

Debug status bits in EDBSR0 are set by hardware-owned debug events only while External Debug Mode is enabled ($DBCR0_{EDM}$=1). When $DBCR0_{EDM}$=1, a set bit in EDBSR0 by an event that is hardware-owned (other than IDE, DAC_OFST, CNT1TRG, or VLES) will cause entry into debug mode.

If $DBCR0_{EDM}$=1, DBSR status bits corresponding to hardware-owned debug events are masked from being set by hardware.

Software-owned resources may be modified by software, but only the corresponding control bits in DBCR0-6 are affected by execution of a **mtspr**, thus only a portion of these registers may be affected, depending on the allocation settings in DBERC0. The debug interrupt handler is still responsible for clearing DBSR bits for software-owned resources prior to returning to normal execution. Hardware always has full access to all registers and register fields through the OnCE register access mechanism, and it is up to the debug firmware to properly implement modifications to these registers with read-modify-write operations to implement any control sharing with software. Settings in DBERC0 should be considered by the debug firmware in order to preserve software settings of control and status registers as appropriate when hardware modifications to the debug registers is performed.

The DBERC0 register is shown in Figure 12-12.

0	IDM	RST	UDE	ICMP	BRT	IRPT	TRAP	IAC1	IAC2	IAC3	IAC4	DAC1	0	DAC2	0	RET	IAC5	IAC6	IAC7	IAC8	DEVT1	DEVT2	DCNT1	DCNT2	CIRPT	CRET	BKPT	DQM	0		FT
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

SPR - 569; Read-only by Software; Reset - Unaffected by **p_reset_b**, cleared by **m_por** or while in the test-logic-reset OnCE controller state

Figure 12-12. DBERC0 register

Table 12-13 provides field descriptions for the Debug External Resource Control Register. Note that DBERC0 controls are disabled when $DBCR0_{EDM}$=0.

Table 12-14. DBERC0 field descriptions

Bit(s)	Name	Description
0	—	Reserved
1	IDM	Internal Debug Mode control 0 Internal Debug mode may not be enabled by software. $DBCR0_{IDM}$ is owned exclusively by hardware. **mtspr** DBCR0-6 or DBCNT is always ignored. No resource sharing occurs, regardless of the settings of other fields in DBERC0. Hardware exclusively owns all resources. 1 Internal Debug mode may be enabled by software. $DBCR0_{IDM}$ is owned by software. $DBCR0_{IDM}$ is software readable/writable. When $DBERC0_{IDM}$=1, software writes to hardware-owned bits in DBCR0-6 and DBCNT via **mtspr** are ignored.
2	RST	Reset Field Control 0 $DBCR0_{RST}$ owned exclusively by hardware debug. No **mtspr** access by software to $DBCR0_{RST}$ field. 1 $DBCR0_{RST}$ accessible by software debug. $DBCR0_{RST}$ is software readable/writable.

Table 12-14. DBERC0 field descriptions (continued)

Bit(s)	Name	Description
3	UDE	Unconditional Debug Event 0 Event owned by hardware debug. 1 Event owned by software debug.
4	ICMP	Instruction Complete Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to DBCR0$_{ICMP}$ field. 1 Event owned by software debug. DBCR0$_{ICMP}$ is software readable/writable.
5	BRT	Branch Taken Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to DBCR0$_{BRT}$ field. 1 Event owned by software debug. DBCR0$_{BRT}$ is software readable/writable.
6	IRPT	Interrupt Taken Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to DBCR0$_{IRPT}$ field. 1 Event owned by software debug. DBCR0$_{IRPT}$ is software readable/writable.
7	TRAP	Trap Taken Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to DBCR0$_{TRAP}$ field. 1 Event owned by software debug. DBCR0$_{TRAP}$ is software readable/writable.
8	IAC1	Instruction Address Compare 1 Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to IAC1 control and status fields. 1 Event owned by software debug. IAC1 control fields are software readable/writable.
9	IAC2	Instruction Address Compare 2 Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to IAC2 control and status fields. 1 Event owned by software debug. IAC2 control fields are software readable/writable.
10	IAC3	Instruction Address Compare 3 Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to IAC3 control and status fields. 1 Event owned by software debug. IAC3 control fields are software readable/writable.
11	IAC4	Instruction Address Compare 4 Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to IAC4 control and status fields. 1 Event owned by software debug. IAC4 control fields are software readable/writable.
12	DAC1	Data Address Compare 1 Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to DAC1 control and status fields. 1 Event owned by software debug. DAC1 control fields are software readable/writable.
13	—	Reserved
14	DAC2	Data Address Compare 2 Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to DAC2 control and status fields. 1 Event owned by software debug. DAC2 control fields are software readable/writable.
15	—	Reserved
16	RET	Return Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to DBCR0$_{RET}$ field. 1 Event owned by software debug. DBCR0$_{RET}$ is software readable/writable.

Table 12-14. DBERC0 field descriptions (continued)

Bit(s)	Name	Description
17	IAC5	Instruction Address Compare 5 Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to IAC5 control and status fields. 1 Event owned by software debug. IAC5 control fields are software readable/writable.
18	IAC6	Instruction Address Compare 6 Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to IAC6 control and status fields. 1 Event owned by software debug. IAC6 control fields are software readable/writable.
19	IAC7	Instruction Address Compare 7 Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to IAC7 control and status fields. 1 Event owned by software debug. IAC7 control fields are software readable/writable.
20	IAC8	Instruction Address Compare 8 Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to IAC8 control and status fields. 1 Event owned by software debug. IAC8 control are software readable/writable.
21	DEVT1	External Debug Event Input 1 Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to $\text{DBCR0}_{\text{DEVT1}}$ field. 1 Event owned by software debug. $\text{DBCR0}_{\text{DEVT1}}$ is software readable/writable.
22	DEVT2	External Debug Event Input 2 Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to $\text{DBCR0}_{\text{DEVT2}}$ field. 1 Event owned by software debug. $\text{DBCR0}_{\text{DEVT2}}$ is software readable/writable.
23	DCNT1	Debug Counter 1 Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to Counter1 control and status fields. 1 Event owned by software debug. Counter1 control and status fields are software readable/writable.
24	DCNT2	Debug Counter 2 Debug Event 0 Event owned by hardware debug.No **mtspr** access by software to Counter2 control and status fields. 1 Event owned by software debug. Counter2 control and status fields are software readable/writable.
25	CIRPT	Critical Interrupt Taken Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to $\text{DBCR0}_{\text{CIRPT}}$ field. 1 Event owned by software debug. $\text{DBCR0}_{\text{CIRPT}}$ is software readable/writable.
26	CRET	Critical Return Debug Event 0 Event owned by hardware debug. No **mtspr** access by software to $\text{DBCR0}_{\text{CRET}}$ field. 1 Event owned by software debug. $\text{DBCR0}_{\text{CRET}}$ is software readable/writable.
27	BKPT	Breakpoint Instruction Debug Control 0 Breakpoint owned by hardware debug. Execution of a bkpt instruction (all 0's opcode) results in entry into debug mode. 1 Breakpoint owned by software debug. Execution of a bkpt instruction (all 0's opcode) results in illegal instruction exception.

Table 12-14. DBERC0 field descriptions (continued)

Bit(s)	Name	Description
28	DQM	Data Acquisition Messaging Registers 0 DEVENT$_{\text{DQTAG}}$ and DDAM register are exclusively owned by hardware debug. No **mtspr** access by software to DEVENT$_{\text{DQTAG}}$ field or DDAM register. Attempted access by software is ignored. 1 DEVENT$_{\text{DQTAG}}$ and DDAM register are owned by software. Software has read/write access to DEVENT$_{\text{DQTAG}}$ field and DDAM register.
29:30	—	Reserved
31	FT	Freeze Timer Debug Control 0 DBCR0$_{\text{FT}}$ owned by hardware debug. No access by software. 1 DBCR0$_{\text{FT}}$ owned by software debug. DBCR0$_{\text{FT}}$ is software readable/writable.

Table 12-15 shows which resources are controlled by DBERC0 settings.

Table 12-15. DBERC0 resource control

DBCR0$_{\text{EDM}}$	DBERC0$_{\text{IDM}}$	DBERC0$_{\text{RST}}$	DBERC0$_{\text{UDE}}$	DBERC0$_{\text{ICMP}}$	DBERC0$_{\text{BRT}}$	DBERC0$_{\text{IRPT}}$	DBERC0$_{\text{TRAP}}$	DBERC0$_{\text{IAC1}}$	DBERC0$_{\text{IAC2}}$	DBERC0$_{\text{IAC3}}$	DBERC0$_{\text{IAC4}}$	DBERC0$_{\text{IAC5}}$	DBERC0$_{\text{IAC6}}$	DBERC0$_{\text{IAC7}}$	DBERC0$_{\text{IAC8}}$	DBERC0$_{\text{DAC1}}$	DBERC0$_{\text{DAC2}}$	DBERC0$_{\text{RET}}$	DBERC0$_{\text{DEVT1}}$	DBERC0$_{\text{DEVT2}}$	DBERC0$_{\text{DCNT1}}$	DBERC0$_{\text{DCNT2}}$	DBERC0$_{\text{CIRPT}}$	DBERC0$_{\text{CRET}}$	DBERC0$_{\text{BKPT}}$	DBERC0$_{\text{DQM}}$	DBERC0$_{\text{FT}}$	Software accessible via mtspr, affected by p_reset_b
0	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	All debug registers
1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	DBCR0$_{\text{IDM}}$
1	1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	DBCR0$_{\text{RST}}$
1	1	—	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	DBCR0$_{\text{UDE}}$
1	1	—	—	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	DBCR0$_{\text{ICMP}}$
1	1	—	—	—	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	DBCR0$_{\text{BRT}}$
1	1	—	—	—	—	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	DBCR0$_{\text{IRPT}}$
1	1	—	—	—	—	—	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	DBCR0$_{\text{TRAP}}$
1	1	—	—	—	—	—	—	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	IAC1, DBCR0$_{\text{IAC1}}$, DBCR1$_{\text{IAC1US,IAC1ER}}$, DBCR6$_{\text{IAC1XM}}$
1	1	—	—	—	—	—	—	—	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	IAC2, DBCR0$_{\text{IAC2}}$, DBCR1$_{\text{IAC2US,IAC2ER}}$, DBCR6$_{\text{IAC2XM}}$
1	1	—	—	—	—	—	—	1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	DBCR1$_{\text{IAC12M}}$
1	1	—	—	—	—	—	—	—	—	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	IAC3, DBCR0$_{\text{IAC3}}$, DBCR1$_{\text{IAC3US,IAC3ER}}$, DBCR6$_{\text{IAC3XM}}$

Table 12-15. DBERC0 resource control (continued)

$DBCR0_{EDM}$	$DBERC0_{IDM}$	$DBERC0_{RST}$	$DBERC0_{UDE}$	$DBERC0_{ICMP}$	$DBERC0_{BRT}$	$DBERC0_{IRPT}$	$DBERC0_{TRAP}$	$DBERC0_{IAC1}$	$DBERC0_{IAC2}$	$DBERC0_{IAC3}$	$DBERC0_{IAC4}$	$DBERC0_{IAC5}$	$DBERC0_{IAC6}$	$DBERC0_{IAC7}$	$DBERC0_{IAC8}$	$DBERC0_{DAC1}$	$DBERC0_{DAC2}$	$DBERC0_{RET}$	$DBERC0_{DEVT1}$	$DBERC0_{DEVT2}$	$DBERC0_{DCNT1}$	$DBERC0_{DCNT2}$	$DBERC0_{CIRPT}$	$DBERC0_{CRET}$	$DBERC0_{BKPT}$	$DBERC0_{DQM}$	$DBERC0_{FT}$	Software accessible via mtspr, affected by p_reset_b
1	1	—	—	—	—	—	—	—	—	—	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	IAC4, $DBCR0_{IAC4}$, $DBCR1_{IAC4US,IAC4ER}$, $DBCR6_{IAC4XM}$
1	1	—	—	—	—	—	—	—	—	1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	$DBCR1_{IAC34M}$
1	1	—	—	—	—	—	—	—	—	—	—	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	IAC5, $DBCR0_{IAC5}$, $DBCR5_{IAC5US,IAC5ER}$, $DBCR6_{IAC5XM}$
1	1	—	—	—	—	—	—	—	—	—	—	—	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	IAC6, $DBCR0_{IAC6}$, $DBCR5_{IAC6US,IAC6ER}$, $DBCR6_{IAC6XM}$
1	1	—	—	—	—	—	—	—	—	—	—	1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	$DBCR5_{IAC56M}$
1	1	—	—	—	—	—	—	—	—	—	—	—	—	1	—	—	—	—	—	—	—	—	—	—	—	—	—	IAC7, $DBCR0_{IAC7}$, $DBCR5_{IAC7US,IAC7ER}$, $DBCR6_{IAC7XM}$
1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	1	—	—	—	—	—	—	—	—	—	—	—	—	IAC8, $DBCR0_{IAC8}$, $DBCR5_{IAC8US,IAC8ER}$, $DBCR6_{IAC8XM}$
1	1	—	—	—	—	—	—	—	—	—	—	—	—	1	1	—	—	—	—	—	—	—	—	—	—	—	—	$DBCR5_{IAC78M}$
1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	1	—	—	—	—	—	—	—	—	—	—	—	DAC1, DVC1 $DBCR0_{DAC1}$, $DBCR2_{DAC1US,DAC1ER}$, $DBCR2_{DVC1M,DVC1BE}$ $DBCR4_{DVC1C,DAC1XM}$
1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	1	—	—	—	—	—	—	—	—	—	—	DAC2, DVC2 $DBCR0_{DAC2}$, $DBCR2_{DAC2US,DAC2ER}$, $DBCR2_{DVC2M,DVC2BE}$ $DBCR4_{DVC2C,DAC2XM}$
1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	1	1	—	—	—	—	—	—	—	—	—	—	$DBCR2_{DAC12M}$
1	1	—	—	—	—	—	—	1	—	—	—	—	—	—	—	1	—	—	—	—	—	—	—	—	—	—	—	$DBCR2_{DAC1LNK}$
1	1	—	—	—	—	—	—	—	—	1	—	—	—	—	—	—	1	—	—	—	—	—	—	—	—	—	—	$DBCR2_{DAC2LNK}$
1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	1	—	—	—	—	—	—	—	—	—	$DBCR0_{RET}$
1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	1	—	—	—	—	—	—	—	—	$DBCR0_{DEVT1}$
1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	1	—	—	—	—	—	—	—	$DBCR0_{DEVT2}$

Table 12-15. DBERC0 resource control (continued)

DBCR0$_{\text{EDM}}$	DBERC0$_{\text{IDM}}$	DBERC0$_{\text{RST}}$	DBERC0$_{\text{UDE}}$	DBERC0$_{\text{ICMP}}$	DBERC0$_{\text{BRT}}$	DBERC0$_{\text{IRPT}}$	DBERC0$_{\text{TRAP}}$	DBERC0$_{\text{IAC1}}$	DBERC0$_{\text{IAC2}}$	DBERC0$_{\text{IAC3}}$	DBERC0$_{\text{IAC4}}$	DBERC0$_{\text{IAC5}}$	DBERC0$_{\text{IAC6}}$	DBERC0$_{\text{IAC7}}$	DBERC0$_{\text{IAC8}}$	DBERC0$_{\text{DAC1}}$	DBERC0$_{\text{DAC2}}$	DBERC0$_{\text{RET}}$	DBERC0$_{\text{DEVT1}}$	DBERC0$_{\text{DEVT2}}$	DBERC0$_{\text{DCNT1}}$	DBERC0$_{\text{DCNT2}}$	DBERC0$_{\text{CIRPT}}$	DBERC0$_{\text{CRET}}$	DBERC0$_{\text{BKPT}}$	DBERC0$_{\text{DQM}}$	DBERC0$_{\text{FT}}$	Software accessible via mtspr, affected by p_reset_b
1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	1	—	—	—	—	—	—	DBCR0$_{\text{DCNT1}}$, DBCR3[$_{\text{DEVT1C1,}}$ DEVT2C1, ICMPC1, IAC1C1, IAC2C1, IAC3C1, IAC4C1, DAC1RC1, DAC1WC1, DAC2RC1, DAC2WC1, IRPTC1, RETC1, DEVT1T1, DEVT2T1, IAC1T1, IAC3T1, DAC1RT1, DAC1WT1, CNT2T1][1], DBCNT$_{\text{CNT1}}$
1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	1	—	—	—	—	—	DBCR0$_{\text{DCNT2}}$, DBCR3[$_{\text{DEVT1C2,}}$ DEVT2C2, ICMPC2, IAC1C2, IAC2C2, IAC3C2, IAC4C2, DAC1RC2, DAC1WC2, DAC2RC2, DAC2WC2][2], DBCNT$_{\text{CNT2}}$
1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	1	1	—	—	—	—	—	DBCR3$_{\text{CONFIG}}$
1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	1	—	—	—	—	DBCR0$_{\text{CIRPT}}$
1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	1	—	—	—	DBCR0$_{\text{CRET}}$
1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	1	—	—	
1	–[3]	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	1	—	DEVENT$_{\text{DQTAG}}$, DDAM
1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	1	DBCR0$_{\text{FT}}$

1 Note that software is given write access to all counter 1 control events and triggers regardless of whether software owns these events. It is considered a programming error to enable counter or trigger events in DBCR3 that are not "owned" by software, and operational results of the counter(s) are undefined if programmed.

2 Note that software is given write access to all counter 2 control events regardless of whether software owns these events. It is considered a programming error to enable counter events in DBCR3 that are not "owned" by software, and operational results of the counter(s) are undefined if programmed.

3 Note: IDM not required to be set to enable software access.

DBERC0 also controls which bits or fields in DBCR0-6 are reset by assertion of **p_reset_b** when DBCR0$_{\text{EDM}}$=1. Only software-owned bits or fields as shown in Table 12-15 are affected in this case, except that DBCR0$_{\text{RST}}$ and DBSR$_{\text{MRR}}$ are updated by assertion of **p_reset_b** regardless of the value of DBCR0$_{\text{EDM}}$ or DBERC0.

12.3.5 Debug Event Select register (DEVENT)

The Debug Event Select register (DEVENT) allows instrumented software to internally generate signals when a **mtspr** instruction is executed and this register is accessed. The values written to this register determine which of the **p_devnt_out[0:7]** processor output signals are asserted upon access. Writing a ‘1’ to any of these bit positions will cause a one clock pulse to be generated on the corresponding output. For **p_devnt_out[0:3]**, a corresponding **jd_watchpt[x]** output is asserted as well to indicate a watchpoint has occurred. These signals may be used for internal core debug resources as well as for SoC level cross-triggering. See the SoC User’s Manual for more information on SoC use cases.

The $\text{DEVENT}_{\text{DEVNT}}$ register field value is undefined on a read; it may or may not remain set to the last value written. Since it is unconditionally shared by hardware debug and software, software should not rely on any value remaining.

The upper 8 bits of the DEVENT register also provide the DQTAG used to identify channels within Data Acquisition Messages. See Section 13.13.1, Data Acquisition ID Tag field, for more detail on the DQTAG.

The DEVENT register is shown in Figure 12-13.

DQTAG	0	DEVNT
0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31

SPR - 975; Reset[1] - 0x0

Figure 12-13. Debug Event Select register (DEVENT)

[1] Reset by processor reset **p_reset_b** if $\text{DBCR0}_{\text{EDM}}$=0, as well as unconditionally by **m_por**. If $\text{DBCR0}_{\text{EDM}}$=1, DBERC0 masks off hardware-owned resources from reset by **p_reset_b** and only software-owned resources indicated by DBERC0 will be reset by **p_reset_b.** Note that DEVNT field is shared by hardware and software but is always reset by **p_reset_b**.

Table 12-16 provides field descriptions for the Debug Event register.

Table 12-16. DEVENT field descriptions

Bit(s)	Name	Description
0:7	DQTAG	Data Acquisition Message IDTAG channel identifier (supplied to Nexus 3)
8:23	—	Reserved, should be cleared.
24:31	DEVNT	Debug Event Signals 00000000 No signal is asserted *xxxxxxx*1 **p_devnt_out[0]** and **jd_watchpt[12]** are asserted for one clock *xxxxxx*1*x* **p_devnt_out[1]** and **jd_watchpt[13]** are asserted for one clock *xxxxx*1*xx* **p_devnt_out[2]** and **jd_watchpt[20]** are asserted for one clock *xxxx*1*xxx* **p_devnt_out[3]** and **jd_watchpt[21]** are asserted for one clock *xxx*1*xxxx* **p_devnt_out[4]** is asserted for one clock *xx*1*xxxxx* **p_devnt_out[5]** is asserted for one clock *x*1*xxxxxx* **p_devnt_out[6]** is asserted for one clock 1*xxxxxxx* **p_devnt_out[7]** is asserted for one clock

12.3.6 Debug Data Acquisition Message register (DDAM)

The Debug Data Acquisition Message register (DDAM) allows instrumented software to generate real-time Data Acquisition Messages (as defined by Nexus 3) via a **mtspr** instruction to this register. See Section 13.13, Data Acquisition messaging, for details.

The DDAM register is shown in Figure 12-14.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
DDAM																															

SPR - 576; Reset[1] - 0x0

Figure 12-14. Debug Data Acquisition Message register (DDAM)

[1] Reset by processor reset **p_reset_b** if DBCR0$_{EDM}$=0, as well as unconditionally by **m_por**. If DBCR0$_{EDM}$=1, DBERC0 masks off hardware-owned resources from reset by **p_reset_b** and only software-owned resources indicated by DBERC0 will be reset by **p_reset_b**.

Table 12-17 provides field descriptions for the Debug Data Acquisition Message register.

Table 12-17. DDAM field descriptions

Bit(s)	Name	Description
0:31	DDAM	Value to be transmitted in a Data Acquisition Message (DQM) (supplied to Nexus 3 with strobe)

12.4 External debug support

External debug support is supplied through the OnCE controller serial interface, which allows access to internal CPU registers and other system state while the CPU is halted in debug mode. All debug resources including DBCR0–6, DBSR, IAC1–8, DAC1–2, DVC1–2, and DBCNT are accessible through the serial OnCE interface in external debug mode. Setting the EDBCR0$_{EDM}$/DBCR0$_{EDM}$ bit to ‘1’ through the OnCE interface enables external debug mode, and unless otherwise permitted by the settings in DBERC0, disables software updates to the debug control registers. When [E]DBCR0$_{EDM}$ is set, debug events enabled to set respective status bits will also cause the CPU to enter Debug Mode if the event is not masked in EDBSRMSK0, as opposed to generating Debug Interrupts, unless the specific events are allocated to software via the settings in DBERC0. In Debug Mode, the CPU is halted at a recoverable boundary, and an external Debug Control Module may control CPU operation through the On-Chip Emulation logic (OnCE).

Note that the descriptions of events in the subsections of Section 12.2, Software debug events and exceptions, refer to setting DBSR status bits, however, when resources are owned by hardware, the events for those resources set the respective status bits in EDBSR0 instead of DBSR.

NOTE

On the initial setting of [E]DBCR0$_{EDM}$ to ‘1’, other bits in DBCR0 will remain unchanged. After [E]DBCR0$_{EDM}$ has been set, all debug register resources may be subsequently controlled through the OnCE interface. The CPU should be placed into debug mode via the OCR$_{DR}$ control bit prior to writing EDM to ‘1’. This gives the debugger the opportunity to cleanly write to the DBCRx registers and the DBSR to clear out any residual state / control information that 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 via the OCR$_{DR}$, by performing go+noexit commands, or by assertion of the **jd_de_b** signal.

NOTE

DBCR0$_{EDM}$ operation will be blocked if 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 will be blocked, all events are blocked, and watchpoints are blocked.

Due to clock domain design, the CPU clock (**m_clk**) must be active in order to perform writes to debug registers other than the OnCE Command register (OCMD), the OnCE Control register (OCR), External Debug Control Register 0 (EDBCR0), External Debug Status register 0 (EDBSR0), External Debug Status Register Mask 0 (EDBSRMSK0), or the DBCR0$_{EDM}$ bit. 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 the DBCRx, DBSR, and DBCNT registers via the OnCE interface should be performed with the CPU in debug mode to guarantee proper operation. Due to the various points in the CPU pipeline where control is sampled and event handshaking is performed, it is possible that modifications to these registers while the CPU is running may result in early or late entry into debug mode, and may have incorrect status posted in the DBSR register.

If resource sharing is enabled via DBERC0, updates to the DBERC0, DBCRx, DBCNT, and DBSR registers must be performed with the CPU in debug mode, since simultaneous updates of register portions could otherwise be attempted, and such updates are not guaranteed to properly occur. The results of such an attempt are undefined.

12.4.1 External debug registers

The external debug registers are used for controlling several debug aspects of the core and reporting status while Zen Z7Zen z445n3Zen z446n3 is in External Debug Mode.

12.4.1.1 External Debug Control Register 0 (EDBCR0)

EDBCR0 is a control register accessible to an external debugger through the OnCE/JTAG port. An external development tool can write to this register in order to enable external debug mode or to enable Debugger Notify Halt instructions (**dnh**, **se_dnh**).

EDBCR0 is not accessible by software, However, the state of $EDBCR0_{EDM}$ is reflected as a read-only bit in $DBCR0_{EDM}$ to software. There is only one physical EDM bit implemented; it is reflected in both the DBCR0 and EDBCR0 registers, and may be written and read using either register by the hardware debugger. For future compatibility, EDBCR0 updates are preferred.

EDBCR0 is shown in Figure 12-15.

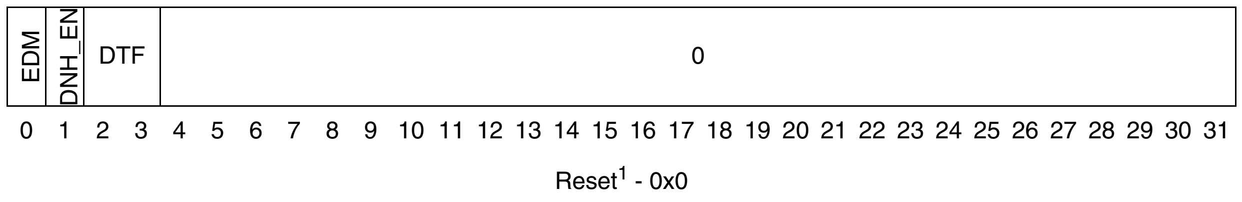


Figure 12-15. External Debug Control Register 0 (EDBCR0)

[1] EDBCR0 is affected (reset) by **j_trst_b** or **m_por** assertion, and remains reset while in the Test_Logic_Reset state, but is not affected by **p_reset_b**.

Table 12-18 provides field descriptions for External Debug Control Register 0.

Table 12-18. EDBCR0 field descriptions

Bit(s)	Name	Description
0	EDM	External Debug Mode. This bit is also reflected in DBCR0 0 External debug mode disabled. Internal debug events not mapped into external debug events. 1 External debug mode enabled. Hardware-owned events will not cause the CPU to vector to interrupt code. Software is not permitted to write to debug registers {DBCRx, DBCNT, IAC1-8, DAC1-2, DVC1-2} unless permitted by settings in DBERC0. When external debug mode is enabled, hardware-owned resources in debug registers are not affected by processor reset **p_reset_b**. This allows the debugger to set up hardware debug events that remain active across a processor reset.
1	DNH_EN	**dnh** Instruction Enable 0 Execution of **dnh** and **se_dnh** instructions cause illegal instruction exceptions to occur. 1 execution of **dnh** and **se_dnh** instructions cause entry into debug mode and a debug halt occurs, regardless of the value of EDM.
2:3	DFT	Debug Freeze Timers Control 00 Timebase, Watchdog timer, and Decrementer are not clocked during a debug session 01 Timebase and Watchdog timer are not clocked during a debug session. Decrementer is unaffected 10 Decrementer is not clocked during a debug session. Timebase and Watchdog timers are unaffected 11 No timer freeze during a debug session
4:31	—	Reserved

12.4.1.2 External Debug Status Register 0 (EDBSR0)

The External Debug Status Register 0 (EDBSR0) contains status on debug events owned by hardware. The External Debug Status Register 0 is set via hardware, and read and cleared via OnCE access by the debugger. Clearing is done by writing to the External Debug Status Register via the OnCE port, with a ‘1’ in any bit position that is to be cleared and ‘0’ in all other bit positions. The write data to EDBSR0 is not direct data, but a mask. A ‘1’ causes the bit to be cleared, and a ‘0’ has no effect. The EDBSR0 register is shown in Figure 12-16.

IDE	UDE	DNH	0	ICMP	BRT	IRPT	TRAP	IAC1	IAC2	IAC3	IAC4-8	DAC1R	DAC1W	DAC2R	DAC2W	RET	0				DEVT1	DEVT2	DCNT1	DCNT2	CIRPT	CRET	VLES	DAC_OFST			CNT1TRG
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Read/Write; Reset[1] - 0x0000_0000

Figure 12-16. External Debug Status Register 0 (EDBSR0)

[1] Reset by **j_trst_b** or **m_por** assertion, and remains reset while in the Test_Logic_Reset state or while $\text{EDBCR0}_{\text{EDM}}$=0.

Table 12-19 provides field descriptions for External Debug Status Register 0.

Table 12-19. EDBSR0 field descriptions

Bit(s)	Name	Description
0	IDE	Imprecise Debug Event Set to 1 if $\text{DBCR0}_{\text{EDM}}$=1 and an imprecise debug event occurs for a hardware-owned DAC event due to a load or store that is terminated with error, or if a hardware-owned ICMP event occurs in conjunction with a EFPU round exception. This bit will not be set for imprecise debug events that are masked via settings in EDBSRMSK0.
1	UDE	Unconditional Debug Event Set to 1 if a hardware-owned Unconditional debug event occurred.
2	DNH	Debugger Notify Halt Event Set to 1 if a debugger notify halt instruction was executed and caused a debug halt.
3	—	Reserved
4	ICMP	Instruction Complete Debug Event Set to 1 if a hardware-owned Instruction Complete debug event occurred.
5	BRT	Branch Taken Debug Event Set to 1 if a hardware-owned Branch Taken debug event occurred.
6	IRPT	Interrupt Taken Debug Event Set to 1 if a hardware-owned Interrupt Taken debug event occurred.
7	TRAP	Trap Taken Debug Event Set to 1 if a hardware-owned Trap Taken debug event occurred.
8	IAC1	Instruction Address Compare 1 Debug Event Set to 1 if a hardware-owned IAC1 debug event occurred.

Table 12-19. EDBSR0 field descriptions (continued)

Bit(s)	Name	Description
9	IAC2	Instruction Address Compare 2 Debug Event Set to 1 if a hardware-owned IAC2 debug event occurred.
10	IAC3	Instruction Address Compare 3 Debug Event Set to 1 if a hardware-owned IAC3 debug event occurred.
11	IAC4-8	Instruction Address Compare 4-8 Debug Event Set to 1 if a hardware-owned IAC4, IAC5, IAC6, IAC7, or IAC8 debug event occurred.
12	DAC1R	Data Address Compare 1 Read Debug Event Set to 1 if a hardware-owned read-type DAC1 debug event occurred while $DBCR0_{DAC1}$=0b10 or $DBCR0_{DAC1}$=0b11
13	DAC1W	Data Address Compare 1 Write Debug Event Set to 1 if a hardware-owned write-type DAC1 debug event occurred while $DBCR0_{DAC1}$=0b01 or $DBCR0_{DAC1}$=0b11
14	DAC2R	Data Address Compare 2 Read Debug Event Set to 1 if a hardware-owned read-type DAC2 debug event occurred while $DBCR0_{DAC2}$=0b10 or $DBCR0_{DAC2}$=0b11
15	DAC2W	Data Address Compare 2 Write Debug Event Set to 1 if a hardware-owned write-type DAC2 debug event occurred while $DBCR0_{DAC2}$=0b01 or $DBCR0_{DAC2}$=0b11
16	RET	Return Debug Event Set to 1 if a hardware-owned Return debug event occurred
17:20	—	Reserved
21	DEVT1	External Debug Event 1 Debug Event Set to 1 if a hardware-owned DEVT1 debug event occurred
22	DEVT2	External Debug Event 2 Debug Event Set to 1 if a hardware-owned DEVT2 debug event occurred
23	DCNT1	Debug Counter 1 Debug Event Set to 1 if a hardware-owned DCNT1 debug event occurred
24	DCNT2	Debug Counter 2 Debug Event Set to 1 if a hardware-owned DCNT2 debug event occurred
25	CIRPT	Critical Interrupt Taken Debug Event Set to 1 if a hardware-owned Critical Interrupt Taken debug event occurred.
26	CRET	Critical Return Debug Event Set to 1 if a hardware-owned Critical Return debug event occurred
27	VLES	VLE Status Set to 1 if a hardware-owned ICMP, BRT, TRAP, RET, CRET, IAC, or DAC debug event occurred on a PowerISA VLE Instruction. Also set for execution of an e_dnh or se_dnh instruction when enabled by $EDBCR0_{DNH\_EN}$. Undefined for IRPT, CIRPT, DEVT[1,2], DCNT[1,2], and UDE events

Table 12-19. EDBSR0 field descriptions (continued)

Bit(s)	Name	Description
28:30	DAC_OFST	Data Address Compare Offset Indicates offset-1 of saved DSRR0 value from the address of the load or store instruction that took a hardware-owned DAC Debug exception, unless a simultaneous DTLB or DSI error occurs, in which case this field is set to 3‘b000 and EDBSR0$_{IDE}$ is set to 1. Normally set to 3‘b000 by a non-DVC DAC. A DVC DAC may set this field to any value.
31	CNT1TRG	Counter 1 Triggered Set to 1 if hardware-owned Debug Counter 1 is triggered by a trigger event.

12.4.1.3 External Debug Status Register Mask 0 (EDBSRMSK0)

The External Debug Status Register Mask 0 (EDBSRMSK0) is used to mask debug events set in EDBSR0 from causing entry into debug halted mode. A ‘1’ stored in any mask bit prevents debug mode entry caused by the corresponding bit being set in EDBSR0. The mask has no effect on DBSR actions or on the setting of EDBSR0 status bits by hardware-owned events, except that the IDE bit will not be set by imprecise hardware-owned debug events that are masked. EDBSRMSK0 may be used to allow debug events owned by hardware to be configured for watchpoint generation purposes without causing debug mode entry when the watchpoint occurs. EDBSRMSK0 is read and written via OnCE access by the debugger. No software access is provided. The EDBSRMSK0 register is shown in Figure 12-17.

0	UDE	DNH	0	ICMP	BRT	IRPT	TRAP	IAC1	IAC2	IAC3	IAC4-8	DAC1R	DAC1W	DAC2R	DAC2W	RET	0				DEVT1	DEVT2	DCNT1	DCNT2	CIRPT	CRET	0				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Read/Write; Reset[1] - 0x0000_0000

Figure 12-17. xternal Debug Status Register Mask 0 (EDBSRMSK0)

[1] Reset by **j_trst_b** or **m_por** assertion, and remains reset while in the Test_Logic_Reset state or while EDBCR0$_{EDM}$=0.

Table 12-20 provides field descriptions for External Debug Status Register Mask 0.

Table 12-20. EDBSRMSK0 field descriptions

Bit(s)	Name	Description
0	—	Reserved
1	UDE	Unconditional Debug Event Set to 1 to mask debug mode entry by EDBSR0$_{UDE}$
2	DNH	Debugger Notify Halt Event Set to 1 to mask debug mode entry by EDBSR0$_{DNH}$
3	—	Reserved
4	ICMP	Instruction Complete Debug Event Set to 1 to mask debug mode entry by EDBSR0$_{ICMP}$

Table 12-20. EDBSRMSK0 field descriptions (continued)

Bit(s)	Name	Description
5	BRT	Branch Taken Debug Event Set to 1 to mask debug mode entry by $EDBSR0_{BRT}$
6	IRPT	Interrupt Taken Debug Event Set to 1 to mask debug mode entry by $EDBSR0_{IRPT}$
7	TRAP	Trap Taken Debug Event Set to 1 to mask debug mode entry by $EDBSR0_{TRAP}$
8	IAC1	Instruction Address Compare 1 Debug Event Set to 1 to mask debug mode entry by $EDBSR0_{IAC1}$
9	IAC2	Instruction Address Compare 2 Debug Event Set to 1 to mask debug mode entry by $EDBSR0_{IAC2}$
10	IAC3	Instruction Address Compare 3 Debug Event Set to 1 to mask debug mode entry by $EDBSR0_{IAC3}$
11	IAC4-8	Instruction Address Compare 4-8 Debug Event Set to 1 to mask debug mode entry by $EDBSR0_{IAC4-8}$
12	DAC1R	Data Address Compare 1 Read Debug Event Set to 1 to mask debug mode entry by $EDBSR0_{DAC1R}$
13	DAC1W	Data Address Compare 1 Write Debug Event Set to 1 to mask debug mode entry by $EDBSR0_{DAC1W}$
14	DAC2R	Data Address Compare 2 Read Debug Event Set to 1 to mask debug mode entry by $EDBSR0_{DAC2R}$
15	DAC2W	Data Address Compare 2 Write Debug Event Set to 1 to mask debug mode entry by $EDBSR0_{DAC2W}$
16	RET	Return Debug Event Set to 1 to mask debug mode entry by $EDBSR0_{RET}$
17:20	—	Reserved
21	DEVT1	External Debug Event 1 Debug Event Set to 1 to mask debug mode entry by $EDBSR0_{DEVT1}$
22	DEVT2	External Debug Event 2 Debug Event Set to 1 to mask debug mode entry by $EDBSR0_{DEVT2}$
23	DCNT1	Debug Counter 1 Debug Event Set to 1 to mask debug mode entry by $EDBSR0_{DCNT1}$
24	DCNT2	Debug Counter 2 Debug Event Set to 1 to mask debug mode entry by $EDBSR0_{DCNT1}$
25	CIRPT	Critical Interrupt Taken Debug Event Set to 1 to mask debug mode entry by $EDBSR0_{CIRPT}$
26	CRET	Critical Return Debug Event Set to 1 to mask debug mode entry by $EDBSR0_{CRET}$
22:31	—	Reserved

12.4.2 OnCE introduction

The e200z759n3 on-chip emulation circuitry (OnCE™/Nexus Class 1 interface) provides a means of interacting with the e200z759n3 core and integrated system so that a user may examine registers, memory, or on-chip peripherals facilitating hardware/software development. OnCE operation is controlled via an industry standard IEEE 1149.1 TAP controller. By using public instructions, the external hardware debugger can freeze or halt the CPU, read and write internal state, and resume normal execution. The core does not contain IEEE 1149.1 standard boundary cells on its interface, as it is a building block for further integration. It does not support the JTAG related boundary scan instruction functionality, although JTAG public instructions may be decoded and signaled to external logic.

The OnCE logic provides for Nexus Class 1 static debug capability (utilizing the same set of resources available to software while in internal debug mode), and is present in all e200z759n3-based designs. The OnCE module also provides support for directly integrating a Nexus class 2 or class 3 Real-Time Debug unit with the e200z759n3 core for development of real-time systems where traditional static debug is insufficient. The partitioning between a OnCE module and a connected Nexus module to provide real-time debug allows for capability and cost trade-offs to be made.

The e200z759n3 core is designed to be a fully integratable module. The OnCE TAP controller and associated enabling logic are designed to allow concatenation with an existing JTAG controller if present in the system. Thus, the e200z759n3 can be easily integrated with existing JTAG designs or as a stand-alone controller.

In order to enable full OnCE operation, the **jd_enable_once** input signal must be asserted. In some system integrations, this is automatic, since the input will be tied asserted. Other integrations may require the execution of the Enable OnCE command via the TAP and appropriate entry of serial data. Exact requirements will be documented by the integrated product specification. The **jd_enable_once** input signal should not change state during a debug session, or undefined activity may occur.

The following figures show the TAP controller state model and the TAP registers implemented by the OnCE logic.

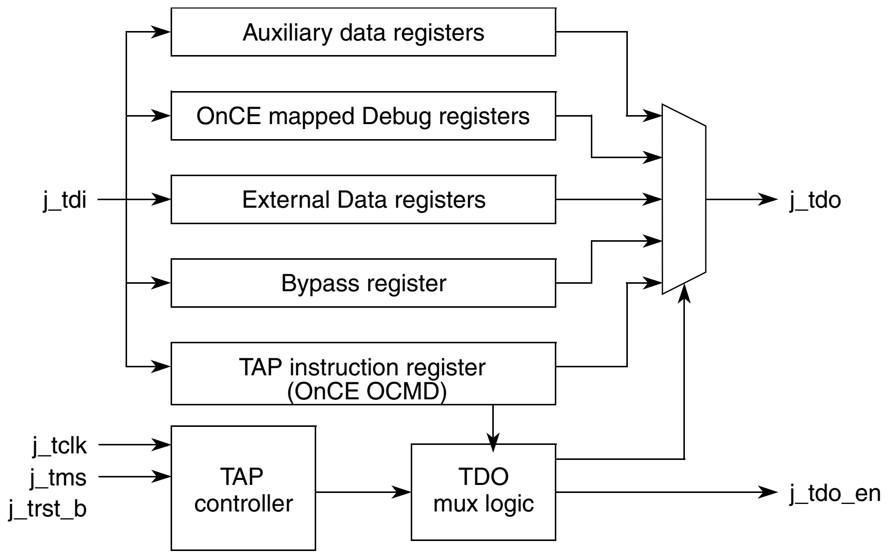


Figure 12-18. OnCE TAP controller and registers

The OnCE controller is implemented as a 16-state FSM, with a one-to-one correspondence to the states defined for the JTAG TAP controller.

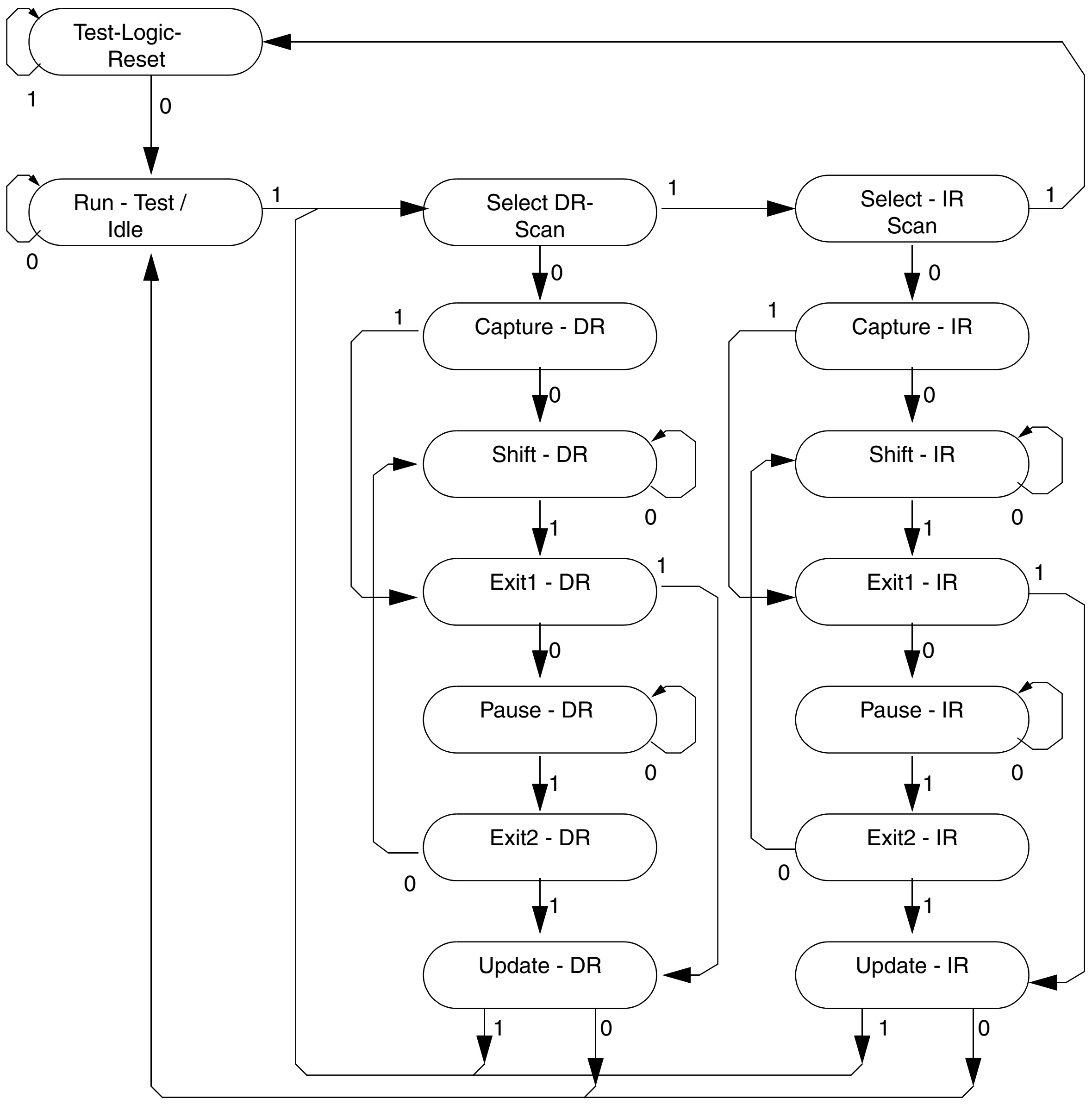


Figure 12-19. OnCE controller stat emachine

Access to processor registers and the contents of memory locations are performed by enabling external debug mode (setting DBCR0$_{EDM}$ to ‘1’), placing the processor into debug mode, followed by scanning instructions and data into and out of the CPU Scan Chain (CPUSCR); execution of scanned instructions by the CPU is used as the method to access required data. Memory locations may be read by scanning a load instruction into the CPU core, which will reference 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, waiting, stopped, or halted states (all indicated via the OnCE Status Register (OSR), Section 12.4.6.1, e200z759n3 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 will indicate the CPU is in the debug state. Instructions may the be single-stepped by scanning new values into the CPUSCR, and performing a OnCE go+noexit command (See Section 12.4.6.2, e200z759n3 OnCE Command register (OCMD)). The CPU will then temporarily exit the debug state (but not the debug session) to execute the instruction, and will then return to the debug state (again indicated via the OnCE Status Register (OSR)). The debug session remains in force until the final OnCE go+exit command is executed, at which time the CPU will return to the previous state it was in (unless a new debug request is pending). A scan into the CPUSCR is required prior to executing each go+exit or go+noexit OnCE command.

12.4.3 JTAG/OnCE pins

The JTAG/OnCE pin interface is used to transfer OnCE instructions and data to the OnCE control block. Depending on the particular resource being accessed, the CPU may need to be placed in the Debug mode. For resources outside of the CPU block and contained in the OnCE block, the processor is not disturbed, and may continue execution. If a processor resource is required, an internal debug request (**dbg_dbgrq**) may be asserted to the CPU by the OnCE controller, and causes the CPU to finish the current instruction being executed, save the instruction pipeline information, enter Debug Mode, and wait for further commands. Asserting **dbg_dbgrq** will cause the chip to exit the low power mode enabled by the setting of MSR_{WE}, as well as temporarily exiting the waiting, stopped or halted power management states.

Table 12-21 details the primary JTAG/OnCE interface signals.

Table 12-21. JTAG/OnCE primary interface signals

Signal Name	Type	Description
j_trst_b	I	JTAG test reset
j_tclk	I	JTAG test clock
j_tms	I	JTAG test mode select
j_tdi	I	JTAG test data input
j_tdo	O	Test data out to master controller or pad
j_tdo_en[1]	O	Enables TDO output buffer

[1] j_tdo_en is asserted when the TAP controller is in the shift_DR or shift_IR state.

A full description of JTAG pins is provided in Section 14.2.23, JTAG support signals.

12.4.4 OnCE internal interface signals

The following paragraphs describe the OnCE interface signals to other internal blocks associated with the OnCE controller.

12.4.4.1 CPU debug request (dbg_dbgrq)

The **dbg_dbgrq** signal is asserted by the OnCE control logic to request the CPU to enter the debug state. It may be asserted for a number of different conditions, and causes the CPU to finish the current instruction being executed, save the instruction pipeline information, enter the debug mode, and wait for further commands.

12.4.4.2 CPU debug acknowledge (cpu_dbgack)

The **cpu_dbgack** signal is asserted by the CPU upon entering the debug state. This signal is used as part of the handshake mechanism between the OnCE control logic and the rest of the CPU. The CPU core may enter debug mode either through a software or hardware event.

12.4.4.3 CPU address, attributes

The CPU address and attribute information are used by a Nexus class 2-4 debug unit with information for real-time address trace information.

12.4.4.4 CPU data

The CPU data buses are used to supply a Nexus class 2-4 debug unit with information for real-time data trace capability.

12.4.5 OnCE interface signals

The following paragraphs describe additional OnCE interface signals to other external blocks such as a Nexus Controller and external blocks that may need information pertaining to debug operation.

12.4.5.1 OnCE enable (jd_en_once)

The OnCE enable signal **jd_en_once** is used to enable the OnCE controller to allow certain instructions and operations to be executed. Assertion of this signal will enable the full OnCE command set, as well as operation of control signals and OnCE Control register functions. When this signal is disabled, only the Bypass, ID and Enable_OnCE commands are executed by the OnCE unit, and all other commands default to a “Bypass” command. The OnCE Status register (OSR) is not visible when OnCE operation is disabled. In addition, OnCE Control register (OCR) functions are disabled, as is the operation of the **jd_de_b** input. Secure systems may choose to leave the **jd_en_once** signal negated until a security check has been performed. Other systems should tie this signal asserted to enable full OnCE operation. The **j_en_once_regsel** output signal is provided to assist external logic performing security checks. Refer to Section 14.2.23.15, Enable OnCE register select (j_en_once_regsel), for a description of the **j_en_once_regsel** output signal.

The **jd_en_once** input must only change state during the Test-Logic-Reset, Run-Test/Idle, or Update_DR TAP states. A new value will take affect after one additional **j_tclk** cycle of synchronization. In addition, **jd_enable_once** input signal must not change state during a debug session, or undefined activity may occur.

12.4.5.2 OnCE debug request/event (jd_de_b, jd_de_en)

If implemented at the SoC level, a system level bidirectional open drain debug event pin **DE_b** (not part of the e200z759n3 interface) provides a fast means of entering the Debug Mode of operation from an external command controller (when input) as well as a fast means of acknowledging the entering of the Debug Mode of operation to an external command controller (when output). The assertion of this pin by a command controller causes the CPU core to finish the current instruction being executed, save the instruction pipeline information, enter Debug Mode, and wait for commands to be entered. If **DE_b** was used to enter the Debug Mode then **DE_b** must be negated after the OnCE controller responds with an acknowledge and before sending the first OnCE command. The assertion of this pin by the CPU Core acknowledges that it has entered the Debug Mode and is waiting for commands to be entered.

To support operation of this system pin, the OnCE logic supplies the **jd_de_en** output and samples the **jd_de_b** input when OnCE is enabled (**jd_en_once** asserted). Assertion of **jd_de_b** will cause the OnCE logic to place the CPU into Debug Mode. Once Debug Mode has been entered, the **jd_de_en** output will be asserted for three **j_tclk** periods to signal an acknowledge. **jd_de_en** can be used to enable the open-drain pulldown of the system level **DE_b** pin.

For systems that do not implement a system level bidirectional open drain debug event pin **DE_b**, the **jd_de_en** and **jd_de_b** signals may still be used to handshake debug entry.

12.4.5.3 e200z759n3 OnCE debug output (jd_debug_b)

The e200z759n3 OnCE Debug output **jd_debug_b** is used to indicate to on-chip resources that a debug session is in progress. Peripherals and other units may use this signal to modify normal operation for the duration of a debug session, which may involve the CPU executing a sequence of instructions solely for the purpose of visibility/system control that are not part of the normal instruction stream the CPU would have executed had it not been placed in debug mode. This signal is asserted the first time the CPU enters the debug state, and remains asserted until the CPU is released by a write to the e200z759n3 OnCE Command Register with the GO and EX bits set, and a register specified as either “No Register Selected” or the CPUSCR. This signal will remain asserted even though the CPU may enter and exit the debug state for each instruction executed under control of the e200z759n3 OnCE controller. See **Section 12.4.6.2** for more information on the function of the GO and EX bits. This signal is not normally used by the CPU.

12.4.5.4 e200z759n3 CPU clock on input (jd_mclk_on)

The e200z759n3 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 e200z759n3 core, is synchronized to the **j_tclk** (scan clock) clock domain, and is presented as a status flag on the **j_tdo** output during the Shift_IR state. External firmware may use this signal to ensure proper scan sequences will occur to access debug resources in the **m_clk** clock domain.

12.4.5.5 Watchpoint events (jd_watchpt[0:29])

The **jd_watchpt[0:29]** signals may be asserted by the e200z759n3 OnCE control logic to signal that a watchpoint condition has occurred. Watchpoints do not cause the CPU to be affected. They are provided to allow external visibility only. Watchpoint events are conditioned by the settings in the DBCR0, DBCR1, and DBCR2 registers, as well as by the DEVENT register, the DTC/DTSA/DTEA registers, and the Performance Monitor control register settings.

12.4.6 e200z759n3 OnCE controller and serial interface

The OnCE controller contains the OnCE command register, the OnCE decoder, and the status/control register. Figure 12-20 is a block diagram of the e200z759n3 OnCE controller. In operation, the OnCE Command register acts as the IR for the e200z759n3 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.

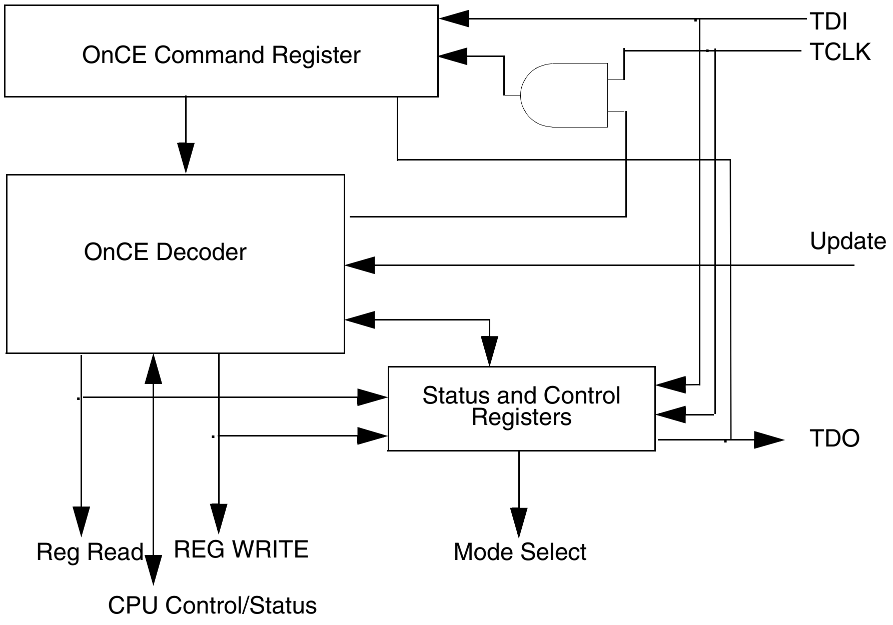


Figure 12-20. e200z759n3 OnCE controller and serial interface

12.4.6.1 e200z759n3 OnCE Status Register (OSR)

Status information regarding the state of the CPU is latched into the OnCE Status Register (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.

MCLK	ERR	0	RESET	HALT	STOP	DEBUG	WAIT	0	1
0	1	2	3	4	5	6	7	8	9

Figure 12-21. OnCE Status Register (OSR)

Table 12-22 provides field descriptions for the OnCE Status Register.

Table 12-22. OSR field descriptions

Bit(s)	Name	Description
0	MCLK	**m_clk** Status Bit 0 Inactive state 1 Active state This status bit reflects the logic level on the **jd_mclk_on** input signal after capture by **j_tclk**.
1	ERR	The ERROR bit is used to indicate that an error condition occurred during attempted execution of the last single-stepped instruction (GO+NoExit with CPUSCR or No Register Selected in OCMD), and that the instruction may not have been properly executed. This could occur if an Interrupt (all classes including External, Critical, machine check, Storage, Alignment, Program, TLB, etc.) occurred while attempting to perform the instruction single step. In this case, the CPUSCR will contain information related to the first instruction of the Interrupt handler, and no portion of the handler will have been executed.
2	—	Reserved, set to zero
3	RESET	RESET Mode This bit reflects the inverted logic level on the CPU **p_reset_b** input after capture by **j_tclk**.
4	HALT	HALT Mode This bit reflects the logic level on the CPU **p_halted** output after capture by **j_tclk**.
5	STOP	STOP Mode This bit reflects the logic level on the CPU **p_stopped** output after capture by **j_tclk**.
6	DEBUG	Debug Mode This bit is asserted once the CPU is in debug mode. It is negated once the CPU exits debug mode (even during a debug session)
7	WAIT	Waiting Mode This bit reflects the logic level on the CPU **p_waiting** output after capture by **j_tclk**.
8	0	Reserved, set to 0 for 1149.1 compliance
9	1	Reserved, set to 1 for 1149.1 compliance

12.4.6.2 e200z759n3 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 e200z759n3 OnCE Decoder. The OCMD is shown in Figure 12-22. The OCMD is updated when the TAP controller enters the Update-IR state. It contains fields for controlling access to a resource, as well as controlling single-step operation and exit from OnCE mode.

Although the OCMD is updated during the Update-IR TAP controller state, the corresponding resource is accessed in the DR scan sequence of the TAP controller, and as such, the Update-DR state must be transitioned through in order for an access to occur. In addition, the Update-DR state must also be transitioned through in order for the single-step and/or exit functionality to be performed, even though the command appears to have no data resource requirement associated with it.

R/W	GO	EX	RS[0:6]						
0	1	2	3	4	5	6	7	8	9

Reset - 10’b1000000010 on assertion of **j_trst_b** or **m_por**, or while in the Test_Logic_Reset state

Figure 12-22. OnCE Command register (OCMD)

Table 12-23 provides field descriptions for the OnCE Command register.

Table 12-23. OCMD field descriptions

Bit(s)	Name	Description
0	R/W	Read/Write Command Bit The R/W bit specifies the direction of data transfer. The table below describes the options defined by the R/W bit. 0 Write the data associated with the command into the register specified by RS[0:6] 1 Read the data contained in the register specified by RS[0:6] **Note:** The R/W bit generally ignored for read-only or write-only registers, although the PC FIFO pointer is only guaranteed to be update when R/W=1. In addition, it is ignored for all bypass operations. When performing writes, most registers are sampled in the Capture-DR state into a 32-bit shift register, and subsequently shifted out on **j_tdo** during the first 32 clocks of Shift-DR.
1	GO	Go Go Command Bit 0 Inactive (no action taken) 1 Execute instruction in IR If the GO bit is set, the chip will execute the instruction that resides in the IR register in the CPUSCR. To execute the instruction, the processor leaves the debug mode, executes the instruction, and if the EX bit is cleared, returns to the debug mode immediately after executing the instruction. The processor goes on to normal operation if the EX bit is set, and no other debug request source is asserted. The GO command is executed only if the operation is a read/write to CPUSCR or a read/write to “No Register Selected”. Otherwise the GO bit is ignored.The processor will leave the debug mode after the TAP controller Update-DR state is entered. On a GO+NoExit operation, returning to debug mode is treated as a debug event, thus exceptions such as machine checks and interrupts may take priority and prevent execution of the intended instruction. Debug firmware should mask these exceptions as appropriate. The OSR_{ERR} bit indicates such an occurrence. **Note:** Asynchronous interrupts are blocked on a GO+Exit operation until the first instruction to be executed begins execution. See Section 12.4.9.6, Exiting debug mode and interrupt blocking.

Table 12-23. OCMD field descriptions (continued)

Bit(s)	Name	Description
2	EX	Exit Command Bit 0 Remain in debug mode 1 Leave debug mode If the EX bit is set, the processor will leave the debug mode and resume normal operation until another debug request is generated. The Exit command is executed only if the Go command is issued, and the operation is a read/write to CPUSCR or a read/write to “No Register Selected”. Otherwise the EX bit is ignored. The processor will leave the debug mode after the TAP controller Update-DR state is entered. **Note:** If the DR bit in the OnCE control register is set or remains set, or if a bit in EDBSR0 is set and DBCR0$_{EDM}$=1 (external debug mode is enabled), or if another debug request source is asserted, then the processor may return to the debug mode without execution of an instruction, even though the EX bit was set. **Note:** Asynchronous interrupts are blocked on a GO+Exit operation until the first instruction to be executed begins execution. See Section 12.4.9.6, Exiting debug mode and interrupt blocking.
3:9	RS	Register Select The Register Select bits define which register is source (destination) for the read (write) operation. Table 12-24 indicates the e200z759n3 OnCE register addresses. Attempted writes to read-only registers are ignored.

Table 12-24 indicates the e200z759n3 OnCE register addresses.

Table 12-24. e200z759n3 OnCE register addressing

RS[0:6]	Register Selected
000 0000	Reserved
000 0001	Reserved
000 0010	JTAG ID (read-only)
000 0011–- 000 1111	Reserved
001 0000	CPU Scan Register (CPUSCR)
001 0001	No Register Selected (Bypass)
001 0010	OnCE Control Register (OCR)
001 0011	Reserved
001 0100 – 001 1111	Reserved
010 0000	Instruction Address Compare 1 (IAC1)
010 0001	Instruction Address Compare 2 (IAC2)
010 0010	Instruction Address Compare 3 (IAC3)
010 0011	Instruction Address Compare 4 (IAC4)
010 0100	Data Address Compare 1 (DAC1)
010 0101	Data Address Compare 2 (DAC2)

Table 12-24. e200z759n3 OnCE register addressing (continued)

RS[0:6]	Register Selected
010 0110	Data Value Compare 1 (DVC1)
010 0111	Data Value Compare 2 (DVC2)
010 1000	Instruction Address Compare 5 (IAC5)
010 1001	Instruction Address Compare 6 (IAC6)
010 1010	Instruction Address Compare 7 (IAC7)
010 1011	Instruction Address Compare 8 (IAC8)
010 1100	Debug Counter Register (DBCNT)
010 1101	Debug PCFIFO (PCFIFO)
010 1110	External Debug Control Register 0 (EDBCR0)
010 1111	External Debug Status Register 0 (EDBSR0)
011 0000	Debug Status Register (DBSR)
011 0001	Debug Control Register 0 (DBCR0)
011 0010	Debug Control Register 1 (DBCR1)
011 0011	Debug Control Register 2 (DBCR2)
011 0100	Debug Control Register 3 (DBCR3)
011 0101	Debug Control Register 4 (DBCR4)
011 0110	Debug Control Register 5 (DBCR5)
011 0111	Debug Control Register 6 (DBCR6)
011 1000– 011 1011	Reserved (do not access)
011 1100	External Debug Status Register MASK 0 (EDBSRMSK0)
011 1101	Debug Data Acquisition Message Register (DDAM)
011 1110	Debug Event Control (DEVENT)
011 1111	Debug External Resource Control (DBERC0)
100 0000– 110 1110	Reserved (do not access)
110 1111	Reserved for Shared Nexus Control Register Select
111 0000– 111 1001	General Purpose register selects [0:9]
111 1010	Cache Debug Access Control Register (CDACNTL) -(See Section 11.19, Cache memory access for debug / error handling)
111 1011	Cache Debug Access Data Register (CDADATA) -(See Section 11.19, Cache memory access for debug / error handling)
111 1100	Nexus3-Access (see Chapter 13, Nexus 3 Module)
111 1101	LSRL Select (see Test Specification)

Table 12-24. e200z759n3 OnCE register addressing (continued)

RS[0:6]	Register Selected
111 1110	Enable_OnCE[1]
111 1111	Bypass

[1] Causes assertion of the j_en_once_regsel output. Refer to Section 14.2.23.15, Enable OnCE register select (j_en_once_regsel)

The OnCE Decoder receives as input the 10-bit command from the OCMD, and status signals from the processor, and generates all the strobes required for reading and writing the selected OnCE registers.

Single stepping of instructions is performed by placing the CPU in debug mode, scanning in appropriate information into the CPUSCR, and setting the Go bit (with the EX bit cleared) with the RS field indicating either the CPUSCR or No Register Selected. After executing a single instruction, the CPU will re-enter debug mode and await further commands. During single-stepping, exception conditions may occur if not properly masked by debug firmware (interrupts, machine checks, bus error conditions, etc.) and may prevent the desired instruction from being successfully executed. The OSR_{ERR} bit is set to indicate this condition. In these cases, values in the CPUSCR will correspond to the first instruction of the exception handler.

Additionally, the $[E]DBCR0_{EDM}$ bit is forced to '1' internally while single-stepping to prevent Debug events from generating Debug interrupts. Also, during a debug session, the DBSR and the DBCNT registers are frozen from updates due to debug events regardless of $[E]DBCR0_{EDM}$. They may still be modified during a debug session via a single-stepped **mtspr** instruction, or via OnCE access if $[E]DBCR0_{EDM}$ is set.

12.4.6.3 e200z759n3 OnCE Control Register (OCR)

The e200z759n3 OnCE Control Register is a 32-bit register used to force the e200z759n3 core into debug mode and to enable / disable sections of the e200z759n3 OnCE control logic. It also provides control over the MMU during a debug session (see Section 12.6, MMU and cache operation during debug). The control bits are read/write. These bits are only effective while OnCE is enabled (**jd_en_once** asserted). The OCR is shown in Figure 12-23.

0–7	8	9–10	11	12	13	14	15	16	17–18	19	20	21	22	23	24–28	29	30	31
0	I_DMDIS	0	I_DVLE	I_DI	I_DM	0	I_DE	D_DMDIS	0	D_DW	D_DI	D_DM	D_DG	D_DE	0	WKUP	FDB	DR

Reset - 0x0000_0000 on **m_por**, **j_trst_b**, or entering Test_logic_Reset state

Figure 12-23. OnCE Control Register (OCR)

Table 12-25 provides field descriptions for the OnCE Control Register.

Table 12-25. OCR field descriptions

Bit(s)	Name	Description
0:7	—	Reserved
8	I_DMDIS	Instruction Side Debug MMU Disable Control Bit (I_DMDIS) 0 MMU not disabled for debug sessions 1 MMU disabled for debug sessions This bit may be used to control whether the MMU is enabled normally, or whether the MMU is disabled during a debug session for Instruction Accesses. When enabled, the MMU functions normally. When disabled, for Instruction Accesses, no address translation is performed (1:1 address mapping), and the TLB VLE, I,M, and E bits are taken from the OCR bits I_VLE, I_DI, I_DM, and I_DE bits. The W and G bits are assumed '0'. The SX and UX access permission control bits are set to'1' to allow full access. When disabled, no TLB miss or TLB exceptions are generated for Instruction accesses. External access errors can still occur.
9:10	—	Reserved
11	I_DVLE	Instruction Side Debug TLB 'VLE' Attribute Bit (I_DVLE) This bit is used to provide the 'VLE' attribute bit to be used when the MMU is disabled during a debug session.
12	I_DI	Instruction Side Debug TLB 'I' Attribute Bit (I_DI) This bit is used to provide the 'I' attribute bit to be used for Instruction accesses when the MMU is disabled for Instruction accesses during a debug session.
13	I_DM	Instruction Side Debug TLB 'M' Attribute Bit (I_DM) This bit is used to provide the 'M' attribute bit to be used for Instruction accesses when the MMU is disabled for Instruction accesses during a debug session.
14	—	Reserved
15	I_DE	Instruction Side Debug TLB 'E' Attribute Bit (I_DE) This bit is used to provide the 'E' attribute bit to be used for Instruction accesses when the MMU is disabled for Instruction accesses during a debug session.
16	D_DMDIS	Data Side Debug MMU Disable Control Bit (D_DMDIS) 0 MMU not disabled for debug sessions 1 MMU disabled for debug sessions This bit may be used to control whether the MMU is enabled normally, or whether the MMU is disabled during a debug session for Data Accesses. When enabled, the MMU functions normally. When disabled, for Data Accesses, no address translation is performed (1:1 address mapping), and the TLB WIMGE bits are taken from the OCR bits D_DW, D_DI, D_DM, D_DG, and D_DE bits. The SR, SW, UR, and UW access permission control bits are set to'1' to allow full access. When disabled, no TLB miss or TLB exceptions are generated for Data accesses. External access errors can still occur.
17:18	—	Reserved
19	D_DW	Data Side Debug TLB 'W' Attribute Bit (D_DW) This bit is used to provide the 'W' attribute bit to be used for Data accesses when the MMU is disabled for Data accesses during a debug session.
20	D_DI	Data Side Debug TLB 'I' Attribute Bit (D_DI) This bit is used to provide the 'I' attribute bit to be used for Data accesses when the MMU is disabled for Data accesses during a debug session.
21	D_DM	Data Side Debug TLB 'M' Attribute Bit (D_DM) This bit is used to provide the 'M' attribute bit to be used for Data accesses when the MMU is disabled for Data accesses during a debug session.

Table 12-25. OCR field descriptions (continued)

Bit(s)	Name	Description
22	D_DG	Data Side Debug TLB ‘G’ Attribute Bit (D_DG) This bit is used to provide the ‘G’ attribute bit to be used for Data accesses when the MMU is disabled for Data accesses during a debug session.
23	D_DE	Data Side Debug TLB ‘E’ Attribute Bit (D_DE) This bit is used to provide the ‘E’ attribute bit to be used for Data accesses when the MMU is disabled for Data accesses during a debug session.
24:28	—	Reserved
29	WKUP	Wakeup Request Bit (WKUP) This control bit may be used to force the e200z759n3 **p_wakeup** output signal to be asserted. This control function may be used by debug firmware to request that the chip-level clock controller restore the **m_clk** input to normal operation regardless of whether the CPU is in a low power state to ensure that debug resources may be properly accessed by external hardware through scan sequences.
30	FDB	Force Breakpoint Debug Mode Bit (FDB) This control bit is used to determine whether the processor is operating in breakpoint debug enable mode or not. The processor may be placed in breakpoint debug enable mode by setting this bit. In breakpoint debug enable mode, execution of the ‘**bkpt**’ pseudo- instruction will cause the processor to enter debug mode, as if the **jd_de_b** input had been asserted. This bit is qualified with DBCR0$_{EDM}$, which must be set for FDB to take effect. Note that this bit has no effect on **dnh** or **se_dnh** instruction operation.
31	DR	CPU Debug Request Control Bit This control bit is used to unconditionally request the CPU to enter the Debug Mode. The CPU will indicate that Debug Mode has been entered via the data scanned out in the shift-IR state. 0 No Debug Mode request 1 Unconditional Debug Mode request When the DR bit is set the processor will enter Debug mode at the next instruction boundary.

12.4.7 Access to debug resources

Resources contained in the e200z759n3 OnCE module that do not require the e200z759n3 processor core to be halted for access may be accessed while the e200z759n3 core is running, and will not interfere with processor execution. Accesses to other resources such as the CPUSCR require the e200z759n3 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 e200z759n3 core prior to access. Note that a scan operation to update the CPUSCR is required prior to exiting debug mode if debug mode has been entered.

Some cases of write accesses other than accesses to the OnCE Command and Control registers, or the EDM bit of DBCR0 require the e200z759n3 **m_clk** to be running for proper operation. The OnCE control register provides a means of signaling this need to a system level clock control module.

In addition, since the CPU may cause multiple bits of certain registers to change state, reads of certain registers while the CPU is running (DBSR, DBCNT, etc.) may not have consistent bit settings unless read twice with the same value indicated. In order to guarantee that the contents are consistent, the CPU should be placed into debug mode, or multiple reads should be performed until consistent values have been obtained on consecutive reads.

Table 12-26 provides a list of access requirements for OnCE registers.

Table 12-26. OnCE register access requirements

Register name	Access Requirements					Notes
	Requires jd_en_once to be asserted	Requires DBCR0 $_{EDM}$ = 1	Requires m_clk active for write access	Requires CPU to be halted for read access	Requires CPU to be halted for write access	
Enable_OnCE	N	N	N	N	—	
Bypass	N	N	N	N	N	
CPUSCR	Y	Y	Y	Y	Y	
DAC1	Y	Y	Y	N	*1	
DAC2	Y	Y	Y	N	*1	
DBCNT	Y	Y	Y	N2	*1	
DBCR0	Y	Y	Y	N	*1	*DBCR0$_{EDM}$ access only requires **jd_en_once** asserted
DBCR1-6	Y	Y	Y	N	*1	
DEVENT	Y	Y	Y	N	*1	
DBERC0	Y	N	Y	N	*1	
DBSR	Y	Y	Y	N2	*1	
EDBCR0	Y	N	N	N	N	
EDBSR0	Y	N	N	N	N	
EDBSRMSK0	Y	N	N	N	N	
IAC1-8	Y	Y	Y	N	*1	
JTAG ID	N	N	—	N	—	Read-only
OCR	Y	N	N	N	N	
OSR	Y	N	—	N	—	Read-only, accessed by scanning out IR while **jd_en_once** is asserted
PC FIFO	Y	N	Y	N	N	Updates frozen while OCMD holds PCFIFO register encoding. Note: No updates will 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

Table 12-26. OnCE register access requirements (continued)

Register name	Access Requirements					Notes
	Requires jd_en_once to be asserted	Requires DBCR0 EDM = 1	Requires m_clk active for write access	Requires CPU to be halted for read access	Requires CPU to be halted for write access	
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 operation, and the fact that updates are not synchronized to a particular clock, instruction, or bus cycle boundary, therefore it is strongly recommended to ensure the processor is first placed into debug mode before updates to these registers are performed.

[2] Reads of these registers while the CPU is running may not give data that is self-consistent due to synchronization across clock domains.

12.4.8 Methods of entering debug mode

The OnCE Status Register indicates that the CPU has entered the debug mode via the DEBUG status bit. The following sections describe how e200z759n3 Debug Mode is entered assuming the OnCE circuitry has been enabled. e200z759n3 OnCE operation is enabled by the assertion of the **jd_en_once** input (see Section 12.4.5.1, OnCE enable (jd_en_once)).

12.4.8.1 External debug request during RESET

Holding the **jd_de_b** signal asserted during the assertion of **p_reset_b**, and continuing to hold it asserted following the negation of **p_reset_b** causes the e200z759n3 core to enter Debug Mode. After receiving an acknowledge via the OnCE Status Register DEBUG bit, the external command controller should negate the **jd_de_b** signal before sending the first command. Note that in this case the e200z759n3 core does not execute an instruction before entering Debug Mode, although the first instruction to be executed may be fetched prior to entering Debug Mode.

In this case, all values in the debug scan chain will be undefined, and the external Debug Control Module is responsible for proper initialization of the chain before debug mode is exited. In particular, the exception processing associated with reset, may not be performed when the debug mode is exited, thus, the Debug controller must initialize the PC, MSR, and IR to the image that the processor would have obtained in performing reset exception processing, or must cause the appropriate reset to be re-asserted.

12.4.8.2 Debug request during RESET

Asserting a debug request by setting the DR bit in the OCR during the assertion of **p_reset_b** causes the chip to enter debug mode. In this case the chip may fetch the first instruction of the reset exception handler, but does not execute an instruction before entering debug mode. In this case, all values in the debug scan chain will be undefined, and the external Debug Control Module is responsible for proper initialization of the chain before debug mode is exited. In particular, the exception processing associated with reset may not be performed when the debug mode is exited, thus, the Debug controller must initialize the PC, MSR, and IR to the image that the processor would have obtained in performing reset exception processing, or must cause the appropriate reset to be re-asserted.

12.4.8.3 Debug request during normal activity

Asserting a debug request by setting the DR bit in the OCR during normal chip activity causes the chip to finish the execution of the current instruction and then enter the debug mode. Note that in this case the chip completes the execution of the current instruction and stops after the newly fetched instruction enters the CPU instruction register. This process is the same for any newly fetched instruction including instructions fetched by the interrupt processing, or those that will be aborted by the interrupt processing.

12.4.8.4 Debug request during Waiting, Halted, or Stopped state

Asserting a debug request by setting the DR bit in the OCR when the chip is in the Waiting state (**p_waiting** asserted), Halted state (**p_halted** asserted) or Stopped state (**p_stopped** asserted) causes the CPU to exit the state and enter the debug mode once the CPU clock **m_clk** has been restored. Note that in this case, the CPU will negate the **p_waiting, p_halted** and **p_stopped** outputs. Once the debug session has ended, the CPU will return to the state it was in prior to entering debug mode.

To signal the chip-level clock generator to re-enable **m_clk**, the **p_wakeup** output will be asserted whenever the debug block is asserting a debug request to the CPU due to $\mathrm{OCR}_{\mathrm{DR}}$ being set, or **jd_de_b** assertion, and will remain set from then until the debug session ends (**jd_debug_b** goes from asserted to negated). In addition, the status of the **jd_mclk_on** input (after synchronization to the **j_tclk** clock domain) may be sampled along with other status bits from the **j_tdo** output during the Shift_IR TAP controller state. This status may be used if necessary by external debug firmware to ensure proper scan sequences occur to registers in the **m_clk** clock domain.

12.4.8.5 Software request during normal activity

Upon executing a ‘**bkpt**’ pseudo-instruction (for e200z759n3, defined to be an all 0’s instruction opcode) when the OCR register’s (FDB) bit is set (debug mode enable control bit is true), and $\mathrm{DBCR0}_{\mathrm{EDM}}$=1, the CPU enters the debug mode after the instruction following the ‘**bkpt**’ pseudo-instruction has entered the instruction register.

12.4.8.6 Debug notify halt instructions

The **dnh, e_dnh,** and **se_dnh** instructions allow software to transition the core from a running state to a debug halted state if enabled by $\mathrm{EDBCR0}_{\mathrm{DNH\_EN}}$, and provide the external debugger with bits reserved in the instruction itself to pass additional information. Entry into debug mode is ***not*** conditioned on $\mathrm{EDBCR0}_{\mathrm{EDM}}$, allowing for debug of software debug handlers as well as other software. For e200z759n3, when the CPU enters a debug halted state due to a **dnh**, **e_dnh**, or **se_dnh** instruction, the instruction will be stored in the CPUSCR[IR] portion, and the CPUSCR[PC] value will point to the instruction. The external debugger should update the CPUSCR prior to exiting the debug halted state to point past the **dnh**, **e_dnh**, or **se_dnh** instruction.

12.4.9 CPU Status and Control Scan Chain Register (CPUSCR)

A number of on-chip registers store the CPU pipeline status and are configured in a single scan chain for access by the e200z759n3 OnCE controller. The CPUSCR register contains these processor resources, which are used to restore the pipeline and resume normal chip activity upon return from the debug mode, as well as a mechanism for the emulator software to access processor and memory contents. Figure 12-24 shows the block diagram of the pipeline information registers contained in the CPUSCR. Once debug mode has been entered, it is required to scan in and update this register prior to exiting debug mode.

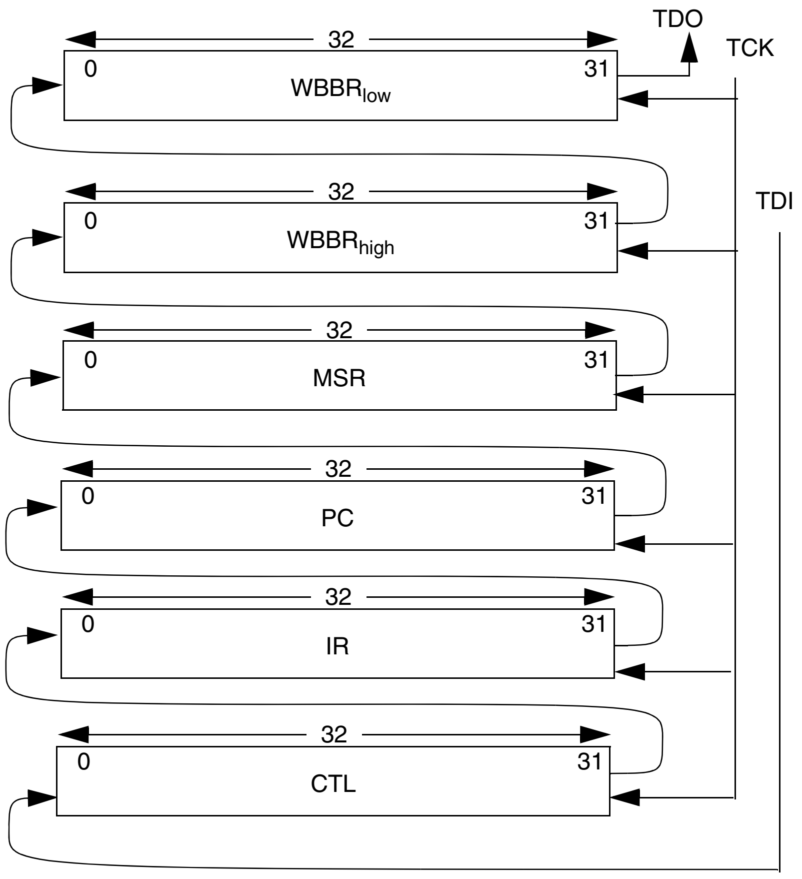


Figure 12-24. CPU Scan Chain Register (CPUSCR)

12.4.9.1 Instruction Register (IR)

The Instruction Register (IR) provides a mechanism for controlling the debug session by serving as a means for forcing in selected instructions, and then causing them to be executed in a controlled manner by the debug control block. The opcode of the next instruction to be executed when entering debug mode is contained in this register when the scan-out of this chain begins. This value should be saved for later restoration if continuation of the normal instruction stream is desired.

On scan-in, in preparation for exiting debug mode, this register is filled with an instruction opcode selected by debug control software. By selecting appropriate instructions and controlling the execution of those instructions, the results of execution may be used to examine or change memory locations and processor registers. The debug control module external to the processor core will control 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.

12.4.9.2 Control State register (CTL)

The Control State register (CTL) is a 32-bit register that stores the value of certain internal CPU state variables before the debug mode is entered. This register is affected by the operations performed during the debug session and should normally be restored by the external command controller when returning to normal mode. In addition to saved internal state variables, two of the bits are used by emulation firmware to control the debug process. In certain circumstances, emulation firmware must modify the content of this register as well as the PC and IR values in the CPUSCR before exiting debug mode. These cases are described below.

*	IRSTAT13	IRSTAT12	IRSTAT11	IRSTAT10	WAITING	PCOFST	PCINV	FFRA	IRSTAT0	IRSTAT1	IRSTAT2	IRSTAT3	IRSTAT4	IRSTAT5	IRSTAT6	IRSTAT7	IRSTAT8	IRSTAT9
0 1 2 3 4 5 6 7 8 9 10	11	12	13	14	15	16 17 18 19	20	21	22	23	24	25	26	27	28	29	30	31

Figure 12-25. Control State Register (CTL)

Table 12-27. CTL field descriptions

Bit(s)	Name	Description
0:10	*	Internal State Bits These control bits represent internal processor state and should be restored to their original value after a debug session is completed, i.e when a e200z759n3 OnCE command is issued with the GO and EX bits set and not ignored. When performing instruction execution during a debug session (see **Section 12.4.5.3**) that is not part of the normal program execution flow, these bits should be set to a 0.
11	IRStat13	IR Status Bit 13 This control bit indicates an Instruction Address Compare 8 event status for the IR. 0 No Instruction Address Compare 8 event occurred on the fetch of this instruction. 1 An Instruction Address Compare 8 event occurred on the fetch of this instruction.
12	IRStat12	IR Status Bit 12 This control bit indicates an Instruction Address Compare 7 event status for the IR. 0 No Instruction Address Compare 7 event occurred on the fetch of this instruction. 1 An Instruction Address Compare 7 event occurred on the fetch of this instruction.
13	IRStat11	IR Status Bit 11 This control bit indicates an Instruction Address Compare 6 event status for the IR. 0 No Instruction Address Compare 6 event occurred on the fetch of this instruction. 1 An Instruction Address Compare 6 event occurred on the fetch of this instruction.
14	IRStat10	IR Status Bit 10 This control bit indicates an Instruction Address Compare 5 event status for the IR. 0 No Instruction Address Compare 5 event occurred on the fetch of this instruction. 1 An Instruction Address Compare 5 event occurred on the fetch of this instruction.

Table 12-27. CTL field descriptions (continued)

Bit(s)	Name	Description
15	WAITING	WAITING State Status This bit indicates whether the CPU was in the waiting state prior to entering debug mode. If set, the CPU was in the waiting state. Upon exiting a debug session, the value of this bit in the restored CPUSCR will determine whether the CPU re-enters the waiting state on a go+exit. 0 CPU was not in the waiting state when debug mode was entered 1 CPU was in the waiting state when debug mode was entered
16:19	PCOFST	PC Offset Field This field indicates whether the value in the PC portion of the CPUSCR must be adjusted prior to exiting debug mode. Due to the pipelined nature of the CPU, the PC value must be backed-up by emulation software in certain circumstances. The PCOFST field specifies the value to be subtracted from the original value of the PC. This adjusted PC value should be restored into the PC portion of the CPUSCR just prior to exiting debug mode with a go+exit. In the event the PCOFST is non-zero, the IR should be loaded with a nop instruction instead of the original IR value, other wise the original value of IR should be restored. (But see PCINV, which overrides this field.) 0000 No correction required. 0001 Subtract 0x04 from PC. 0010 Subtract 0x08 from PC. 0011 Subtract 0x0C from PC. 0100 Subtract 0x10 from PC. 0101 Subtract 0x14 from PC. All other encodings are reserved.
20	PCINV	PC and IR Invalid Status Bit This status bit indicates that the values in the IR and PC portions of the CPUSCR are invalid. Exiting debug mode with the saved values in the PC and IR will have unpredictable results. Debug firmware should initialize the PC and IR values in the CPUSCR with desired values prior to exiting debug mode if this bit was set when debug mode was initially entered. 0 No error condition exists. 1 Error condition exists. PC and IR are corrupted.
21	FFRA	Feed Forward RA Operand Bit This control bit causes the content of the WBBR to be used as the RA operand value (RS for logical, mtspr, mtdcr, cntlzw, and shift operations, RX for VLE se_ instructions, RT for e_{logical_op}2i type instructions, RB for evaddiw, evsubifw, and the value to use as the PC for calculating the LR update value for branch with link type instructions) of the first instruction to be executed following an update of the CPUSCR. This allows the debug firmware to update processor registers — initialize the WBBR with the desired value, set the FFRA bit, and execute a ori Rx,Rx,0 instruction to the desired register. 0 No action. 1 Content of WBBR used as operand value
22	IRSTAT0	IR Status Bit 0 This control bit indicates a TEA status for the IR. 0 No TEA occurred on the fetch of this instruction. 1 TEA occurred on the fetch of this instruction.
23	IRSTAT1	IR Status Bit 1 This control bit indicates a TLB Miss status for the IR. 0 No TLB Miss occurred on the fetch of this instruction. 1 TLB Miss occurred on the fetch of this instruction.

Table 12-27. CTL field descriptions (continued)

Bit(s)	Name	Description
24	IRSTAT2	IR Status Bit 2 This control bit indicates an Instruction Address Compare 1 event status for the IR. 0 No Instruction Address Compare 1 event occurred on the fetch of this instruction. 1 An Instruction Address Compare 1 event occurred on the fetch of this instruction.
25	IRSTAT3	IR Status Bit 3 This control bit indicates an Instruction Address Compare 2 event status for the IR. 0 No Instruction Address Compare 2 event occurred on the fetch of this instruction. 1 An Instruction Address Compare 2 event occurred on the fetch of this instruction.
26	IRSTAT4	IR Status Bit 4 This control bit indicates an Instruction Address Compare 3 event status for the IR. 0 No Instruction Address Compare 3 event occurred on the fetch of this instruction. 1 An Instruction Address Compare 3 event occurred on the fetch of this instruction.
27	IRSTAT5	IR Status Bit 5 This control bit indicates an Instruction Address Compare 4 event status for the IR. 0 No Instruction Address Compare 4 event occurred on the fetch of this instruction. 1 An Instruction Address Compare 4 event occurred on the fetch of this instruction.
28	IRSTAT6	IR Status Bit 6 This control bit indicates a Parity Error status for the IR. 0 No Parity Error occurred on the fetch of this instruction. 1 Parity Error occurred on the fetch of this instruction.
29	IRSTAT7	IR Status Bit 7 This control bit indicates a Precise External Termination Error status for the IR, or a 2nd half TLB Miss for the instruction in the IR. 0 No Precise External Termination Error occurred on the fetch of this instruction. 1 If IRSTAT1 = '0', a Precise External Termination Error occurred on the fetch of this instruction. If IRSTAT1 = '1', a TLB Miss occurred on the 2nd half of this instruction.
30	IRSTAT8	IR Status Bit 8 This control bit indicates the PowerISA VLE status for the IR. 0 IR contains a BookE instruction. 1 IR contains a PowerISA VLE instruction, aligned in the Most Significant Portion of IR if 16-bit.
31	IRSTAT9	IR Status Bit 9 This control bit indicates the PowerISA VLE Byte-ordering Error status for the IR, or a BookE misaligned instruction fetch, depending on the state of IRStat8. 0 IR contains an instruction without a byte-ordering error and no Misaligned Instruction Fetch Exception has occurred (no MIF). 1 If IRSTAT8 = '0', A BookE Misaligned Instruction Fetch Exception has occurred while filling the IR. If IRSTAT8 = '1', IR contains an instruction with a byte-ordering error due to mismatched VLE page attributes, or due to E indicating little-endian for a VLE page.

Emulation firmware should modify the content of the CTL, PC, and IR values in the CPUSCR during execution of debug related instructions as well as just prior to exiting debug with a go+exit command. During the debug session, the CTL register should be written with the FFRA bit set as appropriate, and all other bit set to '0', and the IR set to the value of the desired instruction to be executed. IRStat8 will be used to determine the type of instruction present in the IR.

Just prior to exiting debug mode with a go+exit, the PCINV status bit that was originally present when debug mode was first entered should be tested, 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, then the PCOFST bits should be examined to determine whether the PC value must be adjusted. Due to the pipelined nature of the CPU, the PC value must be backed-up by emulation software in certain circumstances. The PCOFST field specifies the value to be subtracted from the original value of the PC. This adjusted PC value should be restored in to the PC portion of the CPUSCR just prior to exiting debug mode with a go+exit. In the event the PCOFST is non-zero, the IR should be loaded with a 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 will generally point to the last completed instruction, although that instruction will not be re-executed. The nop instruction is executed instead, and instruction fetch and execution will resume at location PC+4. IRStat8 will be used to determine the type of instruction present in the IR, thus should be cleared in this case. Note that debug events that may occur on the nop (ICMP) will be generated (and optionally counted) if enabled.

For the CTL register, the internal state bits should be restored to their original value. The IRStatus bits should be set to ‘0’s if the PC was adjusted. If no PC adjustment was performed, emulation firmware should determine whether other IRStat flags should be set to ‘0’ to avoid re-entry into debug mode for an instruction breakpoint request. Upon exiting debug mode with go+exit, if one of these bits is set, debug mode will be re-entered prior to any further instruction execution.

12.4.9.3 Program Counter register (PC)

The PC is a 32-bit register that stores the value of the program counter that was present when the chip entered the debug mode. It is affected by the operations performed during the debug mode and must be restored by the external command controller when the CPU returns to normal mode. PC normally points to the instruction contained in the IR portion of CPUSCR. If debug firmware wishes to redirect program flow to an arbitrary location, the PC and IR should be initialized to correspond to the first instruction to be executed upon resumption of normal processing. Alternatively, the IR may be set to a nop and the PC set to point to the location prior to the location at which it is desired to redirect flow to. On exiting debug mode, the nop will be executed, and instruction fetch and execution will resume at PC+4.

12.4.9.4 Write-Back Bus Register (WBBR$_{low}$, WBBR$_{high}$)

WBBR is used as a means of passing operand information between the CPU and the external command controller. Whenever the external command controller needs to read the contents of a register or memory location, it will force the chip to execute an instruction that brings that information to WBBR. WBBR$_{low}$ holds the 32-bit result of most instructions including load data returned for a load or load with update instruction. For SPE/EFPU instructions that generate 64-bit results, WBBR$_{low}$ holds the low-order 32 bits of the result. WBBR$_{high}$ holds the updated effective address calculated by a load with update instruction. For SPE/EFPU instructions that generate 64-bit results, WBBR$_{high}$ holds the high-order 32 bits of the result. It is undefined for other instructions.

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 will be latched into WBBR$_{low}$. The contents of WBBR$_{low}$ can then be delivered serially to the external command controller. To update a processor resource, this register is initialized with a data value to be written and an **ori** instruction is executed, which uses this value as a substitute data value. The Control State register FFRA bit forces the value of the WBBR$_{low}$ to be substituted for the normal RS source value of the **ori** instruction, thus allowing updates to processor registers to be performed (refer to Section 12.4.9.2, Control State register (CTL), for more detail on the CTL$_{FFRA}$ bit).

WBBR$_{low}$ and WBBR$_{high}$ are generally undefined on instructions that do not write back 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$_{low}$ and WBBR$_{high}$ 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 that MSR$_{SPE}$ must be set in order for these operations to be performed properly.

12.4.9.5 Machine State Register (MSR)

The MSR is a 32-bit register used to read/write the Machine State Register. Whenever the external command controller needs to save or modify the contents of the Machine State Register, this register is used.This register is affected by the operations performed during the debug mode and must be restored by the external command controller when returning to normal mode.

12.4.9.6 Exiting debug mode and interrupt blocking

When exiting debug mode with a Go+Exit, “asynchronous” interrupts are blocked until the first instruction to be executed begins execution. This includes External and Critical input, NMI, machine check, timer, decrementer, and watchdog interrupts. Asynchronous debug interrupts are not blocked however, and the CPU will re-enter debug mode without executing an instruction following Go+Exit, although it may fetch an instruction and discard it. Exceptions due to an illegal instruction or error flags set within the CPUSCR CTL register are not blocked, since they apply to the instruction in the CPUSCR IR.

12.4.10 Instruction Address FIFO buffer (PC FIFO)

To assist debugging and keep track of program flow, a First-In-First-Out (FIFO) buffer stores the addresses of the last eight instruction change of flow destinations that were fetched. These include exception vectoring to an exception handler and returns, as well as pipeline refills due to execution of the **isync** instruction.

12.4.10.1 PC FIFO

The PC FIFO stores the addresses of the last eight instruction change of flow addresses that were actually taken. The FIFO is implemented as a circular buffer containing eight 32-bit registers and one 3-bit counter. All the registers have the same address, but any access to the FIFO address will cause 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 12-26 shows the block diagram of the PC FIFO.

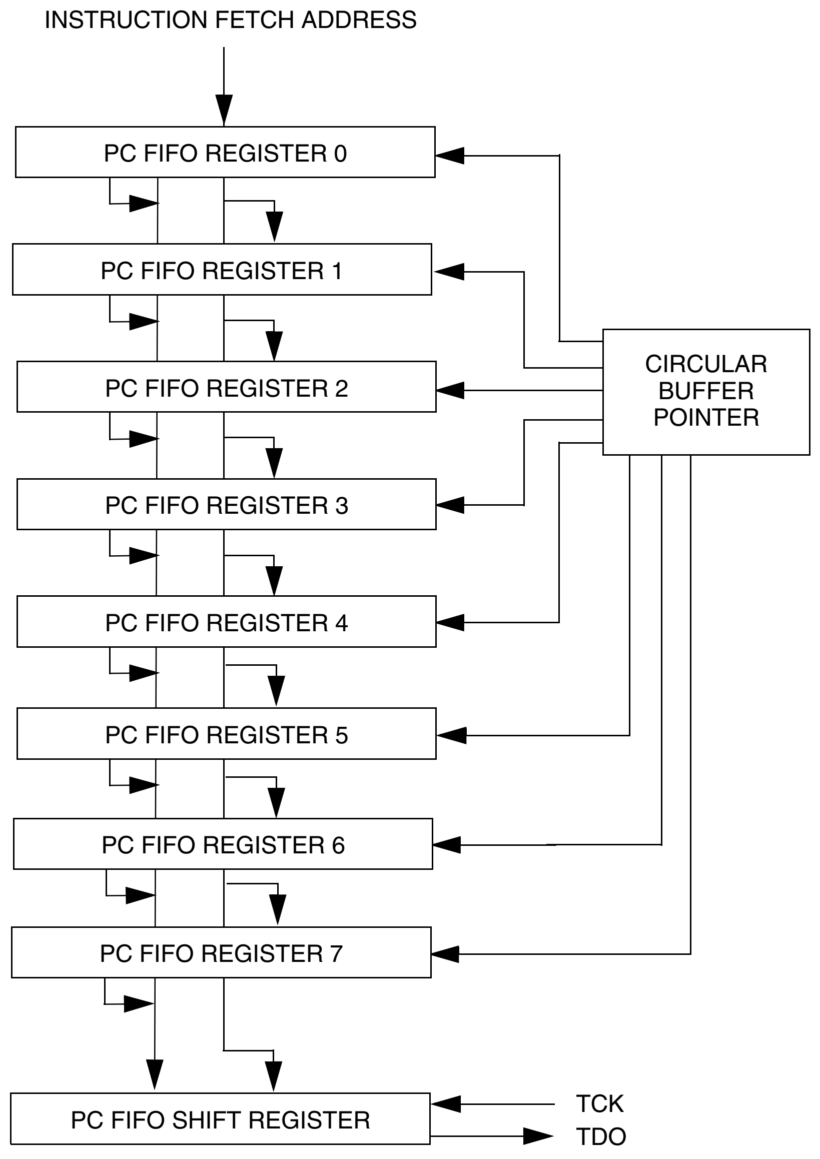


Figure 12-26. OnCE PC FIFO

The FIFO is not affected by the operations performed during a Debug session except for the FIFO pointer increment when accessing the FIFO. When entering Debug mode, the FIFO counter will be pointing to the FIFO register containing the address of the oldest of the eight change of flow prefetches. When the OCMD RS field is loaded with the value corresponding to the PC FIFO (010 1101), the current pointer value is captured into a temporary register. This temporary value (not the actual FIFO counter) is incremented as FIFO reads or writes are performed. The first FIFO read will obtain the oldest address and the following FIFO reads will return the other addresses from the oldest to the newest (the order of execution). Writes will operate similarly.

Updates to the FIFO by change of flows are frozen whenever the OCMD register contains a command whose RS[0:6] field points to the PC FIFO (010 1101) to allow firmware to access the contents of the PC

FIFO without placing the CPU into debug mode. After completing all accesses to the PC FIFO, another OCMD value that does not select the PC FIFO should be entered to allow the PC FIFO to resume updating.

To ensure FIFO coherence, a complete set of eight accesses of the FIFO should be performed since each access increments the temporary FIFO pointer, thus making it point to the next location. After eight accesses, the pointer will point to the same location it pointed to before starting the access procedure. The temporary counter value captures the actual counter each time the OCMD RS field transitions to the value corresponding to the PC FIFO (010 1101).

The FIFO pointer is reset to entry 0 when either **j_trst_b** or **m_por** are asserted.

12.4.11 Reserved registers (reserved)

The Reserved Registers are used to control various test control logic. These registers are not intended for customer use. To preclude device and/or system damage, these registers should not be accessed.

12.5 Watchpoint support

e200z759n3 supports the generation and signaling of watchpoints when operating in internal debug mode ($DBCR0_{IDM}$=1) or in external debug mode ($DBCR0_{EDM}$=1). Watchpoints are indicated with a dedicated set of interface signals. The **jd_watchpt[0:29]** output signals are used to indicate that a watchpoint has occurred. Certain watchpoints (DEVNT-based and DTC-based) are not qualified with $DBCR0_{EDM}$ or $DBCR0_{IDM}$.

Each debug address compare function (IAC1-**8**, DAC1-2), and Debug Counter event (DCNT1-2), as well as other event types are capable of triggering a watchpoint output. The DBCRx control fields are used to configure watchpoints, regardless of whether events are enabled in DBCR0. Watchpoints may occur whenever an associated event would have been posted in the Debug Status Register if enabled. No explicit enable bits are provided for watchpoints; they are always enabled by definition. During a debug session, events (other than DEVT1 and DEVT2) with a corresponding DBSR bit are blocked from asserting a watchpoint. The DEVNT-based and DTC-based watchpoints are not blocked during a debug session. If not desired, for address-based events the base address values for these events may be programmed to an unused system address. MSR_{DE} has no effect on watchpoint generation.

External logic may monitor the assertion of these signals for debugging purposes. Watchpoints are signaled in the clock cycle following the occurrence of the actual event. The Nexus3 module also monitors assertion of these signals for various development control purposes (See Section 13.14, Watchpoint Trace Messaging).

Table 12-28. Watchpoint output signal assignments

Signal name	Type	Description
jd_watchpt[0]	IAC1	Instruction Address Compare 1 watchpoint Asserted whenever an IAC1 compare occurs regardless of being enabled to set DBSR status
jd_watchpt[1]	IAC2	Instruction Address Compare 2 watchpoint Asserted whenever an IAC2 compare occurs regardless of being enabled to set DBSR status

Table 12-28. Watchpoint output signal assignments (continued)

Signal name	Type	Description
jd_watchpt[2]	IAC3	Instruction Address Compare 3 watchpoint Asserted whenever an IAC3 compare occurs regardless of being enabled to set DBSR status
jd_watchpt[3]	IAC4	Instruction Address Compare 4 watchpoint Asserted whenever an IAC4 compare occurs regardless of being enabled to set DBSR status
jd_watchpt[4]	DAC1[1]	Data Address Compare 1 watchpoint Asserted whenever a DAC1 compare occurs regardless of being enabled to set DBSR status
jd_watchpt[5]	DAC2[1]	Data Address Compare 2 watchpoint Asserted whenever a DAC2 compare occurs regardless of being enabled to set DBSR status
jd_watchpt[6]	DCNT1	Debug Counter 1 watchpoint Asserted whenever Debug Counter 1 decrements to zero regardless of being enabled to set DBSR status
jd_watchpt[7]	DCNT2	Debug Counter 2 watchpoint Asserted whenever Debug Counter 2 decrements to zero regardless of being enabled to set DBSR status
jd_watchpt[8]	IAC5	Instruction Address Compare 5 watchpoint Asserted whenever an IAC5 compare occurs regardless of being enabled to set DBSR status
jd_watchpt[9]	IAC6	Instruction Address Compare 6 watchpoint Asserted whenever an IAC6 compare occurs regardless of being enabled to set DBSR status
jd_watchpt[10]	DEVT1	Debug Event Input 1 watchpoint Asserted whenever a DEVT1 debug event occurs regardless of being enabled to set DBSR status
jd_watchpt[11]	DEVT2	Debug Event Input 2 watchpoint Asserted whenever a DEVT2 debug event occurs regardless of being enabled to set DBSR status
jd_watchpt[12]	DEVNT0	Debug Event Output 0 watchpoint Asserted whenever a '1' is written to the bit of the DEVNT field of the DEVENT debug register corresponding to jd_watchpt[12]
jd_watchpt[13]	DEVNT1	Debug Event Output 1 watchpoint Asserted whenever a '1' is written to the bit of the DEVNT field of the DEVENT debug register corresponding to jd_watchpt[13]
jd_watchpt[14]	IAC7	Instruction Address Compare 7 watchpoint Asserted whenever an IAC7 compare occurs regardless of being enabled to set DBSR status
jd_watchpt[15]	IAC8	Instruction Address Compare 8 watchpoint Asserted whenever an IAC8 compare occurs regardless of being enabled to set DBSR status

Table 12-28. Watchpoint output signal assignments (continued)

Signal name	Type	Description
jd_watchpt[16]	IRPT	Interrupt watchpoint Asserted whenever an IRPT debug event occurs regardless of being enabled to set DBSR status
jd_watchpt[17]	RET	Return watchpoint Asserted whenever a RET debug event occurs regardless of being enabled to set DBSR status
jd_watchpt[18]	CIRPT	Critical Interrupt watchpoint Asserted whenever a CIRPT debug event occurs regardless of being enabled to set DBSR status
jd_watchpt[19]	CRET	Critical Return watchpoint Asserted whenever a CRET debug event occurs regardless of being enabled to set DBSR status
jd_watchpt[20]	DEVNT2	Debug Event Output 2 watchpoint Asserted whenever a '1' is written to the bit of the DEVNT field of the DEVENT debug register corresponding to jd_watchpt[20]
jd_watchpt[21]	DEVNT3	Debug Event Output 3 watchpoint Asserted whenever a '1' is written to the bit of the DEVNT field of the DEVENT debug register corresponding to jd_watchpt[21]
jd_watchpt[22]	PMEVENT	Performance Monitor Event input watchpoint Asserted whenever **p_pm_event** transitions from a '0' to a '1'
jd_watchpt[23]	PMC0	Performance Monitor Counter 0 watchpoint Asserted whenever PMC0 triggers an event based on $\text{PMLCa0}_{\text{PMP}}$
jd_watchpt[24]	PMC1	Performance Monitor Counter 1 watchpoint Asserted whenever PMC1 triggers an event based on $\text{PMLCa1}_{\text{PMP}}$
jd_watchpt[25]	PMC2	Performance Monitor Counter 2 watchpoint Asserted whenever PMC2 triggers an event based on $\text{PMLCa2}_{\text{PMP}}$
jd_watchpt[26]	PMC3	Performance Monitor Counter 3 watchpoint Asserted whenever PMC3 triggers an event based on $\text{PMLCa3}_{\text{PMP}}$
jd_watchpt[27]	DTC1	Data Trace Control Range 1 watchpoint Asserted whenever an access meets the conditions for DTC Range 1
jd_watchpt[28]	DTC2	Data Trace Control Range 2 watchpoint Asserted whenever an access meets the conditions for DTC Range 2
jd_watchpt[29]	DTC3	Data Trace Control Range 3 watchpoint Asserted whenever an access meets the conditions for DTC Range 3

[1] If the corresponding event is completely disabled in DBCR0, either load-type or store-type data accesses are allowed to generate watchpoints, otherwise watchpoints are generated only for the enabled conditions.

12.6 MMU and cache operation during debug

Normal operation of the MMU may be modified during a 'debug session' via the OnCE Control Register (OCR). A debug session begins when the CPU initially enters debug mode, and ends when 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 for Page Attribute (VLE, W, I, M, G, E) bits normally provided by a matching TLB entry. In addition, no address translation is performed, and instead, a 1:1 mapping of effective to real addresses is performed.

When disabled during a debug session, no TLB miss or TLB-related storage interrupt conditions will occur. If the debugger desires to use the normal translation process, the MMU may be left enabled in the OnCE OCR, and normal translation (including the possibility of a TLB Miss or storage interrupt) will remain in effect.

The OCR control bits are used when debug mode is entered. Refer to the bit definitions in the OCR (Section 12.4.6.3, e200z759n3 OnCE Control Register (OCR), for more detail. When the MMU is disabled for instruction accesses (OCR_{I_DMDIS}) or for data accesses (OCR_{D_DMDIS}), substituted page attribute bits will control operation on respective accesses initiated during debug. No address translation will be performed; instead, a 1:1 mapping between effective and real addresses will be performed for respective accesses.

12.7 Cache array access during debug

The cache arrays may be read and written during debug mode via the CDACNTL and CDADATA debug registers. This functionality is described in detail in Section 11.19, Cache memory access for debug / error handling.

12.8 Basic steps for enabling, using, and exiting external debug mode

The following steps show one possible scenario for a debugger wishing to use the external debug facilities. *This simplified flow is intended to illustrate basic operations, but does not cover all potential methods in depth.*

Enabling external debug mode and initializing debug registers

1. The debugger should ensure that the **jd_en_once** control signal is asserted in order to enable OnCE operation.
2. Select the OCR and write a value to it in which OCR_{DR} and OCR_{WKUP} are set to '1'. The TAP controller must step through the proper states as outlined earlier. This step will place the CPU in a debug state in which it is halted and awaiting single-step commands or a release to normal mode.
3. Scan out the value of the OSR to determine that the CPU clock is running and the CPU has entered the Debug state. This can be done in conjunction with a Read of the CPUSCR. The OSR is shifted out during the Shift_IR state. The CPUSCR will be shifted out during the Shift_DR state. The debugger should save the scanned-out value of CPUSCR for later restoration.
4. Select the DBCR0 register and update it with the $DBCR0_{EDM}$ bit set.
5. Clear the DBSR status bits.
6. Write appropriate values to the DBCR0–6, IAC, DAC, and DBCNT registers. Note that the initial write to DBCR0 will only affect the EDM bit, so the remaining portion of the register must now be initialized, keeping the EDM bit set.

At this point the system is ready to commence debug operations. Depending on the desired operation, different steps must occur.

- Optionally, set the OCR_{I_DMDIS, D_DMDIS} control bits to ensure that no TLB misses will occur while performing the debug operations.
- Optionally, ensure that the values entered into the MSR portion of the CPUSCR during the following steps cause interrupt to be disabled (clearing MSR_{EE} and MSR_{CE}). This will ensure 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 12.4.9.2, Control State register (CTL)), with a Go+Noexit OnCE Command value.
2. The debugger scans out the OSR with "no-register selected", Go cleared, and determines that the PCU has re-entered the Debug state and that no ERR condition occurred.

To return the CPU to normal operation (without disabling external debug mode).

1. The OCR_{I_DMDIS, D_DMDIS}, OCR_{DR}, control bits should be cleared, leaving the OCR_{WKUP} bit set.
2. The debugger restores the CPUSCR with a previously saved value of the CPUSCR (with appropriate modification of the PC and IR as described in Section 12.4.9.2, Control State register (CTL)), with a Go+Exit OnCE Command value.
3. The OCR_{WKUP} bit may then be cleared.

To exit External Debug Mode:

1. The debugger should place the CPU in the debug state via the OCR_{DR} with OCR_{WKUP} asserted, scanning out and saving the CPUSCR.
2. The debugger should write the DBCR0-6 registers as needed, likely clearing every enable except the $DBCR0_{EDM}$ bit.
3. The debugger should write the DBSR to a cleared state.
4. The debugger should re-write the DBCR0 with all bits including EDM cleared.
5. The debugger should clear the OCR_{DR} bit.
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 12.4.9.2, Control State register (CTL)), with a Go+Exit OnCE Command value.
7. The OCR_{WKUP} bit 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.

12.9 Parallel Signature unit

To support applications requiring system integrity checking during operation, the e200z759n3 core provides a Parallel Signature unit, which is capable of monitoring the internal CPU read and write buses for data accesses, and accumulating a pair of 32-bit MISR signatures of the values transferred over these buses for data accesses.

The primitive polynomial used is P(X) = $1 + X^{10} + X^{30} + X^{31} + X^{32}$. Values are accumulated based on an initially programmed “seed” value, and are qualified based on active byte lanes of the CPU internal read and write buses (**p_d_data_in[0:63]**, **p_d_data_out[0:63]**) as indicated via the **p_d_tsiz[0:2], p_d_elsize[0:1]**, and **p_d_addr[29:31]** signals. Inactive byte lanes use a value of all zeros as input data to the MISRs. Refer to Table 15-12 for active byte lane information. Note that for read data, the data returned from the Cache or BIU is used directly from **p_d_data_in[0:63]** for accumulation. For write cycles however, the data accumulated is based on the data that is written to the cache or BIU after it has been properly aligned and permuted according to the endian mode of the access, thus **p_d_data_out[0:63]** is not used directly. Instead, the proper memory image is used.

If an external termination error (bus error) occurs on any accumulated read data, the returned read data is ignored, a value of all zeros is used instead, and the error is logged. External termination errors occurring on data writes are not logged, even though the data is accumulated, since the data driven by the CPU was valid.

No data is accumulated for transfer errors signaled due to TLB Error, Cache Parity Error, Byte Ordering Error, DSI or ISI due to permissions violations, or for Alignment Errors.

No accumulation occurs for cache control operations such as **dcba**, **dcbi**, **icbi**, **dcbf**, **dcbst**, **dcbt**, **icbt**, **dcbtst**, **dcbz**, **dcbtls**, **dcbtstls**, **dcblc**, or for cache operations initiated via the mtspr L1CSR0 or L1FINV0.

The unit may be independently enabled for data read cycles and data write cycles, allowing for flexible usage. Software may also control accumulation of software provided values via a pair of update registers. In addition, a counter is provided for software use to monitor the number of beats of data that have been compressed.

Updates are performed when the parallel signature registers are initialized, when a qualified internal bus cycle is terminated, when a software update is performed via a high or low update register, and when the parallel signature high or low registers are written with a **mtdcr** instruction.

NOTE

Updates due to qualified bus transfers are suppressed for the duration of a debug session.

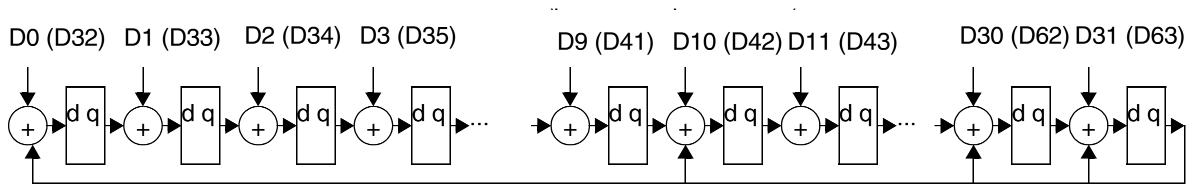


Figure 12-27. Parallel Signature unit operation

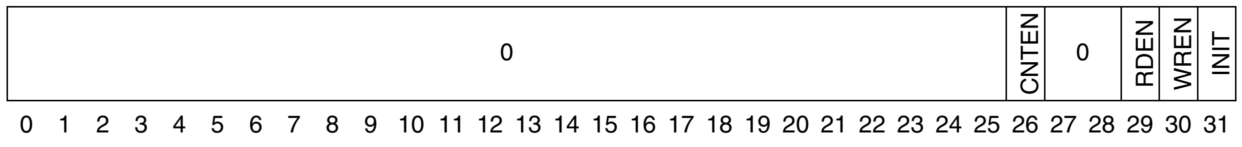
The Parallel Signature unit consists of seven registers as described below. Access to these registers is privileged. No user-mode access is allowed.

NOTE

Proper access of the PSU registers requires that the **mfdcr** instruction that reads a PSU register be preceded by either an **mbar** or an **msync** instruction. To ensure that the effects of a **mtdcr** instruction to one of the PSU registers has taken effect, the **mtdcr** should be followed by a context synchronizing instruction (**sc**, **isync**, **rfi**, **rfci**, **rfdi**).

12.9.1 Parallel Signature Control Register (PSCR)

The Parallel Signature Control Register (PSCR) controls operation of the Parallel Signature unit.



DCR - 272; Read/Write; Reset - 0x0

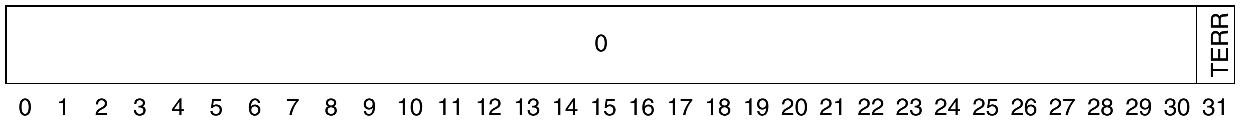
Figure 12-28. Parallel Signature Control Register (PSCR)

Table 12-29. PSCR field descriptions

Bits	Name	Description
0:25	—	These bits are reserved
26	CNTEN	Counter Enable 0 Counter is disabled. 1 Counter is enabled. Counter is incremented on every accumulated transfer, or on a mtdcr psulr,Rn instruction.
27:28	—	These bits are reserved
29	RDEN	Read Enable 0 Processor data read cycles are ignored. 1 Processor data reads cycles are accumulated. For inactive byte lanes, zeros are used for the data values.
30	WREN	Write Enable 0 Processor write cycles are ignored. 1 Processor write cycles are accumulated. For inactive byte lanes, zeros are used for the data values.
31	INIT	This bit may be written with a ‘1’ to set the values in the PSHR, PSLR, and PSCTR registers to all ‘0’s (0x00000000). This bit always reads as ‘0’.

12.9.2 Parallel Signature Status Register (PSSR)

The Parallel Signature Status Register (PSSR) provides status relative to operation of the Parallel Signature unit.



DCR - 273; Read/Write; Reset -Unaffected

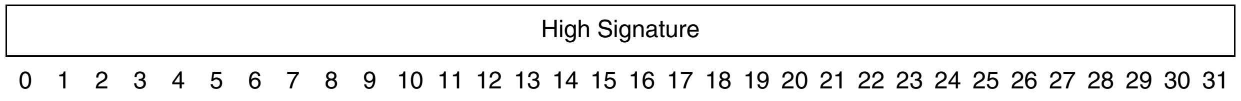
Figure 12-29. Parallel Signature Status Register (PSSR)

Table 12-30. PSSR field descriptions

Bits	Name	Description
0:30	—	These bits are reserved
31	TERR	Transfer Error Status 0 No transfer error has occurred on accumulated read data since this bit was last cleared by software. 1 A transfer error has occurred on accumulated read data since this bit was last cleared by software. This bit indicates whether a transfer error has occurred on accumulated read data, and that the read data values returned were ignored and zeros are used instead. This bit is not cleared by hardware; only a software write of '1' to this bit will cause it to be cleared.

12.9.3 Parallel Signature High Register (PSHR)

The Parallel Signature High Register (PSHR) provides signature information for the high word (bits 0:31) of the internal read and write buses. It may be written via a **mtdcr pshr, Rs** instruction (DCR register 274) to initialize a seed value prior to enabling signature accumulation. The PSCR$_{INIT}$ control bit may also be used to clear the PSHR. This register is unaffected by system reset, thus should be initialized by software prior to performing parallel signature operations.



DCR - 274; Read/Write; Reset -Unaffected

Figure 12-30. Parallel Signature High Register (PSHR)

12.9.4 Parallel Signature Low Register (PSLR)

The Parallel Signature Low Register (PSLR) provides signature information for the low word (bits 32:63) of the internal read and write buses. It may be written via a **mtdcr pslr, Rs** instruction (DCR register 275) to initialize a seed value prior to enabling signature accumulation. The PSCR$_{INIT}$ control bit may also be used to clear the PSLR. This register is unaffected by system reset, thus should be initialized by software prior to performing parallel signature operations.

Low Signature

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

DCR - 275; Read/Write; Reset -Unaffected

Figure 12-31. Parallel Signature Low Register (PSLR)

12.9.5 Parallel Signature Counter Register (PSCTR)

The Parallel Signature Counter Register (PSCTR) provides count information for signature accumulation. The counter is incremented on every accumulated transfer, or on a **mtdcr psulr,Rn** instruction. It may be written via a **mtdcr psctr, Rs** instruction (DCR register 276) to initialize a value prior to enabling signature accumulation. The PSCR$_{INIT}$ control bit may also be used to clear the PSCTR. This register is unaffected by system reset, thus should be initialized by software prior to performing parallel signature operations.

Counter

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

DCR - 276; Read/Write; Reset -Unaffected

Figure 12-32. Parallel Signature Counter Register (PSCTR)

12.9.6 Parallel Signature Update High Register (PSUHR)

The Parallel Signature Update High Register (PSUHR) provides a means for updating the high signature value via software. It may be written via a **mtdcr psuhr, Rs** instruction (DCR register 277) to cause signature accumulation to occur in the parallel signature high register (PSHR) using the data value written. This register is write-only; attempted reads return a value of all zeros. Writing to this register does not cause the PSCTR to increment.

High Signature Update Data

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

DCR - 277; Write-only; Reset -Unaffected

Figure 12-33. Parallel Signature Update High Register (PSUHR)

12.9.7 Parallel Signature Update Low Register (PSULR)

The Parallel Signature Update Low Register (PSULR) provides a means for updating the low signature value via software. It may be written via a **mtdcr psulr, Rs** instruction (DCR register 278) to cause signature accumulation to occur in the parallel signature low register (PSLR) using the data value written. This register is write-only; attempted reads return a value of all zeros. Writing to this register will also cause the PSCTR to increment.

Low Signature Update Data
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

DCR - 278; Write-only; Reset -Unaffected

Figure 12-34. Parallel Signature Update Low Register (PSULR)

Chapter 13 Nexus 3 Module

The e200z759n3 Nexus 3 module provides real-time development capabilities for Zen processors in compliance with the IEEE-ISTO Nexus 5001 standard. This module provides development support capabilities without requiring the use of address and data pins for internal visibility.

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

13.1 Introduction

13.1.1 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 standard. The development features supported are Program Trace, Data Trace, Watchpoint Messaging, Ownership Trace, Data Acquisition Messaging, and Read/Write Access via the JTAG interface. The Nexus 3 module also supports two Class 4 features: Watchpoint Triggering, and Processor Overrun Control.

13.1.2 Terms and definitions

Table 13-1 contains a set of terms and definitions associated with the Nexus 3 module.

Table 13-1. Terms and definitions

Term	Description
IEEE-ISTO 5001	Consortium & 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.
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.
Data Acquisition Messaging (DQM)	Data Acquisition Messaging (DQM) allows code to be instrumented to export customized information to the Nexus Auxiliary Output Port.
JTAG compliant	Device complying to IEEE 1149.1 JTAG standard
JTAG IR & DR sequence	JTAG Instruction Register (IR) scan to load an opcode value for selecting a development register. The JTAG IR corresponds to the OnCE command register (OCMD). The selected development register is then accessed via a JTAG Data Register (DR) scan.

Table 13-1. Terms and definitions (continued)

Term	Description
Nexus1	The Zen (OnCE) debug module. This module integrated with each Zen processor provides all static (core halted) debug functionality. This module is compliant with Class1 of the IEEE-ISTO 5001 standard.
Ownership Trace Message (OTM)	Visibility of process/function that is currently executing.
Public messages	Messages on the auxiliary pins for accomplishing common visibility and controllability requirements
SoC	“System-on-a-Chip”. SoC signifies all of the modules on a single die. This generally includes one or more processors with associated peripherals, interfaces & memory modules.
Standard	The phrase “according to the standard” is used to indicate according to the IEEE-ISTO 5001 standard.
Transfer Code (TCODE)	Message header that identifies the number and/or size of packets to be transferred, and how to interpret each of the packets.
Watchpoint	A Data or Instruction Breakpoint or other debug event that does not cause the processor to halt. Instead, a pin is used to signal that the condition occurred. A Watchpoint message may also be generated.

13.1.3 Feature list

The Nexus 3 module is compliant with Class 3 of the IEEE-ISTO 5001-2008 standard, with additional Class 4 features available. The following features are implemented:

- Program Trace via Branch Trace Messaging (BTM). Branch trace messaging displays program flow discontinuities (direct and indirect branches, exceptions, etc.), allowing the development tool to interpolate what transpires between the discontinuities. Thus static code may be traced
- Data Trace via Data Write Messaging (DWM) and Data Read Messaging (DRM). This provides the capability for the development tool to trace reads and/or writes to selected internal memory resources
- Ownership Trace via Ownership Trace Messaging (OTM). OTM facilitates ownership trace by providing visibility of which process ID or operating system task is activated. An Ownership Trace Message (OTM) is transmitted when a new process/task is activated, allowing the development tool to trace ownership flow
- Run-time access to embedded processor memory map via the JTAG port. This allows for enhanced download/upload capabilities
- Watchpoint Messaging via the auxiliary pins
- Watchpoint Trigger enable of Program and/or Data Trace Messaging
- Data Acquisition Messaging (DQM) allows code to be instrumented to export customized information to the Nexus Auxiliary Output Port
- Address Translation Messaging via program correlation messages displays updates to the TLB for use by the debugger in correlating virtual and physical address information
- Auxiliary interface for higher data input/output

  - Configurable (min/max) Message Data Out pins (**nex_mdo[n:0]**)
  - One (1) or two (2) Message Start/End Out pins (**nex_mseo_b[1:0]**)
  - One (1) Read/Write Ready pin (**nex_rdy_b**) pin
  - One (1) Watchpoint Event output pin (**nex_evto_b**)
  - Four (4) additional Watchpoint Event output pins (**nex_wevto[3:0]**) for SoC use
  - One (1) Event In pin (**nex_evti_b**)
  - One (1) MCKO (Message Clock Out) pin
- Registers for Program Trace, Data Trace, Ownership Trace and Watchpoint Trigger
- All features controllable and configurable via the JTAG port

NOTE

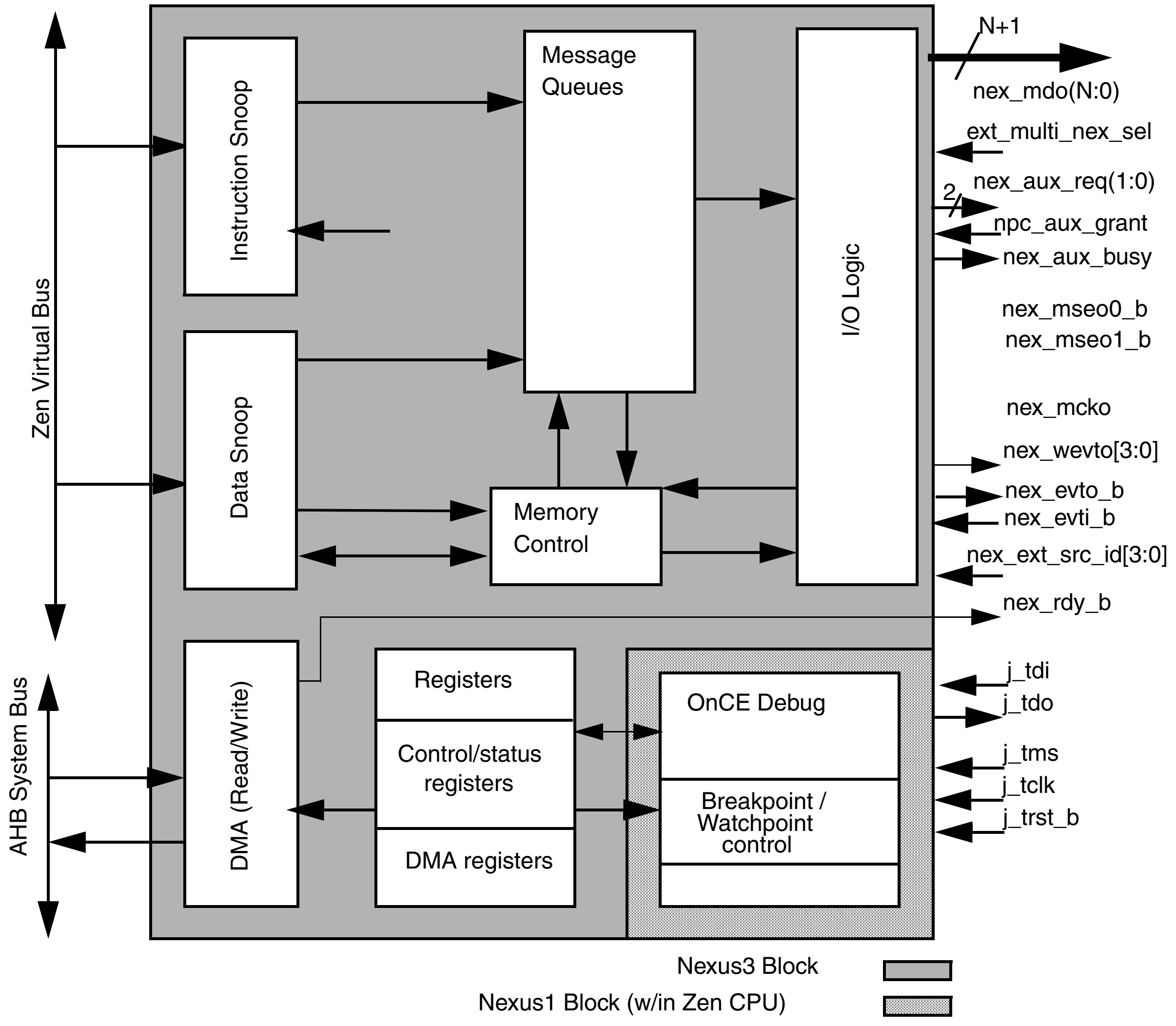
For multi-Nexus implementations, the configuration of the Message Data Out pins is controlled by the Port Control Register (at the SoC level). For single Nexus implementations, this configuration is controlled by Development Control Register 1 (DC1) within the Nexus 3 module.

In either implementation, Full Port Mode (FPM — maximum number of MDO pins) or Reduced Port Mode (RPM — minimum number of MDO pins) are 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 will be hard-wired based on SoC bandwidth requirements.

13.1.4 Functional block diagram



Note: The “nex_aux_req[1:0]”, “npc_aux_grant” & “nex_aux_busy” signals are used for inter-module communication in a multi-Nexus environment. They are not pins on the SoC.

Figure 13-1. Nexus 3 functional block diagram

13.2 Enabling Nexus 3 operation

The Nexus module is enabled by loading a single instruction (*NEXUS3-ACCESS*) into the JTAG Instruction Register (IR) (OnCE OCMD register). For the Nexus3 module, the OCMD value is 0b0001111100. Once enabled, the module will be ready to accept control input via the JTAG/OnCE pins.

Enabling the Nexus 3 module automatically enables the generation of Debug Status messages.

The Nexus module is disabled when the JTAG state machine reaches the Test-Logic-Reset state. This state can be reached by the assertion of the **j_trst_b** pin or by cycling through the state machine using the **j_tms** pin. The Nexus module will also be disabled if a Power-on-Reset (POR) event occurs. If the Nexus 3 module is disabled, no trace output will be provided, and the module will disable (drive inactive) auxiliary port output pins (**nex_mdo[n:0]**, **nex_mseo[1:0]**, **nex_mcko**). Nexus registers will not be available for reads or writes.

NOTE

Please refer to the "Nexus 3 Integration Guide" for details on IEEE-ISTO 5001 compliance with respect to output pins and multiple Nexus module configurations.

13.3 TCODEs supported

The Nexus 3 pins allow for flexible transfer operations via Public messages. A TCODE defines the transfer format, the number and/or size of the packets to be transferred, and the purpose of each packet. The IEEE-ISTO 5001-2008 standard defines a set of public messages and allocates additional TCODEs for vendor-specific features outside the scope of the public messages. The Nexus 3 block supports the TCODEs shown in Table 13-2.

Table 13-2. Supported TCODEs

Message name	Min. field size (bits)	Max. field size (bits)	Field name	Field type	Field description
Debug Status	6	6	TCODE	fixed	TCODE number = 0
	4	4	SRC	fixed	Source processor identifier
	8	8	STATUS	fixed	Debug Status Register (DS[31:24])
Ownership Trace Message	6	6	TCODE	fixed	TCODE number = 2
	4	4	SRC	fixed	Source processor identifier
	1	12	PROCESS	variable	Task/Process ID tag
Program Trace - Direct Branch Message	6	6	TCODE	fixed	TCODE number = 3
	4	4	SRC	fixed	Source processor identifier
	1	8	ICNT	variable	# sequential instructions completed since last predicate instruction, transmitted instruction count, or taken change of flow
Program Trace - Indirect Branch Message	6	6	TCODE	fixed	TCODE number = 4
	4	4	SRC	fixed	Source processor identifier
	1	1	MAP	fixed	Address Space (IS) indicator
	1	8	ICNT	variable	# sequential instructions completed since last predicate instruction, transmitted instruction count, or taken change of flow
	1	32	U-ADDR	variable	Unique part of target address for taken branches/exceptions

Table 13-2. Supported TCODEs (continued)

Message name	Min. field size (bits)	Max. field size (bits)	Field name	Field type	Field description
Data Trace - Data Write Message	6	6	TCODE	fixed	TCODE number = 5
	4	4	SRC	fixed	Source processor identifier
	1	1	MAP	fixed	Address Space (DS) indicator
	4	4	DSZ	fixed	Data size (Refer to Table 13-7)
	1	32	U-ADDR	variable	Unique portion of the data write address
	1	64	DATA	variable	Data write value(s) (see Data Trace section for details)
Data Trace - Data Read Message	6	6	TCODE	fixed	TCODE number = 6
	4	4	SRC	fixed	Source processor identifier
	1	1	MAP	fixed	Address Space (DS) indicator
	4	4	DSZ	fixed	Data size (Refer to Table 13-7)
	1	32	U-ADDR	variable	Unique portion of the data read address
	1	64	DATA	variable	Data read value(s) (see Data Trace section for details)
Data Acquisition Message	6	6	TCODE	fixed	TCODE number = 7
	4	4	SRC	fixed	Source processor identifier
	8	8	DQTAG	fixed	Identification tag taken from $\text{DEVENT}_{\text{DQTAG}}$ register field
	1	32	DQDATA	variable	Exported data taken from DDAM register
Error Message	6	6	TCODE	fixed	TCODE number = 8
	4	4	SRC	fixed	Source processor identifier
	4	4	ETYPE	fixed	Error type
	8	8	ECODE	fixed	Error code
Program Trace - Direct Branch Message w/ Sync	6	6	TCODE	fixed	TCODE number = 11
	4	4	SRC	fixed	Source processor identifier
	1	8	ICNT	variable	# sequential instructions completed since last predicate instruction, transmitted instruction count, or taken change of flow
	1	32	F-ADDR	variable	Full target address (leading zeros truncated)

Table 13-2. Supported TCODEs (continued)

Message name	Min. field size (bits)	Max. field size (bits)	Field name	Field type	Field description
Program Trace - Indirect Branch Message w/ Sync	6	6	TCODE	fixed	TCODE number = 12
	4	4	SRC	fixed	Source processor identifier
	1	1	MAP	fixed	Address Space (IS) indicator
	1	8	ICNT	variable	# sequential instructions completed since last predicate instruction, transmitted instruction count, or taken change of flow
	1	32	F-ADDR	variable	Full target address (leading zeros truncated)
Data Trace - Data Write Message w/ Sync	6	6	TCODE	fixed	TCODE number = 13
	4	4	SRC	fixed	Source processor identifier
	1	1	MAP	fixed	Address Space (DS) indicator
	4	4	DSZ	fixed	Data size (Refer to Table 13-7)
	1	32	F-ADDR	variable	Full access address (leading zeros truncated)
	1	64	DATA	variable	Data write value(s) (see Data Trace section for details)
Data Trace - Data Read Message w/ Sync	6	6	TCODE	fixed	TCODE number = 14
	4	4	SRC	fixed	Source processor identifier
	1	1	MAP	fixed	Address Space (DS) indicator
	4	4	DSZ	fixed	Data size (Refer to Table 13-7)
	1	32	F-ADDR	variable	Full access address (leading zeros truncated)
	1	64	DATA	variable	Data read value(s) (see Data Trace section for details)
Watchpoint Message	6	6	TCODE	fixed	TCODE number = 15
	4	4	SRC	fixed	Source processor identifier
	1	32	WPHIT	variable	Field indicating watchpoint source(s) (leading zeros truncated)
Resource Full Message	6	6	TCODE	fixed	TCODE number = 27
	4	4	SRC	fixed	Source processor identifier
	4	4	RCODE	fixed	Resource code (Refer to Table 13-5) - indicates which resource is the cause of this message
	1	32	RDATA	variable	Branch / predicate instruction history (see Section 13.11.4, Resource Full Messages)

Table 13-2. Supported TCODEs (continued)

Message name	Min. field size (bits)	Max. field size (bits)	Field name	Field type	Field description
Program Trace - Indirect Branch History Message	6	6	TCODE	fixed	TCODE number = 28 (see Note below)
	4	4	SRC	fixed	Source processor identifier
	1	1	MAP	fixed	Address Space (IS) indicator
	1	8	I-CNT	variable	# sequential instructions completed since last predicate instruction, transmitted instruction count, or taken change of flow
	1	32	U-ADDR	variable	Unique part of target address for taken branches/exceptions
	1	32	HIST	variable	Branch / predicate instruction history (see Section 13.11.1, Branch Trace messaging types)
Program Trace - Indirect Branch History Message w/ Sync	6	6	TCODE	fixed	TCODE number = 29 (see Note below)
	4	4	SRC	fixed	Source processor identifier
	1	1	MAP	fixed	Address Space (IS) indicator
	1	8	I-CNT	variable	# sequential instructions completed since last predicate instruction, transmitted instruction count, or taken change of flow
	1	32	F-ADDR	variable	Full target address (leading zero (0) truncated)
	1	32	HIST	variable	Branch / predicate instruction history (see Section 13.11.1, Branch Trace messaging types)

Table 13-2. Supported TCODEs (continued)

Message name	Min. field size (bits)	Max. field size (bits)	Field name	Field type	Field description
Program Trace - Program Correlation Message	6	6	TCODE	fixed	TCODE number = 33
	4	4	SRC	fixed	Source processor identifier
	4	4	EVCODE	fixed	Event correlated w/ program flow (Refer to Table 13-6)
	2	2	CDF	fixed	# fields of information in CDATA. 00 Reserved 01 One field (CDATA1) (reserved) 10 Two fields (CDATA1 + CDATA2) 11 Three fields (CDATA1 + CDATA2 + CDATA3)
	1	8	I-CNT	variable	# sequential instructions completed since last predicate instruction, transmitted instruction count, or taken change of flow
	1	32	CDATA1	variable	Correlation data field 1 - [branch / predicate instruction history or TLB info part1] (see Section 13.11.5, Program Correlation Messages (PCM))
	0	32	CDATA2	variable	Correlation data field 2- PID/IS info or TLB info (F-ADDR_V for virtual address or tlbivax EA) (see Section 13.11.5, Program Correlation Messages (PCM))
	0	32	CDATA3	variable	Correlation data field 3 - TLB info -ADDR_P for physical address (see Section 13.11.5, Program Correlation Messages (PCM))

NOTE

Program Trace can be implemented using either Branch History/Predicate Instruction messages, or traditional Direct/Indirect Branch messages. The user can select between the two types of Program Trace. The advantages for each are discussed in Section 13.11.1, Branch Trace messaging types. If the Branch History method is selected, the shaded TCODES above will not be messaged out.

Table 13-3 shows the error code encodings used when reporting an error via the Nexus 3 Error message.

Table 13-3. Error code encoding (TCODE = 8)

Error code	Description
xxxxxxx1	Watchpoint Trace Message(s) Lost
xxxxxx1x	Data Trace Message(s) Lost
xxxxx1xx	Program Trace Message(s) Lost
xxxx1xxx	Ownership Trace Message(s) Lost
xxx1xxxx	Status Message(s) Lost (Debug Status messages, etc.)
xx1xxxxx	Data Acquisition Message(s) Lost

Table 13-3. Error code encoding (TCODE = 8) (continued)

Error code	Description
x1xxxxxx	Reserved
1xxxxxxx	Reserved

Table 13-4 shows the error type encodings used when reporting an error via the Nexus 3 Error message.

Table 13-4. Error type encoding (TCODE = 8)

Error type	Description
0000	Message Queue Overrun caused one or more messages to be lost
0001	Contention with higher priority messages caused one or more messages to be lost
0010	Reserved
0011	Read/write access error
0100	Reserved
0101	Invalid access opcode (Nexus Register unimplemented)
0110 - 1111	Reserved

Table 13-5 shows the encodings used for resource codes for certain messages.

Table 13-5. RCODE values (TCODE = 27)

Resource code	Description
0000	Program Trace Instruction counter reached 255 and was reset.
0001	Program Trace, Branch / Predicate Instruction History full. This type of packet is terminated by a stop bit set to 1 after the last history bit.

Table 13-6 shows the event code encodings used for certain messages.

Table 13-6. Event code encoding (TCODE = 33)

Event code	Description
0000	Entry into Debug Mode
0001	Entry into Low Power Mode (CPU only)
0010-0011	Reserved for future functionality
0100	Disabling Program Trace
0101	New process ID value is established in PID0 via **mtspr PID0**, or new value for MSR_{IS} is established via a **mtmsr** instruction
0110-1000	Reserved for future functionality
1001	Begin masking of program trace messages due to MSR_{PMM}=0 and $DC4_{PTMARK}$=1
1010	Branch and link occurrence (direct branch function call)

Table 13-6. Event code encoding (TCODE = 33) (continued)

Event code	Description
1011	New Address Translation established in the TLB via **tlbwe**
1100	Address Translation entries invalidated in the TLB via **tlbivax**
1101	Reserved for future functionality
1110	End of BookE tracing (trace disable or entry into a VLE page from a non-VLE page)[1]
1111	End of VLE tracing (trace disabled or entry into a non-VLE page from a VLE page)[1]

[1] If Event Code 1010 is not masked, a PCM for this Event will not be generated if the event is due to a branch and link.

Table 13-7 shows the data trace size encodings used for certain messages.

Table 13-7. Data trace size encodings (TCODE = 5,6,13,14)

DTM size encoding	Transfer size
0000	0 - no data
0001	Byte
0010	Halfword (2 bytes)
0011	Three bytes
0100	Word (4 bytes)
0101	Five bytes
0110	Six bytes
0111	Seven bytes
1000	Doubleword (8 bytes)
1001-1111	Reserved

13.4 Nexus 3 programmer's model

This section describes the Nexus 3 programmers model. Nexus 3 registers are accessed using the JTAG/OnCE port in compliance with IEEE 1149.1. See Section 13.5, Nexus 3 register access via JTAG/OnCE for details on Nexus 3 register access.

NOTE

Nexus 3 registers and output signals are numbered using bit 0 as the least significant bit. This bit ordering is consistent with the ordering defined by the IEEE-ISTO 5001 standard.

Table 13-8 details the register map for the Nexus 3 module.

Table 13-8. Nexus 3 register map

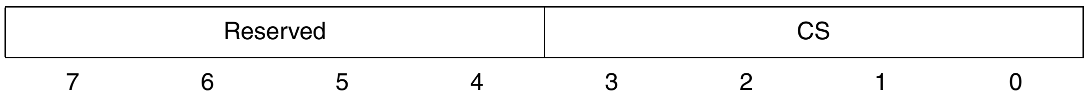
Nexus register	Nexus access opcode	Read/ write	Read address	Write address
Client Select Control (CSC)[1]	0x1	R	0x02	—
Port Configuration Register (PCR)[1]	PCR_INDEX[2]	R/W	—	—
Development Control 1 (DC1)	0x2	R/W	0x04	0x05
Development Control 2 (DC2)	0x3	R/W	0x06	0x07
Development Control 3 (DC3)	0x4	R/W	0x08	0x09
Development Control 4 (DC4)	0x5	R/W	0x0A	0x0B
Read/Write Access Control/Status (RWCS)	0x7	R/W	0x0E	0x0F
Read/Write Access Address (RWA)	0x9	R/W	0x12	0x13
Read/Write Access Data (RWD)	0xA	R/W	0x14	0x15
Watchpoint Trigger (WT)	0xB	R/W	0x16	0x17
Reserved	0xC	R/W	0x18	0x19
Data Trace Control (DTC)	0xD	R/W	0x1A	0x1B
Data Trace Start Address 1 (DTSA1)	0xE	R/W	0x1C	0x1D
Data Trace Start Address 2 (DTSA2)	0xF	R/W	0x1E	0x1F
Data Trace Start Address 3 (DTSA3)	0x10	R/W	0x20	0x21
Data Trace Start Address 4 (DTSA4)	0x11	R/W	0x22	0x23
Data Trace End Address 1 (DTEA1)	0x12	R/W	0x24	0x25
Data Trace End Address 2 (DTEA2)	0x13	R/W	0x26	0x27
Data Trace End Address 3 (DTEA3)	0x14	R/W	0x28	0x29
Data Trace End Address 4 (DTEA4)	0x15	R/W	0x2A	0x2B
Reserved	0x16 → 0x2F	—	0x28 → 0x5E	0x29 → 5F
Development Status (DS)	0x30	R	0x60	-
Reserved	0x31	R/W	0x62	0x63
Overrun Control (OVCR)	0x32	R/W	0x64	0x65
Watchpoint Mask (WMSK)	0x33	R/W	0x66	0x67
Reserved	0x34	—	0x68	0x69
Program Trace Start Trigger Control (PTSTC)	0x35	R/W	0x6A	0x6B
Program Trace End Trigger Control (PTETC)	0x36	R/W	0x6C	0x6D
Data Trace Start Trigger Control (DTSTC)	0x37	R/W	0x6E	0x6F
Data Trace End Trigger Control (DTETC)	0x38	R/W	0x70	0x71
Reserved	0x39 → 0x3F	—	0x72 → 0x7E	0x73 → 7F

[1] The CSC and PCR registers are shown in this table as part of the Nexus programmer's model. They are only present at the top level SoC Nexus controller in a multi-Nexus implementation, not in the Nexus 3 module. The SoC's CSC Register is readable through Nexus, but the PCR is shown for reference only here.

2 The "PCR_INDEX" is a parameter determined by the SoC. Refer to the "Zen Nexus 3 Integration Guide" for more information on how this parameter is implemented for each Nexus module.

13.4.1 Client Select Control register (CSC)

The Client Select Control register (CSC) determines which Nexus client is under development. This register is present at the top-level SOC Nexus 3 controller to select one of multiple on-chip Nexus 3 units.



Nexus Reg# - 0x1; Read-only; Reset - 0x0

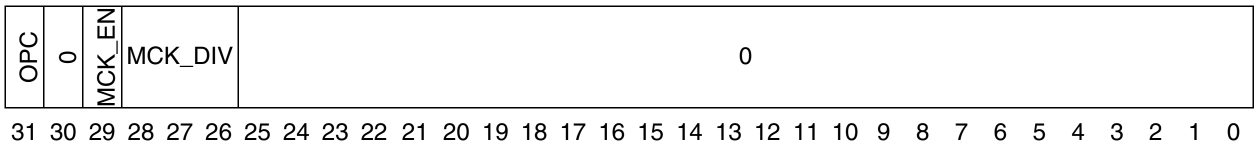
Figure 13-2. Client Select Control register (CSC)

Table 13-9. CSC field descriptions

Field	Description
CSC[7:4]	RES - Reserved for future Nexus Clients (read as 0)
CSC[3:0]	Client Select Control 0xX - Nexus client (SoC level)

13.4.2 Port Configuration Register (PCR) — reference only

The Port Configuration Register (PCR) controls the basic port functions for all Nexus modules in a multi-Nexus environment. This includes clock control and auxiliary port width. All bits in this register are writable only once after system reset.



Nexus Reg# - PCR_INDEX; Read/Write; Reset - 0x0

Figure 13-3. Port Configuration Register (PCR)

Table 13-10. PCR field descriptions

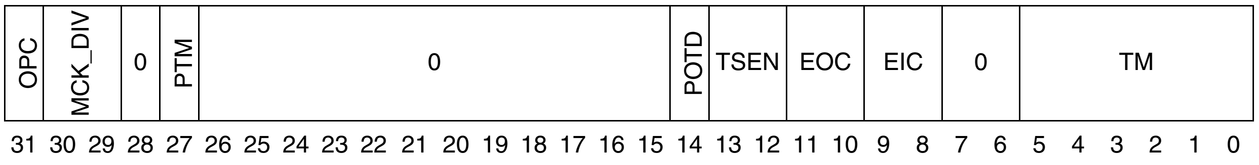
Bit	Name	Description
31	OPC	Output Port Mode Control (SoC Level) 0 Reduced Port Mode configuration (min# **nex_mdo[n:0]** pins defined by SOC) 1 Full Port Mode configuration (max# **nex_mdo[n:0]** pins defined by SOC)
30	—	Reserved for future functionality
29	MCK_EN	MCKO Clock Enable (SoC Level) 0 **nex_mcko** is disabled 1 **nex_mcko** is enabled
28:26	MCK_DIV	MCKO Clock Divide Ratio (see note below) (SoC Level) 000**nex_mcko** is 1x processor clock freq. 001 **nex_mcko** is 1/2x processor clock freq. 010 Reserved (default to 1/2x processor clock freq.) 011 **nex_mcko** is 1/4x processor clock freq. 100–110 Reserved (default to 1/2x processor clock freq.) 111 **nex_mcko** is 1/8x processor clock freq.
25:0	—	Reserved for future functionality

NOTE

The CSC and PCR Registers exist in a separate module at the SoC level in a multi-Nexus environment. If the Zen Nexus 3 module is the only Nexus module, these registers are not implemented and the Zen Nexus 3 defined Development Control Register 1 (DC1) is used to control the SoC-level Nexus port functionality.

13.4.3 Nexus Development Control Register 1 (DC1)

Nexus Development Control Register 1 is used to control the basic development features of the Nexus 3 module. Development Control Register 1 is shown in Figure 13-4 and its fields are described in Table 13-11.



Nexus Reg# - 0x2; Read/Write; Reset - 0x0

Figure 13-4. Development Control Register 1 (DC1)

Table 13-11. DC1 field descriptions

Bits	Name	Description

Table 13-11. DC1 field descriptions (continued)

31	OPC	Output Port Mode Control 0 Reduced Port Mode configuration (min# **nex_mdo[n:0]** pins defined 1 Full Port Mode configuration (max# **nex_mdo[n:0]** pins defined
30:29	MCK_DIV	MCKO Clock Divide Ratio (see note below) 00 **nex_mcko** is 1x processor clock 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	—	Reserved for future functionality
27	PTM	PTM - Program Trace Method 0 Program Trace uses traditional branch messages 1 Program Trace uses Branch History messages
26:15	—	Reserved for future functionality
14	POTD	Periodic Ownership Trace Disable 0 Periodic Ownership Trace message events are enabled 1 Periodic Ownership Trace message events are disabled
13:12	TSEN	Timestamp Enable - (not implemented, write to 00) 00 Timestamp is disabled
11:10	EOC	EVTO Control 00 **nex_evto_b** upon occurrence of Watchpoints (configured in DC2 and DC3) 01 **nex_evto_b** upon entry into Debug Mode 1x Reserved
9:8	EIC	EVTI Control 00 **nex_evti_b** is used for synchronization (Program Trace/ Data Trace) 01 **nex_evti_b** is used for Debug request 1x Reserved
7:6	—	Reserved for future functionality
5:0	TM	Trace Mode[1] 000000 All Trace Disabled xxxxx1 Ownership Trace enabled xxxx1x Data Trace enabled xxx1xx Program Trace enabled xx1xxx Watchpoint Trace enabled x1xxxx Reserved 1xxxxx Data Acquisition Trace enabled

[1] This field may be updated by hardware in response to watchpoint triggering. Writes to this field take precedence over hardware updates in the event of a collision. Refer to Section 13.4.7, Watchpoint Trigger registers (WT, PTSTC, PTETC, DTSTC, DTETC) for more information on watchpoint triggering.

NOTE

The Output Port Mode Control bit (OPC) and MCKO Clock Divide Ratio bits (MCK_DIV) MUST ONLY be modified during system reset or debug mode to insure correct output port and output clock functionality. It is also recommended that all other bits of the DC1 also only be modified in one of these two modes.

13.4.4 Nexus Development Control Registers 2 and 3 (DC2, DC3)

Nexus Development Control Registers 2 and 3 are used to control output signaling on the Nexus 3 module. A table of watchpoints can be found in Table 12-28.

Development Control Register 2 is shown in Figure 13-5 and its fields are described in Table 13-12.

WEVTO[3]C	WEVTO[2]C	WEVTO[1]C	WEVTO[0]C	EWC
31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Nexus Reg# - 0x3; Read/Write; Reset - 0x0

Figure 13-5. Development Control Register 2 (DC2)

Table 13-12. DC2 field descriptions

Bits	Name	Description
31:28	WEVTO[3]C	Watchpoint Event Out 3 Configuration 0000 No Watchpoints #0-14 trigger **nex_wevto[3]** 0001 Watchpoint #0 triggers **nex_wevto[3]** 0010 Watchpoint #1 triggers **nex_wevto[3]** 0011 Watchpoint #2 triggers **nex_wevto[3]** 0100 Watchpoint #3 triggers **nex_wevto[3]** 0101 Watchpoint #4 triggers **nex_wevto[3]** 0110 Watchpoint #5 triggers **nex_wevto[3]** 0111 Watchpoint #6 triggers **nex_wevto[3]** 1000 Watchpoint #7 triggers **nex_wevto[3]** 1001 Watchpoint #8 triggers **nex_wevto[3]** 1010 Watchpoint #9 triggers **nex_wevto[3]** 1011 Watchpoint #10 triggers **nex_wevto[3]** 1100 Watchpoint #11 triggers **nex_wevto[3]** 1101 Watchpoint #12 triggers **nex_wevto[3]** 1110 Watchpoint #13 triggers **nex_wevto[3]** 1111 Watchpoint #14 triggers **nex_wevto[3]**
27:24	WEVTO[2]C	Watchpoint Event Out 2 Configuration 0000 No Watchpoints #0-14 trigger **nex_wevto[2]** 0001 Watchpoint #0 triggers **nex_wevto[2]** 0010 Watchpoint #1 triggers **nex_wevto[2]** 0011 Watchpoint #2 triggers **nex_wevto[2]** 0100 Watchpoint #3 triggers **nex_wevto[2]** 0101 Watchpoint #4 triggers **nex_wevto[2]** 0110 Watchpoint #5 triggers **nex_wevto[2]** 0111 Watchpoint #6 triggers **nex_wevto[2]** 1000 Watchpoint #7 triggers **nex_wevto[2]** 1001 Watchpoint #8 triggers **nex_wevto[2]** 1010 Watchpoint #9 triggers **nex_wevto[2]** 1011 Watchpoint #10 triggers **nex_wevto[2]** 1100 Watchpoint #11 triggers **nex_wevto[2]** 1101 Watchpoint #12 triggers **nex_wevto[2]** 1110 Watchpoint #13 triggers **nex_wevto[2]** 1111 Watchpoint #14 triggers **nex_wevto[2]**

Table 13-12. DC2 field descriptions (continued)

23:20	WEVTO[1]C	Watchpoint Event Out 1 Configuration 0000 No Watchpoints #0-14 trigger **nex_wevto[1]** 0001 Watchpoint #0 triggers **nex_wevto[1]** 0010 Watchpoint #1 triggers **nex_wevto[1]** 0011 Watchpoint #2 triggers **nex_wevto[1]** 0100 Watchpoint #3 triggers **nex_wevto[1]** 0101 Watchpoint #4 triggers **nex_wevto[1]** 0110 Watchpoint #5 triggers **nex_wevto[1]** 0111 Watchpoint #6 triggers **nex_wevto[1]** 1000 Watchpoint #7 triggers **nex_wevto[1]** 1001 Watchpoint #8 triggers **nex_wevto[1]** 1010 Watchpoint #9 triggers **nex_wevto[1]** 1011 Watchpoint #10 triggers **nex_wevto[1]** 1100 Watchpoint #11 triggers **nex_wevto[1]** 1101 Watchpoint #12 triggers **nex_wevto[1]** 1110 Watchpoint #13 triggers **nex_wevto[1]** 1111 Watchpoint #14 triggers **nex_wevto[1]**
19:16	WEVTO[0]C	Watchpoint Event Out 0 Configuration 0000 No Watchpoints #0-14 trigger **nex_wevto[0]** 0001 Watchpoint #0 triggers **nex_wevto[0]** 0010 Watchpoint #1 triggers **nex_wevto[0]** 0011 Watchpoint #2 triggers **nex_wevto[0]** 0100 Watchpoint #3 triggers **nex_wevto[0]** 0101 Watchpoint #4 triggers **nex_wevto[0]** 0110 Watchpoint #5 triggers **nex_wevto[0]** 0111 Watchpoint #6 triggers **nex_wevto[0]** 1000 Watchpoint #7 triggers **nex_wevto[0]** 1001 Watchpoint #8 triggers **nex_wevto[0]** 1010 Watchpoint #9 triggers **nex_wevto[0]** 1011 Watchpoint #10 triggers **nex_wevto[0]** 1100 Watchpoint #11 triggers **nex_wevto[0]** 1101 Watchpoint #12 triggers **nex_wevto[0]** 1110 Watchpoint #13 triggers **nex_wevto[0]** 1111 Watchpoint #14 triggers **nex_wevto[0]**
15:0	EWC	EVTO Watchpoint Configuration[1] 0000000000000000 No Watchpoints #0-15 trigger **nex_evto_b** *xxxxxxxxxxxxxxx*1 Watchpoint #0 triggers **nex_evto_b** *xxxxxxxxxxxxxx*1*x* Watchpoint #1 triggers **nex_evto_b** *xxxxxxxxxxxxx*1*xx* Watchpoint #2 triggers **nex_evto_b** *xxxxxxxxxxxx*1*xxx* Watchpoint #3 triggers **nex_evto_b** *xxxxxxxxxxx*1*xxxx* Watchpoint #4 triggers **nex_evto_b** *xxxxxxxxxx*1*xxxxx* Watchpoint #5 triggers **nex_evto_b** *xxxxxxxxx*1*xxxxxx* Watchpoint #6 triggers **nex_evto_b** *xxxxxxxx*1*xxxxxxx* Watchpoint #7 triggers **nex_evto_b** *xxxxxxx*1*xxxxxxxx* Watchpoint #8 triggers **nex_evto_b** *xxxxxx*1*xxxxxxxxx* Watchpoint #9 triggers **nex_evto_b** *xxxxx*1*xxxxxxxxxx* Watchpoint #10 triggers **nex_evto_b** *xxxx*1*xxxxxxxxxxx* Watchpoint #11 triggers **nex_evto_b** *xxx*1*xxxxxxxxxxxx* Watchpoint #12 triggers **nex_evto_b** *xx*1*xxxxxxxxxxxxx* Watchpoint #13 triggers **nex_evto_b** *x*1*xxxxxxxxxxxxxx* Watchpoint #14 triggers **nex_evto_b** 1*xxxxxxxxxxxxxxx* Watchpoint #15 triggers **nex_evto_b**

[1] The EOC bits in DC1 must be programmed to trigger $\overline{\text{EVTO}}$ on Watchpoint occurrence for the EWC bits to have any effect.

Development Control Register 3 (DC3) is shown in Figure 13-6 and its fields are described in Table 13-13.

WEVTO[3]C	WEVTO[2]C	WEVTO[1]C	WEVTO[0]C	0	EWC
31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14	13 12 11 10 9 8 7 6 5 4 3 2 1 0

Nexus Reg# - 0x4; Read/Write; Reset - 0x0

Figure 13-6. Development Control Register 3 (DC3)

Table 13-13. DC3 field descriptions

Bits	Name	Description
31:28	WEVTO[3]C	Watchpoint Event Out 3 Configuration 0000 No Watchpoints #15-#26 trigger **nex_wevto[3]** 0001 Watchpoint #15 triggers **nex_wevto[3]** 0010 Watchpoint #16 triggers **nex_wevto[3]** 0011 Watchpoint #17 triggers **nex_wevto[3]** 0100 Watchpoint #18 triggers **nex_wevto[3]** 0101 Watchpoint #19 triggers **nex_wevto[3]** 0110 Watchpoint #20 triggers **nex_wevto[3]** 0111 Watchpoint #21 triggers **nex_wevto[3]** 1000 Watchpoint #22 triggers **nex_wevto[3]** 1001 Watchpoint #23 triggers **nex_wevto[3]** 1010 Watchpoint #24 triggers **nex_wevto[3]** 1011 Watchpoint #25 triggers **nex_wevto[3]** 1100 Watchpoint #26 triggers **nex_wevto[3]** 1101 – 1111 Reserved
27:24	WEVTO[2]C	Watchpoint Event Out 2 Configuration 0000 No Watchpoints #15-#26 trigger **nex_wevto[2]** 0001 Watchpoint #15 triggers **nex_wevto[2]** 0010 Watchpoint #16 triggers **nex_wevto[2]** 0011 Watchpoint #17 triggers **nex_wevto[2]** 0100 Watchpoint #18 triggers **nex_wevto[2]** 0101 Watchpoint #19 triggers **nex_wevto[2]** 0110 Watchpoint #20 triggers **nex_wevto[2]** 0111 Watchpoint #21 triggers **nex_wevto[2]** 1000 Watchpoint #22 triggers **nex_wevto[2]** 1001 Watchpoint #23 triggers **nex_wevto[2]** 1010 Watchpoint #24 triggers **nex_wevto[2]** 1011 Watchpoint #25 triggers **nex_wevto[2]** 1100 Watchpoint #26 triggers **nex_wevto[2]** 1101 -– 1111 Reserved

Table 13-13. DC3 field descriptions (continued)

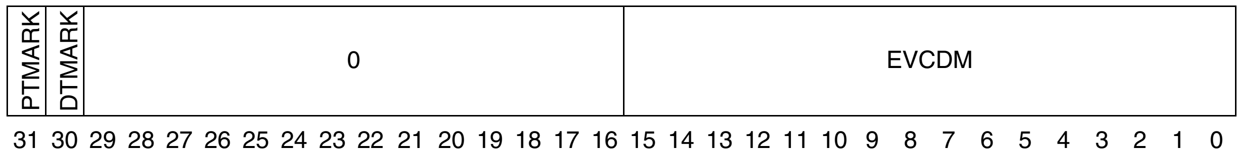
23:20	WEVTO[1]C	Watchpoint Event Out 1 Configuration 0000 No Watchpoints #15-#26 trigger **nex_wevto[1]** 0001 Watchpoint #15 triggers **nex_wevto[1]** 0010 Watchpoint #16 triggers **nex_wevto[1]** 0011 Watchpoint #17 triggers **nex_wevto[1]** 0100 Watchpoint #18 triggers **nex_wevto[1]** 0101 Watchpoint #19 triggers **nex_wevto[1]** 0110 Watchpoint #20 triggers **nex_wevto[1]** 0111 Watchpoint #21 triggers **nex_wevto[1]** 1000 Watchpoint #22 triggers **nex_wevto[1]** 1001 Watchpoint #23 triggers **nex_wevto[1]** 1010 Watchpoint #24 triggers **nex_wevto[1]** 1011 Watchpoint #25 triggers **nex_wevto[1]** 1100 Watchpoint #26 triggers **nex_wevto[1]** 1101 Watchpoint #27 triggers **nex_wevto[1]** 1110 Watchpoint #28 triggers **nex_wevto[1]** 1111 Watchpoint #29 triggers **nex_wevto[1]**
19:16	WEVTO[0]C	Watchpoint Event Out 0 Configuration 0000 No Watchpoints #15-#26 trigger **nex_wevto[0]** 0001 Watchpoint #15 triggers **nex_wevto[0]** 0010 Watchpoint #16 triggers **nex_wevto[0]** 0011 Watchpoint #17 triggers **nex_wevto[0]** 0100 Watchpoint #18 triggers **nex_wevto[0]** 0101 Watchpoint #19 triggers **nex_wevto[0]** 0110 Watchpoint #20 triggers **nex_wevto[0]** 0111 Watchpoint #21 triggers **nex_wevto[0]** 1000 Watchpoint #22 triggers **nex_wevto[0]** 1001 Watchpoint #23 triggers **nex_wevto[0]** 1010 Watchpoint #24 triggers **nex_wevto[0]** 1011 Watchpoint #25 triggers **nex_wevto[0]** 1100 Watchpoint #26 triggers **nex_wevto[0]** 1101 Watchpoint #27 triggers **nex_wevto[0]** 1110 Watchpoint #28 triggers **nex_wevto[0]** 1111 Watchpoint #29 triggers **nex_wevto[0]**
15:14	—	Reserved for watchpoint expansion
13:0	EWC	EVTO Watchpoint Configuration[1] 00000000000000 No Watchpoints #16-#29 trigger **nex_evto_b** *xxxxxxxxxxxxx*1 Watchpoint #16 triggers **nex_evto_b** *xxxxxxxxxxxx*1*x* Watchpoint #17 triggers **nex_evto_b** *xxxxxxxxxxx*1*xx* Watchpoint #18 triggers **nex_evto_b** *xxxxxxxxxx*1*xxx* Watchpoint #19 triggers **nex_evto_b** *xxxxxxxxx*1*xxxx* Watchpoint #20 triggers **nex_evto_b** *xxxxxxxx*1*xxxxx* Watchpoint #21 triggers **nex_evto_b** *xxxxxxx*1*xxxxxx* Watchpoint #22 triggers **nex_evto_b** *xxxxxx*1*xxxxxxx* Watchpoint #23 triggers **nex_evto_b** *xxxxx*1*xxxxxxxx* Watchpoint #24 triggers **nex_evto_b** *xxxx*1*xxxxxxxxx* Watchpoint #25 triggers **nex_evto_b** *xxx*1*xxxxxxxxxx* Watchpoint #26 triggers **nex_evto_b** *xx*1*xxxxxxxxxxx* Watchpoint #27 triggers **nex_evto_b** *x*1*xxxxxxxxxxxx* Watchpoint #28 triggers **nex_evto_b** 1*xxxxxxxxxxxxx* Watchpoint #29 triggers **nex_evto_b**

[1] The EOC bits in DC1 must be programmed to trigger $\overline{\text{EVTO}}$ on Watchpoint occurrence for the EWC bits to have any effect.

13.4.5 Nexus Development Control Register 4 (DC4)

Nexus Development Control Register 4 is used to control mark selection for Program and Data Trace Messaging, as well as masking of events that initiate Program Correlation messages on the Nexus 3 module.

Development Control Register 4 is shown in Figure 13-7 and its fields are described in Table 13-14.



Nexus Reg# - 0x5; Read/Write; Reset - 0x0

Figure 13-7. Development Control Register 4 (DC4)

Table 13-14. DC4 field descriptions

Bits	Name	Description
31	PTMARK	Program Trace Mark 0 Ignore MSR$_{PMM}$ for masking program trace messages 1 Mask program trace messages when MSR$_{PMM}$='0', unmask program trace messages when MSR$_{PMM}$='1'
30	DTMARK	Data Trace Mark 0 Ignore MSR$_{PMM}$ for masking data trace messages 1 Mask data trace messages when MSR$_{PMM}$='0', unmask data trace messages when MSR$_{PMM}$='1'
29:16	—	Reserved
15:0	EVCDM	Event Code (EVCODE) Mask[1] 0000000000000000 No EVCODEs masked for Program Correlation messages *xxxxxxxxxxxxxxx*1 EVCODE #0 is masked for Program Correlation messages *xxxxxxxxxxxxxx*1*x* EVCODE #1 is masked for Program Correlation messages *xxxxxxxxxxxxx*1*xx* EVCODE #2 is masked for Program Correlation messages *xxxxxxxxxxxx*1*xxx* EVCODE #3 is masked for Program Correlation messages *xxxxxxxxxxx*1*xxxx* EVCODE #4 is masked for Program Correlation messages *xxxxxxxxxx*1*xxxxx* EVCODE #5 is masked for Program Correlation messages *xxxxxxxxx*1*xxxxxx* EVCODE #6 is masked for Program Correlation messages *xxxxxxxx*1*xxxxxxx* EVCODE #7 is masked for Program Correlation messages *xxxxxxx*1*xxxxxxxx* EVCODE #8 is masked for Program Correlation messages *xxxxxx*1*xxxxxxxxx* EVCODE #9 is masked for Program Correlation messages *xxxxx*1*xxxxxxxxxx* EVCODE #10 is masked for Program Correlation messages *xxxx*1*xxxxxxxxxxx* EVCODE #11 is masked for Program Correlation messages *xxx*1*xxxxxxxxxxxx* EVCODE #12 is masked for Program Correlation messages *xx*1*xxxxxxxxxxxxx* EVCODE #13 is masked for Program Correlation messages *x*1*xxxxxxxxxxxxxx* EVCODE #14 is masked for Program Correlation messages 1*xxxxxxxxxxxxxxx* EVCODE #15 is masked for Program Correlation messages

[1] Refer to Table 13-6 for implemented EVCODEs

13.4.6 Development Status register (DS)

The Development Status Register is used to report system debug status. When Debug mode is entered or exited, or an SoC or Zen defined Low Power Mode is entered (see Note below), a Debug Status message is transmitted with DS[31:24]. The external tool can read this register at any time.

DBG	LPS			LPC		0																									
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Nexus Reg# - 0x4; Read-only; Reset - 0x0

Figure 13-8. Development Status (DS) register

Table 13-15. DS field descriptions

Bits	Name	Description
31	DBG	Zen CPU Debug Mode Status 0 CPU not in Debug mode 1 CPU in Debug mode (**jd_debug_b** signal asserted)
30:28	LPS	Zen System Low Power Mode Status 000 Normal (Run) mode *xx*1 DOZE mode (**p_doze** signal asserted) *x*1*x* NAP mode (**p_nap** signal asserted) 1*xx* SLEEP mode (**p_sleep** signal asserted)
27:26	LPC	Zen CPU Low Power Mode Status 00 Normal (Run) mode 01 CPU in Halted state (**p_halted** signal asserted) 10 CPU in Stopped state (**p_stopped** signal asserted) 11 CPU in Waiting state (**p_waiting** signal asserted)
25:0	—	Reserved for future functionality (read as 0)

13.4.7 Watchpoint Trigger registers (WT, PTSTC, PTETC, DTSTC, DTETC)

The Watchpoint Trigger Registers allows the watchpoints defined within the Zen Nexus1 logic to trigger actions. These watchpoints can control Program and/or Data Trace enable and disable. The control bits can be used to produce a related “window” for triggering trace messages.Watchpoint trigger register WT is used to control triggering by a single selected watchpoint. The Program Trace Start Trigger Control (PTSTC), Program Trace End Trigger Control (PTETC), Data Trace Start Trigger Control (DTSTC), and Data Trace End Trigger Control (DTETC) are used for extended trigger controls for the respective function. If multiple watchpoints are desired for triggering, or a watchpoint beyond watchpoint #13 is required, then one or more of the extended watchpoint trigger registers may be used. A field encoding of 4’b1111 in one of the WT register fields enables the corresponding extended trigger register. For all other WT field encodings, the corresponding extended trigger register is disabled and the contents are ignored.

When a start trigger is detected, the designated trace features become enabled, and the corresponding enable bits of the DC1 register are set. Whenever a stop trigger is detected, the designated trace features become disabled, and the corresponding enable bits of the DC1 register are cleared. If the same trigger condition is used for both start and stop triggering, then the designated trace features will toggle between being enabled and disabled at each occurrence of the trigger condition. Similarly, if start and stop triggers for a trace feature occur simultaneously, then the designated trace feature will toggle between enabled and disabled depending on the enable state at the time of the trigger events. For example, if tracing is enabled, and a start and stop trigger occur simultaneously, then tracing will be disabled. Direct writes of the DC1 register take precedence over any trace feature enable state that is derived from watchpoint triggering. A table of watchpoints can be found in Table 12-28.

PTS	PTE	DTS	DTE	0
31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Nexus Reg# - 0xB; Read/Write; Reset - 0x0

Figure 13-9. Watchpoint Trigger (WT) register

Table 13-16 details the Watchpoint Trigger register fields.

Table 13-16. WT field descriptions

Bits	Name	Description
31:28	PTS	Program Trace Start Control 0000 Trigger disabled 0001 Use Watchpoint #0 0010 Use Watchpoint #1 . . . 1110 Use Watchpoint #13 1111 Use control settings in the PTSTC register
27:24	PTE	Program Trace End Control 0000 Trigger disabled 0001 Use Watchpoint #0 0010 Use Watchpoint #1 . . . 1110 Use Watchpoint #13 1111 Use control settings in the PTETC register
23:20	DTS	Data Trace Start Control 0000 Trigger disabled 0001 Use Watchpoint #0 0010 Use Watchpoint #1 . . . 1110 Use Watchpoint #13 1111 Use control settings in the DTSTC register
19:16	DTE	Data Trace End Control 0000 Trigger disabled 0001 Use Watchpoint #0 0010 Use Watchpoint #1 . . . 1110 Use Watchpoint #13 1111 Use control settings in the DTETC register
15:0	—	Reserved for future functionality (read as 0)

For extended Program Trace start trigger control, the PTSTC register is used.

0	PTST
31 30	29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Nexus Reg# - 0x35; Read/Write; Reset - 0x0

Figure 13-10. Program Trace Start Trigger Control (PTSTC) register

Table 13-17 details the PTSTC register fields.

Table 13-17. PTSTC field descriptions

Bits	Name	Description
31:30	—	Reserved for future functionality (read as 0)
29:0	PTST	Program Trace Start Trigger Control 000000000000000000000000000000 Trigger disabled *xxxxxxxxxxxxxxxxxxxxxxxxxxxxx*1 Use Watchpoint #0 *xxxxxxxxxxxxxxxxxxxxxxxxxxxx*1*x* Use Watchpoint #1 *xxxxxxxxxxxxxxxxxxxxxxxxxxx*1*xx* Use Watchpoint #2 *xxxxxxxxxxxxxxxxxxxxxxxxxx*1*xxx* Use Watchpoint #3 *xxxxxxxxxxxxxxxxxxxxxxxxx*1*xxxx* Use Watchpoint #4 *xxxxxxxxxxxxxxxxxxxxxxxx*1*xxxxx* Use Watchpoint #5 *xxxxxxxxxxxxxxxxxxxxxxx*1*xxxxxx* Use Watchpoint #6 *xxxxxxxxxxxxxxxxxxxxxx*1*xxxxxxx* Use Watchpoint #7 *xxxxxxxxxxxxxxxxxxxxx*1*xxxxxxxx* Use Watchpoint #8 *xxxxxxxxxxxxxxxxxxxx*1*xxxxxxxxx* Use Watchpoint #9 *xxxxxxxxxxxxxxxxxxx*1*xxxxxxxxxx* Use Watchpoint #10 *xxxxxxxxxxxxxxxxxx*1*xxxxxxxxxxx* Use Watchpoint #11 *xxxxxxxxxxxxxxxxx*1*xxxxxxxxxxxx* Use Watchpoint #12 *xxxxxxxxxxxxxxxx*1*xxxxxxxxxxxxx* Use Watchpoint #13 *xxxxxxxxxxxxxxx*1*xxxxxxxxxxxxxx* Use Watchpoint #14 *xxxxxxxxxxxxxx*1*xxxxxxxxxxxxxxx* Use Watchpoint #15 *xxxxxxxxxxxxx*1*xxxxxxxxxxxxxxxx* Use Watchpoint #16 *xxxxxxxxxxxx*1*xxxxxxxxxxxxxxxxx* Use Watchpoint #17 *xxxxxxxxxxx*1*xxxxxxxxxxxxxxxxxx* Use Watchpoint #18 *xxxxxxxxxx*1*xxxxxxxxxxxxxxxxxxx* Use Watchpoint #19 *xxxxxxxxx*1*xxxxxxxxxxxxxxxxxxxx* Use Watchpoint #20 *xxxxxxxx*1*xxxxxxxxxxxxxxxxxxxxx* Use Watchpoint #21 *xxxxxxx*1*xxxxxxxxxxxxxxxxxxxxxx* Use Watchpoint #22 *xxxxxx*1*xxxxxxxxxxxxxxxxxxxxxxx* Use Watchpoint #23 *xxxxx*1*xxxxxxxxxxxxxxxxxxxxxxxx* Use Watchpoint #24 *xxxx*1*xxxxxxxxxxxxxxxxxxxxxxxxx* Use Watchpoint #25 *xxx*1*xxxxxxxxxxxxxxxxxxxxxxxxxx* Use Watchpoint #26 *xx*1*xxxxxxxxxxxxxxxxxxxxxxxxxxx* Use Watchpoint #27 *x*1*xxxxxxxxxxxxxxxxxxxxxxxxxxxx* Use Watchpoint #28 1*xxxxxxxxxxxxxxxxxxxxxxxxxxxxx* Use Watchpoint #29

For extended Program Trace end trigger control, the PTETC register is used.

0	PTET

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Nexus Reg# - 0x36; Read/Write; Reset - 0x0

Figure 13-11. Program Trace End Trigger Control (PTETC) register

Table 13-18 details the PTETC register fields.

Table 13-18. PTETC field descriptions

Bits	Name	Description
31:30	—	Reserved for future functionality (read as 0)
29:0	PTET	Program Trace End Trigger Control 000000000000000000000000000000 Trigger disabled *xxxxxxxxxxxxxxxxxxxxxxxxxxxxx*1 Use Watchpoint #0 *xxxxxxxxxxxxxxxxxxxxxxxxxxxx*1*x* Use Watchpoint #1 *xxxxxxxxxxxxxxxxxxxxxxxxxxx*1*xx* Use Watchpoint #2 *xxxxxxxxxxxxxxxxxxxxxxxxxx*1*xxx* Use Watchpoint #3 *xxxxxxxxxxxxxxxxxxxxxxxxx*1*xxxx* Use Watchpoint #4 *xxxxxxxxxxxxxxxxxxxxxxxx*1*xxxxx* Use Watchpoint #5 *xxxxxxxxxxxxxxxxxxxxxxx*1*xxxxxx* Use Watchpoint #6 *xxxxxxxxxxxxxxxxxxxxxx*1*xxxxxxx* Use Watchpoint #7 *xxxxxxxxxxxxxxxxxxxxx*1*xxxxxxxx* Use Watchpoint #8 *xxxxxxxxxxxxxxxxxxxx*1*xxxxxxxxx* Use Watchpoint #9 *xxxxxxxxxxxxxxxxxxx*1*xxxxxxxxxx* Use Watchpoint #10 *xxxxxxxxxxxxxxxxxx*1*xxxxxxxxxxx* Use Watchpoint #11 *xxxxxxxxxxxxxxxxx*1*xxxxxxxxxxxx* Use Watchpoint #12 *xxxxxxxxxxxxxxxx*1*xxxxxxxxxxxxx* Use Watchpoint #13 *xxxxxxxxxxxxxxx*1*xxxxxxxxxxxxxx* Use Watchpoint #14 *xxxxxxxxxxxxxx*1*xxxxxxxxxxxxxxx* Use Watchpoint #15 *xxxxxxxxxxxxx*1*xxxxxxxxxxxxxxxx* Use Watchpoint #16 *xxxxxxxxxxxx*1*xxxxxxxxxxxxxxxxx* Use Watchpoint #17 *xxxxxxxxxxx*1*xxxxxxxxxxxxxxxxxx* Use Watchpoint #18 *xxxxxxxxxx*1*xxxxxxxxxxxxxxxxxxx* Use Watchpoint #19 *xxxxxxxxx*1*xxxxxxxxxxxxxxxxxxxx* Use Watchpoint #20 *xxxxxxxx*1*xxxxxxxxxxxxxxxxxxxxx* Use Watchpoint #21 *xxxxxxx*1*xxxxxxxxxxxxxxxxxxxxxx* Use Watchpoint #22 *xxxxxx*1*xxxxxxxxxxxxxxxxxxxxxxx* Use Watchpoint #23 *xxxxx*1*xxxxxxxxxxxxxxxxxxxxxxxx* Use Watchpoint #24 *xxxx*1*xxxxxxxxxxxxxxxxxxxxxxxxx* Use Watchpoint #25 *xxx*1*xxxxxxxxxxxxxxxxxxxxxxxxxx* Use Watchpoint #26 *xx*1*xxxxxxxxxxxxxxxxxxxxxxxxxxx* Use Watchpoint #27 *x*1*xxxxxxxxxxxxxxxxxxxxxxxxxxxx* Use Watchpoint #28 1*xxxxxxxxxxxxxxxxxxxxxxxxxxxxx* Use Watchpoint #29

For extended Data Trace start trigger control, the DTSTC register is used.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0		DTST																													

Nexus Reg# - 0x37; Read/Write; Reset - 0x0

Figure 13-12. Data Trace Start Trigger Control (DTSTC) register

Table 13-19 details the DTSTC register fields.

Table 13-19. DTSTC field descriptions

Bits	Name	Description
31:30	—	Reserved for future functionality (read as 0)
29:0	DTST	Data Trace Start Trigger Control 000000000000000000000000000000 Trigger disabled xxxxxxxxxxxxxxxxxxxxxxxxxxxxx1 Use Watchpoint #0 xxxxxxxxxxxxxxxxxxxxxxxxxxxx1x Use Watchpoint #1 xxxxxxxxxxxxxxxxxxxxxxxxxxx1xx Use Watchpoint #2 xxxxxxxxxxxxxxxxxxxxxxxxxx1xxx Use Watchpoint #3 xxxxxxxxxxxxxxxxxxxxxxxxx1xxxx Use Watchpoint #4 xxxxxxxxxxxxxxxxxxxxxxxx1xxxxx Use Watchpoint #5 xxxxxxxxxxxxxxxxxxxxxxx1xxxxxx Use Watchpoint #6 xxxxxxxxxxxxxxxxxxxxxx1xxxxxxx Use Watchpoint #7 xxxxxxxxxxxxxxxxxxxxx1xxxxxxxx Use Watchpoint #8 xxxxxxxxxxxxxxxxxxxx1xxxxxxxxx Use Watchpoint #9 xxxxxxxxxxxxxxxxxxx1xxxxxxxxxx Use Watchpoint #10 xxxxxxxxxxxxxxxxxx1xxxxxxxxxxx Use Watchpoint #11 xxxxxxxxxxxxxxxxx1xxxxxxxxxxxx Use Watchpoint #12 xxxxxxxxxxxxxxxx1xxxxxxxxxxxxx Use Watchpoint #13 xxxxxxxxxxxxxxx1xxxxxxxxxxxxxx Use Watchpoint #14 xxxxxxxxxxxxxx1xxxxxxxxxxxxxxx Use Watchpoint #15 xxxxxxxxxxxxx1xxxxxxxxxxxxxxxx Use Watchpoint #16 xxxxxxxxxxxx1xxxxxxxxxxxxxxxxx Use Watchpoint #17 xxxxxxxxxxx1xxxxxxxxxxxxxxxxxx Use Watchpoint #18 xxxxxxxxxx1xxxxxxxxxxxxxxxxxxx Use Watchpoint #19 xxxxxxxxx1xxxxxxxxxxxxxxxxxxxx Use Watchpoint #20 xxxxxxxx1xxxxxxxxxxxxxxxxxxxxx Use Watchpoint #21 xxxxxxx1xxxxxxxxxxxxxxxxxxxxxx Use Watchpoint #22 xxxxxx1xxxxxxxxxxxxxxxxxxxxxxx Use Watchpoint #23 xxxxx1xxxxxxxxxxxxxxxxxxxxxxxx Use Watchpoint #24 xxxx1xxxxxxxxxxxxxxxxxxxxxxxxx Use Watchpoint #25 xxx1xxxxxxxxxxxxxxxxxxxxxxxxxx Use Watchpoint #26 xx1xxxxxxxxxxxxxxxxxxxxxxxxxxx Use Watchpoint #27 x1xxxxxxxxxxxxxxxxxxxxxxxxxxxx Use Watchpoint #28 1xxxxxxxxxxxxxxxxxxxxxxxxxxxxx Use Watchpoint #29

For extended Data Trace end trigger control, the DTETC register is used.

31:30	29:0
0	DTET

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Nexus Reg# - 038; Read/Write; Reset - 0x0

Figure 13-13. Data Trace End Trigger Control (DTETC) Register

Table 13-20 details the DTETC register fields.

Table 13-20. DTETC field descriptions

Bits	Name	Description
31:30	—	Reserved for future functionality (read as 0)
29:0	DTET	Data Trace End Trigger Control 000000000000000000000000000000 Trigger disabled xxxxxxxxxxxxxxxxxxxxxxxxxxxxx1 Use Watchpoint #0 xxxxxxxxxxxxxxxxxxxxxxxxxxxx1x Use Watchpoint #1 xxxxxxxxxxxxxxxxxxxxxxxxxxx1xx Use Watchpoint #2 xxxxxxxxxxxxxxxxxxxxxxxxxx1xxx Use Watchpoint #3 xxxxxxxxxxxxxxxxxxxxxxxxx1xxxx Use Watchpoint #4 xxxxxxxxxxxxxxxxxxxxxxxx1xxxxx Use Watchpoint #5 xxxxxxxxxxxxxxxxxxxxxxx1xxxxxx Use Watchpoint #6 xxxxxxxxxxxxxxxxxxxxxx1xxxxxxx Use Watchpoint #7 xxxxxxxxxxxxxxxxxxxxx1xxxxxxxx Use Watchpoint #8 xxxxxxxxxxxxxxxxxxxx1xxxxxxxxx Use Watchpoint #9 xxxxxxxxxxxxxxxxxxx1xxxxxxxxxx Use Watchpoint #10 xxxxxxxxxxxxxxxxxx1xxxxxxxxxxx Use Watchpoint #11 xxxxxxxxxxxxxxxxx1xxxxxxxxxxxx Use Watchpoint #12 xxxxxxxxxxxxxxxx1xxxxxxxxxxxxx Use Watchpoint #13 xxxxxxxxxxxxxxx1xxxxxxxxxxxxxx Use Watchpoint #14 xxxxxxxxxxxxxx1xxxxxxxxxxxxxxx Use Watchpoint #15 xxxxxxxxxxxxx1xxxxxxxxxxxxxxxx Use Watchpoint #16 xxxxxxxxxxxx1xxxxxxxxxxxxxxxxx Use Watchpoint #17 xxxxxxxxxxx1xxxxxxxxxxxxxxxxxx Use Watchpoint #18 xxxxxxxxxx1xxxxxxxxxxxxxxxxxxx Use Watchpoint #19 xxxxxxxxx1xxxxxxxxxxxxxxxxxxxx Use Watchpoint #20 xxxxxxxx1xxxxxxxxxxxxxxxxxxxxx Use Watchpoint #21 xxxxxxx1xxxxxxxxxxxxxxxxxxxxxx Use Watchpoint #22 xxxxxx1xxxxxxxxxxxxxxxxxxxxxxx Use Watchpoint #23 xxxxx1xxxxxxxxxxxxxxxxxxxxxxxx Use Watchpoint #24 xxxx1xxxxxxxxxxxxxxxxxxxxxxxxx Use Watchpoint #25 xxx1xxxxxxxxxxxxxxxxxxxxxxxxxx Use Watchpoint #26 xx1xxxxxxxxxxxxxxxxxxxxxxxxxxx Use Watchpoint #27 x1xxxxxxxxxxxxxxxxxxxxxxxxxxxx Use Watchpoint #28 1xxxxxxxxxxxxxxxxxxxxxxxxxxxxx Use Watchpoint #29

13.4.8 Nexus Watchpoint Mask register (WMSK)

The Nexus Watchpoint Mask register (WMSK) controls which watchpoint events are enabled to produce Watchpoint Trace Messages (DC1$_{TM}$ must also be programmed to generate Watchpoint Trace Messages).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0		WEM																													

Nexus Reg# - 0x33; Read/Write; Reset - 0x0

Figure 13-14. Watchpoint Mask register (WMSK)

Table 13-21 details the Watchpoint Trigger register fields.

Table 13-21. WMSK field descriptions

Bits	Name	Description
31:30	—	Reserved for future functionality (read as 0)
29:0	WEM	Watchpoint Enable for Messaging 000000000000000000000000000000 No Watchpoints enabled for Watchpoint Trace Messaging *xxxxxxxxxxxxxxxxxxxxxxxxxxxxx*1 Watchpoint #0 enabled for WTM *xxxxxxxxxxxxxxxxxxxxxxxxxxxx*1*x* Watchpoint #1 enabled for WTM *xxxxxxxxxxxxxxxxxxxxxxxxxxx*1*xx* Watchpoint #2 enabled for WTM *xxxxxxxxxxxxxxxxxxxxxxxxxx*1*xxx* Watchpoint #3 enabled for WTM *xxxxxxxxxxxxxxxxxxxxxxxxx*1*xxxx* Watchpoint #4 enabled for WTM *xxxxxxxxxxxxxxxxxxxxxxxx*1*xxxxx* Watchpoint #5 enabled for WTM *xxxxxxxxxxxxxxxxxxxxxxx*1*xxxxxx* Watchpoint #6 enabled for WTM *xxxxxxxxxxxxxxxxxxxxxx*1*xxxxxxx* Watchpoint #7 enabled for WTM *xxxxxxxxxxxxxxxxxxxxx*1*xxxxxxxx* Watchpoint #8 enabled for WTM *xxxxxxxxxxxxxxxxxxxx*1*xxxxxxxxx* Watchpoint #9 enabled for WTM *xxxxxxxxxxxxxxxxxxx*1*xxxxxxxxxx* Watchpoint #10 enabled for WTM *xxxxxxxxxxxxxxxxxx*1*xxxxxxxxxxx* Watchpoint #11 enabled for WTM *xxxxxxxxxxxxxxxxx*1*xxxxxxxxxxxx* Watchpoint #12 enabled for WTM *xxxxxxxxxxxxxxxx*1*xxxxxxxxxxxxx* Watchpoint #13 enabled for WTM *xxxxxxxxxxxxxxx*1*xxxxxxxxxxxxxx* Watchpoint #14 enabled for WTM *xxxxxxxxxxxxxx*1*xxxxxxxxxxxxxxx* Watchpoint #15 enabled for WTM *xxxxxxxxxxxxx*1*xxxxxxxxxxxxxxxx* Watchpoint #16 enabled for WTM *xxxxxxxxxxxx*1*xxxxxxxxxxxxxxxxx* Watchpoint #17 enabled for WTM *xxxxxxxxxxx*1*xxxxxxxxxxxxxxxxxx* Watchpoint #18 enabled for WTM *xxxxxxxxxx*1*xxxxxxxxxxxxxxxxxxx* Watchpoint #19 enabled for WTM *xxxxxxxxx*1*xxxxxxxxxxxxxxxxxxxx* Watchpoint #20 enabled for WTM *xxxxxxxx*1*xxxxxxxxxxxxxxxxxxxxx* Watchpoint #21 enabled for WTM *xxxxxxx*1*xxxxxxxxxxxxxxxxxxxxxx* Watchpoint #22 enabled for WTM *xxxxxx*1*xxxxxxxxxxxxxxxxxxxxxxx* Watchpoint #23 enabled for WTM *xxxxx*1*xxxxxxxxxxxxxxxxxxxxxxxx* Watchpoint #24 enabled for WTM *xxxx*1*xxxxxxxxxxxxxxxxxxxxxxxxx* Watchpoint #25 enabled for WTM *xxx*1*xxxxxxxxxxxxxxxxxxxxxxxxxx* Watchpoint #26 enabled for WTM *xx*1*xxxxxxxxxxxxxxxxxxxxxxxxxxx* Watchpoint #27 enabled for WTM *x*1*xxxxxxxxxxxxxxxxxxxxxxxxxxxx* Watchpoint #28 enabled for WTM 1*xxxxxxxxxxxxxxxxxxxxxxxxxxxxx* Watchpoint #29 enabled for WTM

13.4.9 Nexus Overrun Control Register (OVCR)

The Nexus Overrun Control register controls Nexus behavior as the internal message queues fill up. Response options include suppressing selected message types, or stalling processor instruction execution.

0	SPTHOLD	0	SPEN	0	STTHOLD	0	STEN
31 30	29 28	27 26 25 24 23 22	21 20 19 18 17 16	15 14	13 12	11 10 9 8 7 6 5 4 3 2 1	0

Nexus Reg# - 0x32; Read/Write; Reset - 0x0

Figure 13-15. Nexus Overrun Control Register (OVCR)

Figure 13-16. OVCR field descriptions

Bits	Name	Description
31:30	—	Reserved, should be cleared
29:28	SPTHOLD	Suppression Threshold 00 Suppression threshold is when message queues are 1/4 full 01 Suppression threshold is when message queues are 1/2 full 10 Suppression threshold is when message queues are 3/4 full 11 Reserved
27:22	—	Reserved, should be cleared
21:16	SPEN	Suppression Enable 000000 Suppression is disabled *xxxxx*1 Ownership Trace message suppression is enabled *xxxx*1*x* Data Trace message suppression is enabled *xxx*1*xx* Program Trace message suppression is enabled *xx*1*xxx* Watchpoint Trace message suppression is enabled *x*1*xxxx* Reserved 1*xxxxx* Data Acquisition message suppression is enabled
15:14	—	Reserved, should be cleared
13:12	STTHOLD	Stall Threshold 00 Stall threshold is when message queues are 1/4 full 01 Stall threshold is when message queues are 1/2 full 10 Stall threshold is when message queues are 3/4 full 11 Reserved
11:1	—	Reserved, should be cleared
0	STEN	Stall Enable 0 Stalling is disabled 1 Stalling is enabled

13.4.10 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 four Data Trace channels controlled by the DTC for the Nexus 3 module. Channels can be programmed to trace data accesses or instruction accesses, but not independently.

RWT1		RWT2		RWT3		RWT4		0																RC1	RC2	RC3	RC4	DI	0		
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Nexus Reg# - 0xD; Read/Write; Reset - 0x0

Figure 13-17. Data Trace Control Register (DTC)

Table 13-22 details the Data Trace Control register fields.

Table 13-22. DTC field descriptions

Bits	Name	Description
31:30	RWT1	Read/Write Trace 1 00 No trace enabled *x*1 Enable Data Read Trace 1*x* Enable Data Write Trace
29:28	RWT2	Read/Write Trace 2 00 No trace enabled *x*1 Enable Data Read Trace 1*x* - Enable Data Write Trace
27:26	RWT3	Read/Write Trace 3 00 No trace enabled *x*1 Enable Data Read Trace 1*x* Enable Data Write Trace
25:24	RWT4	Read/Write Trace 4 00 No trace enabled *x*1 Enable Data Read Trace 1*x* Enable Data Write Trace
23:8	—	Reserved for future functionality (read as 0)
7	RC1	Range Control 1 0 Condition trace on address within range 1 Condition trace on address outside of range
6	RC2	Range Control 2 0 Condition trace on address within range 1 Condition trace on address outside of range
5	RC3	Range Control 3 0 Condition trace on address within range 1 Condition trace on address outside of range
4	RC4	Range Control 4 0 Condition trace on address within range 1 Condition trace on address outside of range
3	DI	Data Access / Instruction Access Trace 0 Condition trace on data accesses 1 Condition trace on instruction accesses
2:0	—	Reserved for future functionality (read as 0)

13.4.11 Data Trace Start Address Registers (DTSA1–4)

The Data Trace Start Address Registers define the start addresses for each trace channel.

Data Trace Start Address																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Nexus Reg# - 0xE; Read/Write; Reset - 0x0

Figure 13-18. Data Trace Start Address 1 (DTSA1) register

Data Trace Start Address																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Nexus Reg# - 0xF; Read/Write; Reset - 0x0

Figure 13-19. Data Trace Start Address 2 (DTSA2) register

Data Trace Start Address																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Nexus Reg# - 0x10; Read/Write; Reset - 0x0

Figure 13-20. Data Trace Start Address 3 (DTSA3) register

Data Trace Start Address																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Nexus Reg# - 0x11; Read/Write; Reset - 0x0

Figure 13-21. Data Trace Start Address 4 (DTSA4) register

13.4.12 Data Trace End Address registers (DTEA1–4)

The Data Trace End Address Registers define the end addresses for each trace channel.

Data Trace End Address																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Nexus Reg# - 0x12; Read/Write; Reset - 0x0

Figure 13-22. Data Trace End Address 1 (DTEA1) register

Data Trace End Address

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Nexus Reg# - 0x13; Read/Write; Reset - 0x0

Figure 13-23. Data Trace End Address 2 (DTEA2) register

Data Trace End Address

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Nexus Reg# - 0x14; Read/Write; Reset - 0x0

Figure 13-24. Data Trace End Address 3 (DTEA3) register

Data Trace End Address

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Nexus Reg# - 0x15; Read/Write; Reset - 0x0

Figure 13-25. Data Trace End Address 4 (DTEA4) register

Table 13-23 illustrates the range that will be selected for Data Trace for various cases of DTSA being less than, greater than, or equal to DTEA.

Table 13-23. Data trace address range options

Programmed values	Range control bit value	Range selected
DTSA < DTEA	0	DTSA → ← DTEA
DTSA < DTEA	1	← DTSA DTEA →
DTSA > DTEA	N/A	Invalid range — no trace
DTSA = DTEA	N/A	Invalid range— no trace

NOTE

DTSA must be less than DTEA in order to guarantee correct Data Write/Read Traces. Data Trace ranges are *inclusive* of the DTSA and DTEA addresses for Range Control settings indicating “within range”, and are *exclusive* of the DTSA and DTEA addresses for Range Control settings indicating “outside of range”.

Accesses that meet the range and access type qualifiers will cause assertion of a watchpoint output for Ranges 1, 2, and 3. There are three dedicated watchpoint outputs, one for each DTSA/DTEA sets 1, 2, and 3. Range 4 does not provide a watchpoint output. Note that when DTC_{DI}=1, all instruction fetches and prefetches (including discarded prefetches) are monitored, and thus theses range watchpoints differ from the IACx watchpoint outputs, which are not asserted for instructions that are not executed (i.e. when the instruction prefetch is discarded).

13.4.13 Read/Write Access Control/Status register (RWCS)

The Read Write Access Control/Status register (RWCS) provides control for Read/Write Access. Read/Write access provides DMA-like access to memory-mapped resources on the AHB System bus either while the processor is halted, or during runtime. Control is provided over access type, size, count, and certain bus attributes. RWCS also provides Read/Write Access Status information per Table 13-25.

AC	RW	SZ			MAP			PR		ATTR				0		CNT														ERR	DV
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Nexus Reg# - 0x7; Read/Write[1]; Reset - 0x0

Figure 13-26. Read/Write Access Control/Status register (RWCS)

[1] ERR and DV are read-only

Table 13-24. RWCS field descriptions

Bits	Name	Description
RWCS[31]	AC	Access Control 0 End access 1 Start access
RWCS[30]	RW	Read/Write Select 0 Read access 1 Write access
RWCS[29:27]	SZ	Word Size 000 8-bit (byte) 001 16-bit (half-word) 010 32-bit (word) 011 64-bit (doubleword, requires two passes through RWD) 100-111 Reserved (default to word)
RWCS[26:24]	MAP	MAP Select 000 Primary memory map 001-111 Reserved
RWCS[23:22]	PR[1]	Read/Write Access Priority 00 Reserved (default to highest priority) 01 Reserved (default to highest priority) 10 Reserved (default to highest priority) 11 Highest access priority

Table 13-24. RWCS field descriptions (continued)

Bits	Name	Description
RWCS[21:18]	ATTR	Access Attributes 0*xxx* **p_d_gbl** driven to 0 for accesses 1*xxx* **p_d_gbl** driven to 1 for accesses *x*0*xx* **p_d_hprot[4]** driven to 0 for accesses *x*1*xx* **p_d_hprot[4]** driven to 1 for accesses *xx*0*x* **p_d_hprot[3]** driven to 0 for accesses *xx*1*x* **p_d_hprot[3]** driven to 1 for accesses *xxx*0 **p_d_hprot[2]** driven to 0 for accesses *xxx*1 **p_d_hprot[2]** driven to 1 for accesses
RWCS[17:16]	—	Reserved for future functionality
RWCS[15:2]	CNT	Access Control Count hhhh - Number of accesses of word size SZ
RWCS[1]	ERR[2]	Read/Write Access Error (see Table 13-25)
RWCS[0]	DV[2]	Read/Write Access Data Valid (see Table 13-25)

[1] The priority functionality is not currently implemented

[2] ERR and DV are read-only

Table 13-25. Read/write access status bit encoding

Read action	Write action	ERR	DV
Read access has not completed	Write access completed without error	0	0
Read access error has occurred	Write access error has occurred	1	0
Read access completed without error	Write access has not completed	0	1
Not allowed	Not allowed	1	1

13.4.14 Read/Write Access Data (RWD)

The Read/Write Access Data Register (RWD) provides the data to/from system bus memory-mapped locations when initiating a read or a write access.

Read/Write Data
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Nexus Reg# - 0xA; Read/Write; Reset - 0x0

Figure 13-27. Read/Write Access Data register (RWD)

Read/Write accesses to the AHB require that the debug firmware properly retrieve/place the data in the RWD. Table 13-26 shows the proper placement of data into the RWD. Note that doubleword transfers require two passes through RWD.

Table 13-26. RWD data placement for transfers

Transfer Size and byte offset	RWA(2:0)	RWCS[SZ]	RWD			
			31:24	23:16	15:8	7:0
Byte	x x x	0 0 0	—	—	—	X
Half	x x 0	0 0 1	—	—	X	X
Word	x 0 0	0 1 0	X	X	X	X
Doubleword	0 0 0	0 1 1				
First RWD pass (low order data)			X	X	X	X
Second RWD pass (high order data)			X	X	X	X

Table Notes:
"X" indicates byte lanes with valid data
"—" indicates byte lanes that contain unused data.

Table 13-27 shows the mapping of RWD bytes to byte lanes of the AHB read and write data buses.

Table 13-27. RWD byte lane mapping

Transfer Size and byte offset	RWA(2:0)	RWD			
		31:24	23:16	15:8	7:0
Byte @000	0 0 0	—	—	—	AHB[7:0]
Byte @001	0 0 1	—	—	—	AHB[15:8]
Byte @010	0 1 0	—	—	—	AHB[23:16]
Byte @011	0 1 1	—	—	—	AHB[31:24]
Byte @100	1 0 0	—	—	—	AHB[39:32]
Byte @101	1 0 1	—	—	—	AHB[47:40]
Byte @110	1 1 0	—	—	—	AHB[55:48]
Byte @111	1 1 1	—	—	—	AHB[63:56]
Half @000	0 0 0	—	—	AHB[15:8]	AHB[7:0]
Half @010	0 1 0	—	—	AHB[31:24]	AHB[23:16]
Half @100	1 0 0	—	—	AHB[47:40]	AHB[39:32]
Half @110	1 1 0	—	—	AHB[63:56]	AHB[55:48]
Word @000	0 0 0	AHB[31:24]	AHB[23:16]	AHB[15:8]	AHB[7:0]
Word @100	1 0 0	AHB[63:56]	AHB[55:48]	AHB[47:40]	AHB[39:32]

Table 13-27. RWD byte lane mapping

Transfer Size and byte offset	RWA(2:0)	RWD			
		31:24	23:16	15:8	7:0
Doubleword @000	0 0 0				
first RWD pass		AHB[31:24]	AHB[23:16]	AHB[15:8]	AHB[7:0]
second RWD pass		AHB[63:56]	AHB[55:48]	AHB[47:40]	AHB[39:32]

Table Notes:
"—" indicates byte lanes that contain unused data.

13.4.15 Read/Write Access Address register (RWA)

The Read/Write Access Address register (RWA) provides the system bus address to be accessed when initiating a read or a write access.

Read/Write Address
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Nexus Reg# - 0x9; Read/Write; Reset - 0x0

Figure 13-28. Read/Write Access Address register (RWA)

13.5 Nexus 3 register access via JTAG/OnCE

Access to Nexus 3 register resources is enabled by loading a single instruction ("*NEXUS3-ACCESS*") into the JTAG Instruction Register (IR) (OnCE OCMD register). For the Nexus 3 block, the OCMD value is 0b0001111100.

Once the "*NEXUS3-ACCESS*" instruction has been loaded, the JTAG/OnCE port allows tool/target communications with all Nexus 3 registers according to the register map in Table 13-8.

Reading/writing of a Nexus 3 register then requires two (2) passes through the Data-Scan (DR) path of the JTAG state machine (see Section 13.21, IEEE 1149.1 (JTAG) RD/WR sequences).

1. The first pass through the DR selects the Nexus 3 register to be accessed by providing an index (see Table 13-8), and the direction (read/write). This is achieved by loading an 8-bit value into the JTAG Data Register (DR). This register has the following format:

(7bits)	(1 bit)
Nexus Register Index	R/W

RESET Value: 0x00

Figure 13-29. Data Register (DR) format

Table 13-28. Nexus register index values

Nexus register index	Selected from values in Table 13-8
Read/Write (R/W):	0 Read 1 Write

2. The second pass through the DR then shifts the data in or out of the JTAG port, LSB first.
   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.

13.6 Nexus message fields

Nexus messages are comprised of fields. Each field contains a distinct piece of information within a message, and each message contains multiple fields. Messages are transferred in packets over the Auxiliary Output protocol. A packet is a collection of fields. A packet may contain any number of fixed length fields, but may contain at most one variable length field. The variable length field must be the last field in a packet. The following sub-sections describe a subset of the message field types.

13.6.1 TCODE field

The TCODE field is a 6-bit fixed length field that identifies the type of message and its format. The field encodings are assigned by IEEE-ISTO 5001.

13.6.2 Source ID field (SRC)

Each Nexus module in a device is identified by a unique Client Source Identification Number. The number assigned to each Nexus module is determined by the SoC integrator, and is provided on the **nex3_ext_src_id[0:3]** input signals. Multi-threaded processors may assign additional source ID information to indicate which thread a message is associated with. The e200z759n3 Nexus 3 module implements a 4-bit fixed length Source ID field consisting of a Client Source ID.

13.6.3 Relative address field (U-ADDR)

The non-sync forms of the Program and Data Trace messages include addresses that are relative to the address that was transmitted in the previous Program or Data Trace message respectively. The relative address format is compliant with IEEE-ISTO 5001 and is designed to reduce the number of bits transmitted for address fields.

The relative address is generated by XORing the new address with the previous, and then using only the results up to the most significant ‘1’. To recreate the original address, the relative address is XORed with the previously decoded address.

The relative address of a Program Trace message is calculated with respect to the previous Program Trace message, regardless of any address information that may have been sent in any other trace messages in the interim between the two Program Trace messages.

The relative address of a Data Trace message is calculated with respect to the previous Data Trace message, regardless of any address information that may have been sent in any other trace messages in the interim between the two Data Trace messages.

Previous Address (A1) =0x0003FC01, New Address (A2) = 0x0003F365

```
Message Generation:

A1 = 0000 0000 0000 0011 1111 1100 0000 0001
A2 = 0000 0000 0000 0011 1111 0011 0110 0101

A1 Ý A2 = 0000 0000 0000 0000 0000 1111 0110 0100

Address Message (M1) = 1111 0110 0100


Address Re-creation:

A1 Ý M1 = A2
A1 = 0000 0000 0000 0011 1111 1100 0000 0001
M1 = 0000 0000 0000 0000 0000 1111 0110 0100
------------------------------------------------
A2 = 0000 0000 0000 0011 1111 0011 0110 0101
```

Figure 13-30. Relative address generation and re-creation

13.6.4 Full address field (F-ADDR)

Program Trace synchronization messages provide the full address associated with the trace event (leading zeroes may be truncated) with the intent of providing a reference point for development tools to operate from when reconstructing relative addresses. Synchronization messages are generated at significant mode switches and are also generated periodically to ensure that development tools are guaranteed to have a reference address given a sufficiently large sample of trace messages.

13.6.5 Address space indication field (MAP)

Data trace messages and indirect-type program trace messages provide the address space status (DS or IS value) in the address space (MAP) field. For Data Trace, the MAP field indicates the DS space (MSR_{DS} value) used for the data access. For Program Trace, the MAP field is used to indicate the future space used for instruction execution (new value of MSR_{IS}). A change in instruction address space will only occur on reset, on an exception, or via a **mtmsr**, **rfi**, **rfci**, **rfdi**, or **rfmci** instruction. A potential change in address space via an exception or via an **rfi**, **rfci**, **rfdi**, or **rfmci** instruction will cause a program trace indirect branch message to be generated indicating the new address space (IS) value, along with ICNT and HIST information for instructions executed up to the change (including the **rfi**, **rfci**, **rfdi**, or **rfmci**). A change in address space via a **mtmsr** instruction will cause a program correlation message to be generated indicating the new address space (IS) value, along with ICNT and HIST information for instructions executed prior to the change (including the **mtmsr**).

13.7 Nexus message queues

The Nexus 3 module implements internal message queues capable of storing up to three messages per cycle into a small initial queue, which then fills a larger queue at up to two messages per cycle. Messages that enter the queues are transmitted in the order in which they are received.

If more than three messages attempt to enter the queue in the same cycle, the highest priority messages are stored and the remaining message(s) will be dropped due to a collision. Collision events are expected to be rare.

The Overrun Control register (OVCR) controls the Nexus behavior as the message queue fills. The Nexus block may be programmed to:

- Allow the queue to overflow, drain the contents, queue an overrun error message and resume tracing.
- Stall the processor when the queue utilization reaches the selected threshold.
- Suppress selected message types when the queue utilization reaches the selected threshold.

13.7.1 Message queue overrun

In this mode, the message queue will stop accepting messages when an overrun condition is detected. The contents of the queues will be allowed to drain until empty. Incoming messages are discarded until the queue is emptied. Once empty, an overrun error message is enqueued that contains information about the types of messages that were discarded due to the overrun condition.

13.7.2 CPU stall

In this mode, processor instruction issue is stalled when the queue utilization reaches the selected threshold. The processor is stalled long enough drop one threshold level below the level that triggered the stall. For example, if stalling the processor is triggered at 1/4 full, the stall will stay in effect until the queue utilization drops to empty. There may be significant skid from the time that the stall request is made until the processor is able to stop completing instructions. This skid should be taken into consideration when programming the threshold. Refer to Section 13.4.9, Nexus Overrun Control Register (OVCR), for complete programming options.

13.7.3 Message suppression

In this mode, the message queue will disable selected message types when the queue utilization reaches the selected threshold. This allows lower bandwidth tracing to continue and possibly avoid an overrun condition. If an overrun condition occurs despite this message suppression, the queue will respond according to the behavior described in Section 13.7.1, Message queue overrun. Once triggered, message suppression will remain in effect until queue utilization drops to the threshold below the level selected to trigger suppression.

13.7.4 Nexus message priority

Nexus messages may be lost due to contention with other message types under the following circumstances:

- More than three messages are generated in the same cycle

Table 13-29 lists the various message types and their relative priority from highest to lowest.

Up to three message requests can be queued into the message buffer in a given cycle. If more than three message requests exist in a given cycle, the three highest priority message classes are queued into the message buffer. The remaining messages that did not successfully queue into the message buffer in that cycle will generate subsequent responses as detailed in Table 13-29.

The CPU is capable of completing two instructions per cycle. If multiple trace messages need to be queued at the same time, they will be queued with the following priority: Instruction 0 (oldest instruction) (WPM $\rightarrow$ DQM $\rightarrow$ PCM_{PIDMSG} $\rightarrow$ OTM $\rightarrow$ BTM $\rightarrow$ DTM) $\rightarrow$ Instruction1 (newer instruction) (WPM $\rightarrow$ DQM $\rightarrow$ OTM $\rightarrow$ BTM $\rightarrow$ DTM). Up to three messages may be simultaneously queued. Note that for the cycle following a dropped PTM, non-periodic OTM, or DQM message, only two other messages may be queued in addition to the dropped error message.

Watchpoint messages from instructions that complete at the same time or events that occur during the same cycle will be combined.

Table 13-29. Message type priority and message dropped responses

Message type	Message	Priority	Message dropped response
Error	Error	0 (highest)	N/A[1]
WP (Watchpoint Trace)	WPM (Watchpoint Message)	1	N/A[1]
DQ (Data Acquisition)	DQM (Data Acquisition Message)	2	DQM Error Message
Program Trace (PID MSG)	PCM — PID or mtmsr IS update (Program Correlation Message)	2	OTM Error Message
OT (Ownership)	OTM - PID update (Ownership Trace Message)	2	OTM Error Message[2]
Program Trace	BTM (Branch Trace Message)	2	BTM Error Message, Sync upgrade next BTM
	RFM (Resource Full for Instruction counter or history buffer)	3	BTM Error Message Sync upgrade next BTM
	DS (Debug Status Message)	4	Sync upgrade next BTM
	PCM (Program Correlation Message)	5	BTM Error Message Sync upgrade next BTM

Table 13-29. Message type priority and message dropped responses (continued)

Message type	Message	Priority	Message dropped response
DT (Data Trace)	DTM (Data Trace Message)	6	Sync upgrade next DTM
OT (Ownership)	OTM — Periodic update (Ownership Trace Message)	7 (lowest)	none

[1] Error and Watchpoint messages are not dropped due to collisions, due to their priority.

[2] Message will always be dropped if program trace is enabled, and program correlation messages for PID0 /mtmsr IS messages are not masked (Event Code = 0101). No error message is sent for this case since the PID value is contained in the higher priority message.

13.7.5 Data Acquisition Message (DQM) priority loss response

If a Data Acquisition Message (DQM) loses arbitration due to contention with higher priority messages, an error message will be generated to indicate that a DQM has been lost due to contention.

13.7.6 Ownership Trace Message (OTM) priority loss response

If an Ownership Trace message (OTM) due to software updates to the Process ID state loses arbitration due to contention with higher priority messages other than a program correlation message with EVCODE = 0101 (PID or MSR_{IS} update), an error message will be generated to indicate that a OTM has been lost due to contention. If the pending OTM is a periodic update, the event is dropped without generating an error message.

13.7.7 Program Trace Message (PTM) priority loss response

If a Program Trace message (PTM) loses arbitration due to contention with higher priority messages, and the discarded PTM is a Program Correlation message, a Resource Full message for instruction count or history buffer, or a Branch Trace message, then an Error message is generated to indicate that branch trace information has been lost, and the next Branch Trace message will be upgraded to a sync-type message.

If the discarded PTM is a Program Correlation message with PID information (EVCODE=0101), the Error message will indicate a dropped OTM and a dropped Program Trace (Error code = xxxx11xx).

13.7.8 Data Trace Message (DTM) priority loss response

If a Data Trace message (DTM) loses arbitration due to contention with higher priority messages, the DTM event is discarded, and the next DTM is upgraded to a sync-type message.

13.8 Debug Status messages

Debug Status messages report low power mode and debug status. Debug Status messages are enabled when Nexus 3 is enabled. Entering/exiting Debug mode as well as entering, exiting, or changing low power mode(s) will trigger a Debug Status message, indicating the value of the most significant byte in the Development Status register. Debug status information is sent out in the following format:

(8 bits)	(4 bits)	(6 bits)
DS[31:24]	Src. Proc.	TCODE (000000)

Fixed length = 18 bits

Figure 13-31. Debug Status message format

13.9 Error messages

Error messages are enabled whenever the debug logic is enabled. There are two conditions that will produce an error message, each receiving a separate error type designation:

- A message is discarded due to contention with other (higher priority) message types. These errors will have an Error Type value of 1.
- The message queue overruns. After the queue is drained, an error message is enqueued with an error code that indicates what types of messages were discarded during the interim. These errors will have an Error Type value of 0.

NOTE

The OVCR Register can be used in order to alleviate potential overrun situations.

Error information is messaged out in the following format (also see Table 13-3 and Table 13-4):

(6 bits)	(4 bits)	(4 bits)	(6 bits)
Error Code	Error Type	Src. Proc.	TCODE (001000)

Fixed length = 20 bits

Figure 13-32. Error message format

13.10 Ownership trace

This section details the ownership trace features of the Nexus 3 module.

13.10.1 Overview

Ownership trace provides a macroscopic view, such as task flow reconstruction, when debugging software written in a high level (or object-oriented) language. It offers the highest level of abstraction for tracking operating system software execution. This is especially useful when the developer is not interested in debugging at lower levels.

13.10.2 Ownership Trace Messaging (OTM)

Ownership trace information is messaged via the auxiliary port using an Ownership Trace Message (OTM). Zen processors contain a Power Architecture BookE defined “Process ID” register within the CPU. It is updated by the operating system software to provide task/process ID information. The contents of this register are replicated on the pins of the processor and connected to Nexus. The Process ID register value can be accessed using the **mfspr**/**mtspr** instructions.

NOTE

The CPU includes a Process ID register (PID0), thus the Nexus UBA functionality is not implemented.

There are two conditions that will cause an Ownership Trace Message when Ownership Trace is enabled:

- When new information is updated in the PID0 register by the Zen processor, the data is latched within Nexus, and is messaged out via the auxiliary port, allowing development tools to trace ownership flow. However, if Program Trace is enabled, and program correlation messages for PID0 /**mtmsr** IS messages are not masked (Event Code = 0101), then an OTM will not be generated for an update to the PID0 register, since the program correlation message will provide this PID0 update information.
- Periodically, at least once every 256 messages, the most recent state of the PID0 register is messaged out. The resulting Ownership Trace message will indicate in the PID Index sub-field that PID0 status is being reported and the most recent value of the PID0 register will be conveyed in the Process ID value sub-field. These periodic Ownership Trace message events can be disabled by setting $DC1_{POTD}$.

Ownership trace information is messaged out in the following format:

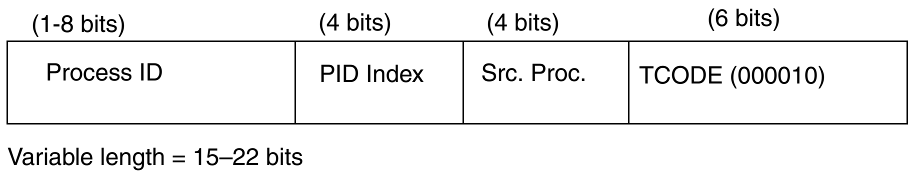


Figure 13-33. Ownership Trace Message (OTM) format

13.11 Program trace

This section details the program trace mechanism supported by Nexus3 for the e200z759n3 processor. Program trace is implemented via Branch Trace Messaging (BTM) as per the IEEE-ISTO 5001 standard definition. Branch Trace Messaging for Zen processors is accomplished by snooping the Zen virtual address bus (between the CPU and MMU), attribute signals, and CPU Status (**p_mode[0:3]**, **p_pstat_pipe{0,1}[0:5]**).

13.11.1 Branch Trace messaging types

Traditional Branch Trace messaging facilitates program trace by providing the following types of information:

- Messaging for taken direct branches includes how many sequential instructions were executed since the last taken branch or exception, including the taken direct branch. Branch instructions are included in the count of sequential instructions.
- Messaging for taken indirect branches and exceptions includes how many sequential instructions were executed since the last taken branch or exception and the unique portion of the branch target address or exception vector address. Branch instructions are included in the count of sequential instructions. For taken indirect branches that trigger generation of a message, the branch is also

included in the count. Messaging for taken indirect branches and exceptions also include the newly established value of the MSR_{IS} bit in the MAP field if the indirect branch message is due to an exception or **rfi**, **rfci**, **rfdi**, or **rfmci** class instruction. For all other indirect branches, the MAP field will reflect the current value of MSR_{IS}.

Branch History messaging facilitates program trace by providing the following information.

- Messaging for taken indirect branches and exceptions includes a) how many sequential instructions (I-CNT) were executed since the last predicate instruction, taken/not taken direct branch, taken/not-taken indirect branch, or exception, b) the unique portion of the branch target address or exception vector address, and c) a branch/predicate instruction history field. Each bit in the history field represents a direct branch or predicated instruction where a value of one (1) indicates taken, and a value of zero (0) indicates not taken. Certain instructions (**evsel**) generate a pair of predicate bits that are both reported as consecutive bits in the history field. Not-taken indirect branches will generate a history bit with a value of zero (0). Instructions that generate history bits are not included in instruction counts. For taken indirect branches that trigger generation of this message type, the branch is included in the count, but not in the history field. Messaging for taken indirect branches and exceptions also include the newly established value of the MSR_{IS} bit in the MAP field if the indirect branch message is due to an exception or **rfi**, **rfci**, **rfdi**, or **rfmci** class instruction. For all other indirect branches, the MAP field will reflect the current value of MSR_{IS}.

13.11.1.1 Zen Indirect Branch message instructions

Table 13-30 shows the types of instructions and events that cause Indirect Branch messages or Branch History messages to be encoded.

Table 13-30. Indirect Branch message sources

Source of Indirect Branch Message	Instructions / detail
Taken branch relative to a register value	**bcctr**, **bcctrl**, **bclr**, **bclrl**, **se_bctr**, **se_bctrl**, **se_blr**, **se_blrl**
System Call / Trap exceptions taken	**sc**, **se_sc**, **tw**, **twi**
Return from interrupts / exceptions	**rfi**, **rfci**, **rfdi**, **se_rfi**, **se_rfci**, **se_rfdi**
Exit from reset with Program Trace Enabled	Indirect branch with Sync, target address is initial instruction, count=1

13.11.1.2 Zen Direct Branch Message instructions

Table 13-31 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 13-31. Direct Branch message sources

Source of Direct Branch Message	Instructions
Taken direct branch instructions Instruction Synchronize	**b**, **ba**, **bl**, **bla**, **bc**, **bca**, **bcl**, **bcla**, **se_b**. **se_bc**, **se_bl**, **e_b**, **e_bc**, **e_bl**, **e_bcl**, **isync**, **se_isync**

13.11.1.3 BTM using Branch History Messages

Traditional BTM Messaging can accurately track the number of sequential instructions between branches, but cannot accurately indicate which instructions were conditionally executed, and which were not.

Branch History Messaging solves this problem by providing a predicated instruction history field in each Indirect Branch Message. Each bit in the history represents a predicated instruction or direct branch, or a not-taken indirect branch. A value of one (1) indicates the conditional instruction was executed or the direct branch was taken. A value of zero (0) indicates the conditional instruction was not executed or the branch was not taken. Certain instructions (**evsel**) generate a pair of predicate bits that 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.

13.11.1.4 BTM using Traditional Program Trace messages

Based on the PTM bit in the DC1 Register, Program Tracing can utilize either Branch History Messages (PTM=1) or traditional Direct/Indirect Branch Messages (PTM=0).

Branch History will save bandwidth and keep 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 will enter the FIFO in the order it occurred and will be messaged out maintaining that order.

13.11.2 BTM Message formats

The Nexus 3 block supports three types of traditional BTM messages — Direct, Indirect, and Synchronization messages. It supports two types of branch history BTM Messages — Indirect Branch History, and Indirect Branch History with Synchronization Messages.

13.11.2.1 Indirect Branch Messages (history)

Indirect branches include all taken branches whose destination is determined at run time, interrupts, and exceptions. If DC1$_{PTM}$ is set to '1', indirect branch information is messaged out in the following format:

(1-32 bits)	(1-32 bits)	(1-8 bits)	(1 bit)	(4 bits)	(6 bits)
Branch History	Relative Address	Sequence Count	Inst Space	Source Proc.	TCODE (011100)

Max length = 83 bits; Min length = 14 bits

Figure 13-34. Indirect Branch Message (History) format

13.11.2.2 Indirect Branch Messages (traditional)

If DC1$_{PTM}$ is cleared to ‘0’, indirect branch information is messaged out in the following format:

(1-32 bits)	(1-8 bits)	(1 bit)	(4 bits)	(6 bits)
Relative Address	Sequence Count	Inst Space	Source Proc.	TCODE (000100)

Max length = 51 bits; Min length = 13 bits

Figure 13-35. Indirect Branch Message format

13.11.2.3 Direct Branch Messages (traditional)

Direct branches (conditional or unconditional) are all taken branches whose destination is fixed in the instruction opcode. Direct branch information is messaged out in the following format:

(1-8 bits)	(4 bits)	(6 bits)
Sequence Count	Src. Proc.	TCODE (000011)

Max length = 18 bits; Min length = 11bits

Figure 13-36. Direct Branch Message format

NOTE

When DC1$_{PTM}$ is set, Direct Branch Messages will not be transmitted. Instead, each direct branch, not-taken indirect branch, or predicated instruction will be recorded in the history buffer.

13.11.3 Program Trace message fields

The following subsections describe specific fields used for Program Trace messages.

13.11.3.1 Sequential Instruction Count field (ICNT)

Most of the program trace messages include an instruction count field. For traditional Branch Messages, ICNT represents the number of sequential instructions including non-taken branches since the last Direct/Indirect Branch Messages. Branch instructions that trigger message generation are included in the ICNT.

For Branch History messages, ICNT represents the number of instructions executed since the last taken/non-taken direct branch, predicate instruction, last taken/not-taken indirect branch, or exception. Branch instructions that trigger message generation are included in the ICNT. Instructions that generate history bits are not included in the ICNT.

The sequential instruction counter overflows after its value reaches 255 and is reset to 0. In addition, the next BTM message (corresponding to the 256th or later instruction) will be converted to a synchronization type message.

The instruction counter is reset every time the instruction count is transmitted in a message or whenever there is a branch/predicate history event, as well as on exiting from debug mode.

13.11.3.2 Branch/Predicate Instruction History (HIST)

If $DC1_{PTM}$ is set, BTM messaging will use the Branch History format. The branch history (HIST) field in these messages provides a history of branch execution used for reconstructing the program flow. The branch/predicate history buffer stores information about branch and predicate instruction execution. The buffer is implemented as a left-shifting register. The buffer is preloaded with a one (1), which acts as a stop bit (the most significant 1 in the history field is a termination bit for the field). The pre-loaded bit itself is not part of the history, but is transmitted with the packet.

A value of one (1) is shifted into the history buffer for each taken direct branch (program counter relative branch) or predicate instruction whose condition evaluates to true. A value of zero (0) is shifted into the history buffer for each not-taken branch (including indirect branch instructions) or predicate instruction whose condition evaluates to false. For the **evsel** instruction, two bits are shifted in, corresponding to the low element (shifted in first) and the high element (shifted in second) conditions.

This history buffer information is transmitted as part of an Indirect Branch with History message, as part of a Program Correlation message, or as part of a Resource Full message if the history buffer becomes full. The history buffer is reset every time the history information is transmitted in a message, as well as on exiting from debug mode.

Table 13-32. Branch/predicate history events

Branch/predicate history event	History bit(s)	Relevant instructions
Not taken register indirect branches	0	**bcctr, bcctrl, bclr, bclrl**
Not taken direct branches	0	**b, ba, bc, bca, bla, bcla, bl, bcl**
Taken direct branches	1	**b, ba, bc, bca, bla, bcla, bl, bcl**[1]
evsel instruction	00,01,10, or 11	**evsel**

[1] If the EVCODE for direct branch function calls is not masked in DC4, taken bl and bcl instructions will generate program correlation messages and will not be logged in the history buffer.

13.11.3.3 Execution mode indication

In order for a development tool to properly interpret instruction count and history information, it must be aware of the execution mode context of that information. VLE instructions will be interpreted differently from non-VLE instructions.

Program trace messages provide the execution mode status in the least significant bit of the **reconstructed** address field. A value of '0' indicates that preceding instruction count and history information should be interpreted in a non-VLE context. A value of '1' indicates that the preceding instruction count and history information should be interpreted in a VLE context. Note that when a branch results in an execution mode switch, the program trace message resulting from that branch will indicate the previous execution state. The new state will not be signaled until the next program trace message.

In some cases, a Program Correlation Message is generated to indicate execution mode status. Refer to Section 13.11.5, Program Correlation Messages (PCM), for more information on these cases.

13.11.4 Resource Full Messages

The Resource Full Message is used in conjunction with Branch Trace and Branch History Messages. The Resource Full Message is generated when either the internal branch/predicate history buffer is full, or if the BTM Instruction sequence counter (I-CNT) overflows. If synchronization is needed at the time this message is generated, the synchronization is delayed until the next Branch Trace Message that is not a Resource Full Message.

For history buffer overflow, the Resource Full Message transmits a Resource Code (RCODE) of 0b0001 and the current contents of the history buffer, including the stop bit, are transmitted in the Resource Data (RDATA) field. This history information can be concatenated by the development tool with the branch/predicate history information from subsequent messages to obtain the complete branch/predicate history between indirect changes of flow.

For instruction counter overflow, the Resource Full Message transmits an RCODE of 0b0000 and a value of 0xFF is transmitted in the RDATA field, indicating that 255 sequential instructions have been executed since the last change of flow or, if program trace is in history mode, since the last instruction that recorded history information.

(1-32 bits)	(4 bits)	(4 bits)	(6 bits)
RDATA	RCODE	Src. Proc.	TCODE (011011)

Max length = 46 bits; Min length = 15 bits

Figure 13-37. Resource Full Message format

Table 13-33 shows the RCODE encodings and RDATA information used for Resource Full messages.

Table 13-33. RCODE encoding

RCODE	Description	RDATA field
0000	Program Trace Instruction counter reached 255 and was reset.	0xFF
0001	Program Trace, Branch / Predicate Instruction History full.	Branch HIstory. This type of packet is terminated by a stop bit set to 1 after the last history bit.

13.11.5 Program Correlation Messages (PCM)

Program Correlation Messages (PCMs) are used to correlate events to the program flow that may or may not be associated with the instruction stream. The following events will result in a PCM when program trace is enabled:

- When the CPU enters debug mode, a PCM is generated. The instruction count and history information provided by the PCM can be used to determine the last sequence of instructions executed prior to debug mode entry.
- When the CPU first enters a low power mode in which instructions are no longer executed, a PCM is generated. The instruction count and history information provided by the PCM can be used to determine the last sequence of instructions executed prior to low power mode entry.

- Whenever program trace is disabled by any means, a PCM is generated. The instruction count and history information provided by the PCM can be used to determine the last sequence of instructions executed prior to disabling program trace. A second PCM is generated on this event if there has been an execution mode switch into or out of a sequence of VLE instructions. This VLE state information allows the development tool to interpret any preceding instruction count or history information in the proper context.
- When a “Branch and Link” instruction executes (direct branch function call - **bl/bcl/bla/bcla**-type instructions)
- Whenever the CPU crosses a page boundary that results in an execution mode switch into or out of a sequence of VLE instructions, a PCM is generated. The PCM effectively breaks up any running instruction count and history information between the two modes of operation so that the instruction count and history information can be processed by the development tool in the proper context.
- When using program trace in history mode, when a direct branch results in an execution mode switch into or out of a sequence of VLE instructions, a PCM is generated. The PCM effectively breaks up any running history information between the two modes of operation so that the history information can be processed by the development tool in the proper context.
- When program trace becomes masked due to MSR_{PMM}=‘0’ and $DC4_{PTMARK}$=‘1’.
- When a new address translation is established in the TLB via a **tlbwe** instruction.
- When address translation(s) are invalidated in the TLB via a **tlbivax** instruction.
- When a new instruction address space setting (IS) is established in the MSR via a **mtmsr** instruction.
- When an update to the process ID register (PID0) is made via a **mtspr** PID0.

Refer to Table 13-6 for the event codes that are supported in this implementation. Event code masking is available via the EVCDM field of the DC4 register to allow for control over generation of Program Correlation messages for each event type.

Program Correlation is messaged out in the following formats:

(1-32 bits)	(1-8 bits)	(2 bits)	(4 bits)	(4 bits)	(6 bits)
Branch History	Sequence Count	CDF*	EVCODE	Src. Proc.	TCODE (100001)

Max length = 56 bits; Min length = 18 bits

* - CDF=01,
EVCODE = Any but 0101, 1100

Figure 13-38. Program Correlation Message formats (1 of 4)

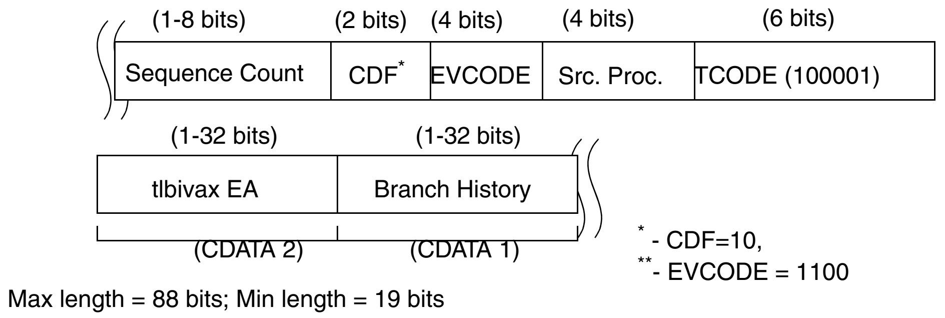


Figure 13-39. Program Correlation Message formats (2 of 4)

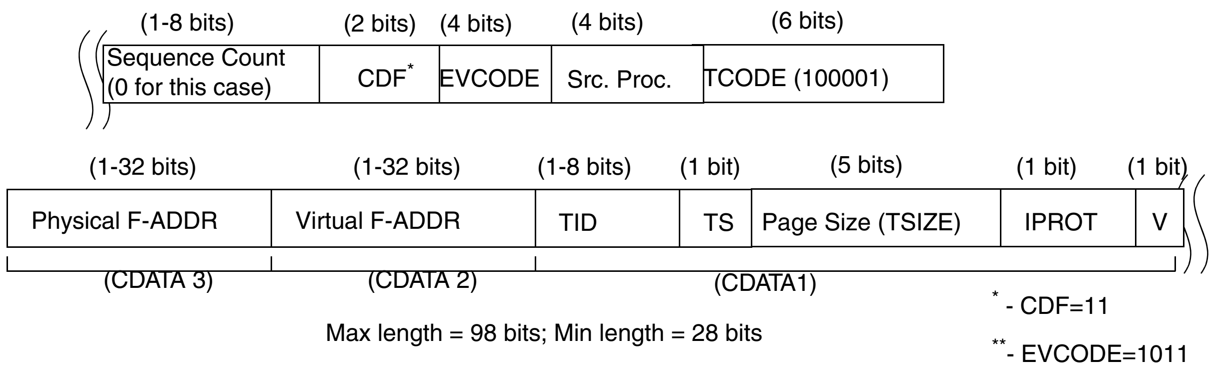


Figure 13-40. Program Correlation Message formats (3 of 4)

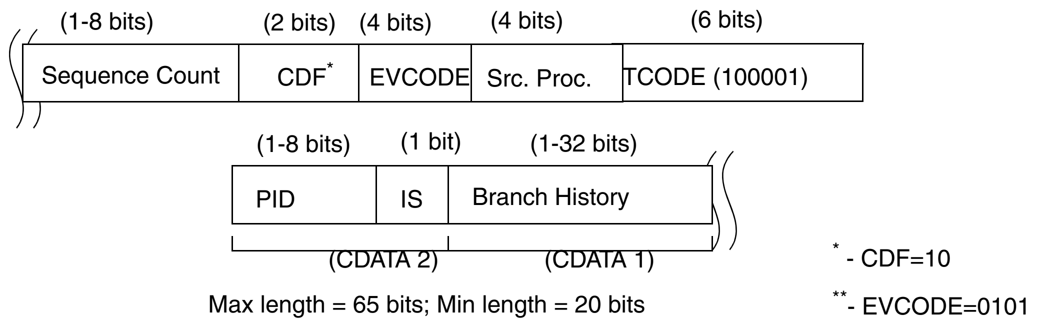


Figure 13-41. Program Correlation Message formats (4 of 4)

13.11.5.1 Program Correlation Message generation for TLB update with new address translation

When a new address translation is established in the TLB, a Program Correlation message is generated containing the information regarding the new TLB entry using EVCODE = 1011. A PCM with current history and instruction count will also be generated using EVCODE = 1011 (unless collapsed with a different EVCODE) and sent just prior to sending the PCM containing the newly established address translation. The messages are provided so that the address translation information can be processed by the development tool in the proper program flow.

13.11.5.2 Program Correlation Message generation for TLB invalidate (tlbivax) operations

When a **tlbivax** is executed to invalidate one or more entries in the TLB, a Program Correlation message is generated containing the information regarding the **tlbivax** EA used for invalidation using EVCODE = 1100. The current history and instruction count (which includes the **tlbivax** instruction) is also included in the message. The messages are provided so that the address translation information can be processed by the development tool in the proper program flow.

13.11.5.3 Program Correlation Message generation for PID updates or MSR_{IS} updates

When a (potentially) new value is established in the PID via a **mtspr** PID0, a Program Correlation message is generated containing the information regarding the new PID0 value. This PCM also contains the current history and instruction count, and the current value of MSR_{IS}. The message is provided so that address translation information can be processed by the development tool in the proper program flow. The **mtspr** PID0 is included in the instruction count information. Note that Ownership Trace Messages (other than the periodic OTM) are redundant with the information provided, and may be disabled to avoid unnecessary message bandwidth or collisions.

When a new value is established in MSR_{IS} via a **mtmsr** instruction, a Program Correlation message is generated containing the information regarding the new MSR_{IS} value. This PCM also contains the current history and instruction count, and the current value of PID0. The message is provided so that address translation information can be processed by the development tool in the proper program flow. The **mtmsr** instruction is included in the instruction count information.

13.11.6 Program trace overflow error messages

An error message occurs when a new message cannot be queued due to the message queue being full. The FIFO will discard incoming messages until it has completely emptied the queue. Once emptied, an error message will be queued. The error encoding will indicate which type(s) of messages attempted to be queued while the FIFO was being emptied.

13.11.7 Program trace synchronization messages

By default, program trace messages will perform XOR compression on the branch target address to produce the address field for the message. This compression is consistent with the specification in IEEE-ISTO 5001.

Under some conditions an uncompressed address is sent to provide development tools with a baseline reference address. A Program Trace Direct/Indirect Branch with Sync Message is messaged via the auxiliary port (provided Program Trace is enabled) for the following conditions (see Table 13-34):

- Initial Program Trace Message upon the first direct/indirect branch after exit from system reset or whenever program trace is enabled.
- Upon direct/indirect branch after returning from a CPU Low Power state.
- Upon direct/indirect branch after returning from Debug mode.

- Upon direct/indirect branch after occurrence of queue overrun (can be caused by any trace message), provided Program Trace is enabled.
- Upon direct/indirect branch after the periodic program trace counter has expired indicating 255 *without-sync* Program Trace Messages have occurred since the last *with-sync* message occurred.
- Upon direct/indirect branch after assertion of the Event In (**nex_evti_b**) pin if the EIC bits within the DC1 Register have enabled this feature.
- Upon direct/indirect branch after the sequential instruction counter has expired indicating 255 instructions have occurred since the last change of flow.
- Upon direct/indirect branch after a BTM Message was lost due to a collision while attempting to enter the message queue.
- Upon the first direct/indirect branch message after an execution mode switch into or out of a sequence of VLE instructions.
- When program trace becomes unmasked due to $MSR_{PMM} \rightarrow$ '1' with $DC4_{PTMARK}$='1'.

Note that the ICNT and History information for the first message will not be meaningful for some of these cases, since the temporary masking of program trace may result in ambiguous values. Subsequent w/sync messages will not have this issue.

The format for Program Trace Direct/Indirect Branch with Sync Messages is as follows:

(1-32 bits)	(1-8 bits)	(1 bit)	(4 bits)	(6 bits)
Full Target Address	Sequence Count	Inst Space	Source Proc.	TCODE (001011 or 001100)

Max length = 51 bits; Min length = 13 bits

Figure 13-42. Direct/indirect branch with sync message format

The format for Program Trace Indirect Branch History with Sync. Messages is as follows:

(1-32 bits)	(1-32 bits)	(1-8 bits)	(1 bit)	(4 bits)	(6 bits)
Branch History	Full Target Address	Sequence Count	Inst Space	Source Proc.	TCODE (011101)

Max length = 83 bits; Min length = 14 bits

Figure 13-43. Indirect branch history w/ Sync. Message Format

Exception conditions that result in Program Trace Synchronization are summarized in Table 13-34.

Table 13-34. Program Trace exception summary

Exception condition	Exception Handling
System Reset Negation	At the negation of JTAG reset (**j_trst_b**), queue pointers, counters, state machines, and registers within the Nexus 3 module are reset. Upon exiting system reset, if Program Trace is already enabled), a Program Trace Message is sent as an Indirect Branch w/ Sync. Message.
Program Trace Enabled	The first Program Trace Message (after Program Trace has been enabled) is a synchronization message.

Table 13-34. Program Trace exception summary (continued)

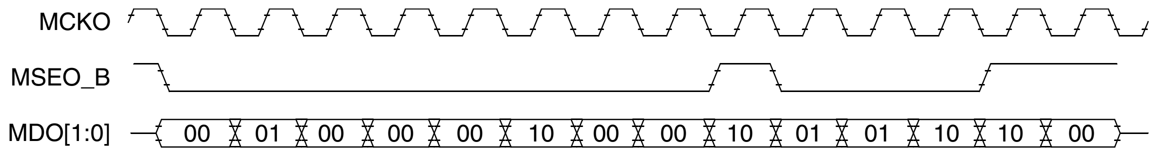
Exception condition	Exception Handling
Exit from Low Power/Debug	Upon exit from a Low Power mode or Debug mode the next direct/indirect branch will be converted to a Direct/Indirect Branch with Sync. Message.
Queue Overrun	An error message occurs when a new message cannot be queued due to the message queue being full. The FIFO will discard messages until it has completely emptied the queue. Once emptied, an error message will be queued. The error encoding will indicate which type(s) of messages attempted to be queued while the FIFO was being emptied. The next BTM message in the queue will be a Direct/Indirect Branch w/ Sync. Message.
Periodic Program Trace Sync.	A forced synchronization occurs periodically after 255 non-sync Program Trace Messages have been queued. A Direct/Indirect Branch w/ Sync. Message is queued. The periodic program trace message counter then resets.
Event In	If the Nexus module is enabled, a **nex_evti_b** assertion initiates a Direct/Indirect Branch w/ Sync. Message upon the next direct/indirect branch (if Program Trace is enabled and the EIC bits of the DC1 Register have enabled this feature).
Sequential Instruction Count Overflow	After the sequential instruction counter reaches its maximum count (up to 255 sequential instructions may be executed), a forced synchronization occurs. The sequential counter then resets. A Program Trace Direct/Indirect Branch w/ Sync.Message is queued upon execution of the next branch. A Resource Full Message is Queued on the overflow event. If a branch instruction is the 255th instruction to occur, and causes a Program Trace message to be queued, then no Resource Full Message is queued, and the w/Sync message will be queued for the *next* Program Trace Direct/Indirect Branch Message.
Collision Priority	All Messages have the following priority: Instruction 0 (WPM $\rightarrow$ DQM $\rightarrow$ PCM$_{PIDMSG}$- > OTM $\rightarrow$ BTM $\rightarrow$ DTM) $\rightarrow$ Instruction1 (WPM $\rightarrow$ DQM $\rightarrow$ OTM $\rightarrow$ BTM $\rightarrow$ DTM), where instruction0 is the oldest instruction. A BTM Message from Instruction1 that attempts to enter the queue at the same time as three higher priority messages from either instruction will be lost. An Error Message will be sent indicating the BTM was lost. The following direct/indirect branch will queue a Direct/Indirect Branch w/ Sync. Message. The count value within this message will reflect the number of sequential instructions executed after the last successful BTM Message was generated. This count will include the branch that did not generate a message due to the collision.
Execution Mode Switch	Whenever the CPU switches execution mode into or out of a sequence of VLE instructions, the next branch trace message will be a Direct/Indirect Branch w/ Sync Message.

13.11.8 Enabling Program Trace

Program Trace Messaging can be enabled in one of two ways:

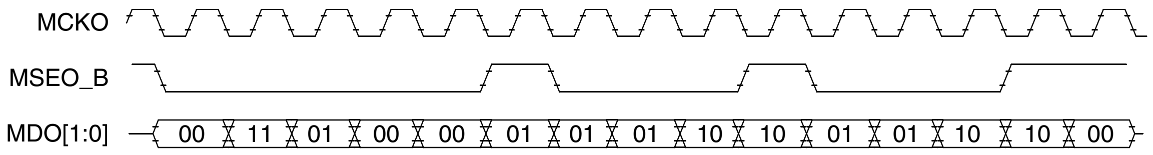
- Setting the TM field of the DC1 Register to enable Program Trace
- Using the PTS field of the WT Register to enable Program Trace on Watchpoint hits (Zen watchpoints are configured within the CPU)
- Filtering of Program Trace messages may be performed using the MSR$_{PMM}$ bit and the setting of DC4$_{PTMARK}$

13.11.9 Program Trace timing diagrams (2 MDO / 1 MSEO configuration)



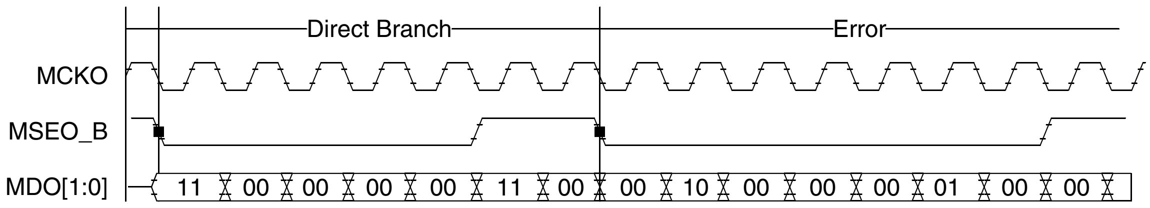
TCODE = 4
source processor = 0000, IS=1
# of sequential instructions = 128
relative address = 8'ha5

Figure 13-44. Program Trace — Indirect Branch Message (traditional)



TCODE = 28
source processor = 0000, IS=1
# of sequential instructions = 0
relative address = 8'ha5
branch history = 8'b10100101 (w/ stop)

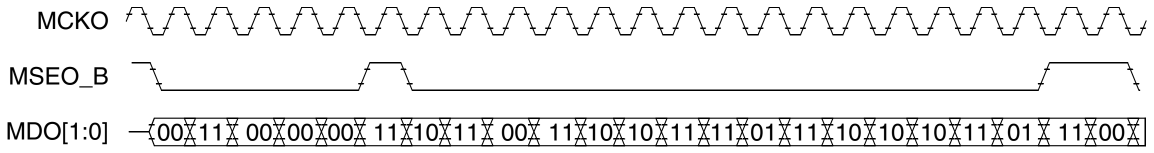
Figure 13-45. Program Trace — Indirect Branch Message (history)



DBM:
TCODE = 3
source processor = 0000
# of sequential instructions = 3

Error:
TCODE = 8
source processor = 0000
error code = 1 (queue overrun - BTM only)

Figure 13-46. Program Trace — Direct Branch (traditional) and error messages



TCODE = 12
source processor = 0000, IS = 1,
# of sequential instructions = 1
full target address = 32'hdeadface

Figure 13-47. Program Trace — Indirect Branch w/ sync. message

13.12 Data Trace

This section deals with the Data Trace mechanism supported by the Nexus 3 module. Data Trace is implemented via Data Write Messaging (DWM) and Data Read Messaging (DRM), as per the IEEE-ISTO 5001 standard.

13.12.1 Data Trace Messaging (DTM)

Data Trace Messaging for Zen is accomplished by snooping the Zen address and internal data buses, and storing the information for qualifying accesses (based on enabled features and matching target addresses). The Nexus 3 module traces all data access that meet the selected range and attributes.

NOTE

Data Trace is only performed on the Zen internal data buses. This allows for data visibility for Zen processors that incorporate a data cache. Only Zen CPU initiated accesses will be traced. No DMA accesses to the AHB system bus will be traced.

Data Trace Messaging can be enabled in one of two ways.

- Setting the TM field of the DC1 Register to enable Data Trace.
- Using the DTS field of the WT Register to enable Data Trace on Watchpoint hits (Zen watchpoints are configured within the Nexus1 module).

13.12.2 DTM Message formats

The Nexus 3 block supports five types of DTM Messages — Data Write, Data Read, Data Write Synchronization, Data Read Synchronization, and Error Messages.

13.12.2.1 Data Write Messages

The Data Write Message contains the data write value and the address of the write access, relative to the previous Data Trace Message. Data Write Message information is messaged out in the following format:

(1-64 bits)	(1-32 bits)	(4 bits)	(1 bit)	(4 bits)	(6 bits)
Data Value(s)*	Relative Address	Data Size	Data Space	Src. Proc	TCODE (000101)

Max length = 111 bits; Min length = 17 bits

Figure 13-48. Data Write Message Format

13.12.2.2 Data Read Messages

The Data Read Message contains the data read value and the address of the read access, relative to the previous Data Trace Message. Data Read Message information is messaged out in the following format:

(1-64 bits)	(1-32 bits)		(1 bit)	(4 bits)	(6 bits)
Data Value(s)*	Relative Address	Data Size	Data Space	Src. Proc	TCODE (000110)

Max length = 111 bits; Min length = 17 bits

Figure 13-49. Data Read Message format

NOTE

Zen based CPUs are capable of generating two (2) reads or writes per clock cycle in cases where multiple registers are accessed with a single instruction (lmw/stmw). These will have a double word pair size encoding (**p_tsiz** = 0b000). In these cases, the Nexus 3 module will send one (1) Data Trace Message with the two 32-bit data values as one combined 64-bit value for each message.

For Zen based CPUs, the doubleword encoding (**p_tsiz** = 0b000) may also indicate a doubleword access and will be sent out as a single Data Trace Message with a single 64-bit data value.

The debug/development tool will need to distinguish the two cases based on the family of Zen processor.

13.12.2.3 Data Trace Synchronization Messages

A Data Trace Write/Read with Sync. Message is messaged via the auxiliary port (provided Data Trace is enabled) for the following conditions (see Table 13-35):

- Initial Data Trace Message after exit from system reset or whenever Data Trace is enabled.
- Upon returning from a CPU Low Power state.
- Upon returning from Debug mode.
- After occurrence of queue overrun (can be caused by any trace message), provided Data Trace is enabled.
- After the periodic data trace counter has expired indicating 255 *without-sync* Data Trace Messages have occurred since the last *with-sync* message occurred.
- Upon assertion of the Event In (**nex_evti_b**) pin, the first Data Trace Message will be a synchronization message if the EIC bits of the DC1 Register have enabled this feature.
- Upon Data Trace Write/Read after the previous DTM Message was lost due to an attempted access to a secure memory location (for SOC's w/ security).
- Upon Data Trace Write/Read after the previous DTM Message was lost due to a collision entering the FIFO between the DTM Message and any two of the following: Watchpoint Message, Ownership Trace Message, or Program Trace Message.

Data Trace Synchronization Messages provide the full address (without leading zeros) and insure that development tools fully synchronize with Data Trace regularly. Synchronization messages provide a reference address for subsequent DTMs, in which only the unique portion of the Data Trace address is transmitted. The format for Data Trace Write/Read with Sync. Messages is as follows:

(1-64 bits)	(1-32 bits)	(4 bits)	(1 bit)	(4 bits)	(6 bits)
Data Value	Full Address	Data Size	Data Space	Source Proc.	TCODE (001101 or 001110)

Max length = 111 bits; Min length = 17 bits

Figure 13-50. Data write/read with sync. message format

Exception conditions that result in Data Trace Synchronization are summarized in Table 13-35.

Table 13-35. Data trace exception summary

Exception condition	Exception handling
System Reset Negation	At the negation of JTAG reset (**j_trst_b**), queue pointers, counters, state machines, and registers within the Nexus 3 module are reset. If Data Trace is enabled, the first Data Trace Message is a Data Write/Read w/ Sync. Message.
Data Trace Enabled	The first Data Trace Message (after Data Trace has been enabled) is a synchronization message.
Exit from Low Power/Debug	Upon exit from a Low Power mode or Debug mode the next Data Trace Message will be converted to a Data Write/Read with Sync. Message.
Queue Overrun	An Error Message occurs when a new message cannot be queued due to the message queue being full. The FIFO will discard messages until it has completely emptied the queue. Once emptied, an Error Message will be queued. The error encoding will indicate which type(s) of messages attempted to be queued while the FIFO was being emptied. The next DTM message in the queue will be a Data Write/Read w/ Sync. Message.
Periodic Data Trace Sync.	A forced synchronization occurs periodically after 255 Data Trace Messages have been queued. A Data Write/Read w/ Sync. Message is queued. The periodic data trace message counter then resets.
Event In	If the Nexus module is enabled, a **nex_evti_b** assertion initiates a Data Trace Write/Read w/ Sync. Message upon the next data write/read (if Data Trace is enabled and the EIC bits of the DC1 Register have enabled this feature).
Attempted Access to Secure Memory	For SoCs that implement security, any attempted read or write to secure memory locations will temporarily disable Data Trace & cause the corresponding DTM to be lost. A subsequent read/write will queue a Data Trace Read/Write w/ Sync. Message.
Collision Priority	All Messages have the following priority: Instruction 0 (WPM $\rightarrow$ DQM $\rightarrow$ $\text{PCM}_{\text{PIDMSG}}$ $\rightarrow$ OTM $\rightarrow$ BTM $\rightarrow$ DTM) $\rightarrow$ Instruction1 (WPM $\rightarrow$ DQM $\rightarrow$ OTM $\rightarrow$ BTM $\rightarrow$ DTM), where instruction0 is the oldest instruction. A DTM Message that attempts to enter the queue at the same time as three other higher priority messages will be lost. A subsequent read/write will queue a Data Trace Read/Write w/ Sync. Message.

13.12.3 DTM operation

13.12.3.1 Data trace windowing

Data Write/Read Messages are enabled via the RWT field in the Data Trace Control Register (DTC) for each DTM channel. Data Trace windowing is achieved via the address range defined by the DTEA and

DTSA Registers and by the RC field in the DTC register. All Zen-initiated read/write accesses that fall inside or outside these address ranges, as programmed, are candidates to be traced.

13.12.3.2 Data access / instruction access data tracing

The Nexus3 module is capable of tracing either instruction access data or data access data and can be configured for either type of data trace by setting the DI1 field within the Data Trace Control Register. This setting applies to all DTM channels.

13.12.3.3 Data trace filtering

Data Trace filtering is available base on the settings of MSR_{PMM} and $DC4_{DTMARK}$.

13.12.3.4 Zen bus cycle special cases

Table 13-36. Zen bus cycle cases

Special Case	Action
Zen bus cycle aborted	Cycle ignored
Zen bus cycle with data error ($\overline{TEA}$)[1]	Data Trace Message discarded
Zen bus cycle completed without error[1]	Cycle captured & transmitted
Zen (AHB) bus cycle initiated by Nexus 3	Cycle ignored
Zen bus cycle is an instruction fetch	Cycle selectively ignored based on DTC_{DI} setting
Zen bus cycle accesses misaligned data (across 64-bit boundary) - both 1st & 2nd transactions within data trace range	1st & 2nd cycle captured & a single or a pair of DTM(s) is (are) transmitted (see Note)
Zen bus cycle accesses misaligned data (across 64-bit boundary) - 1st transaction within data trace range; 2nd transaction out of data trace range	1st & 2nd cycle captured & a single or a pair of DTM(s) is (are) transmitted (see Note)
Zen bus cycle accesses misaligned data (across 64-bit boundary) - 1st transaction within data trace range; 2nd transaction (regardless of within range or not) receives a bus error	Data Trace Message discarded
Zen bus cycle accesses misaligned data (across 64-bit boundary) - 1st transaction out of data trace range; 2nd transaction within data trace range	1st & 2nd cycle captured & a single or a pair of DTM(s) is (are) transmitted (see Note)
Zen bus cycle accesses misaligned data (across 64-bit boundary) - 1st transaction out of data trace range; 2nd transaction within range, receives a bus error	Data Trace Message discarded

[1] Buffering of stores in the CPU store buffer may generate a DTM prior to the actual memory access, regardless of an error termination condition from memory.

NOTE

For misaligned accesses (crossing 64-bit boundary), the access is broken into two accesses by the CPU. If either access is within the data trace range, a single DTM will be sent with a size encoding indicating the size of the original access (i.e. word), and the address indicating the original misaligned accesses, unless the misaligned access wraps over the end of a circular buffer when using the SPE2 specialized load or store with modify, mode=1000 (circular addressing). In this case, since the two portions of the misaligned access are not contiguous, two DTMs will be sent, one for each portion. The size encodings and the addresses of the DTMs will indicate the accessed bytes of data.

NOTE

A store to the cache's store buffer within the data trace range may initiate a DTM message prior to completion of the actual memory access.

13.12.4 Data Trace Timing Diagrams(8 MDO / 2 MSEO configuration)

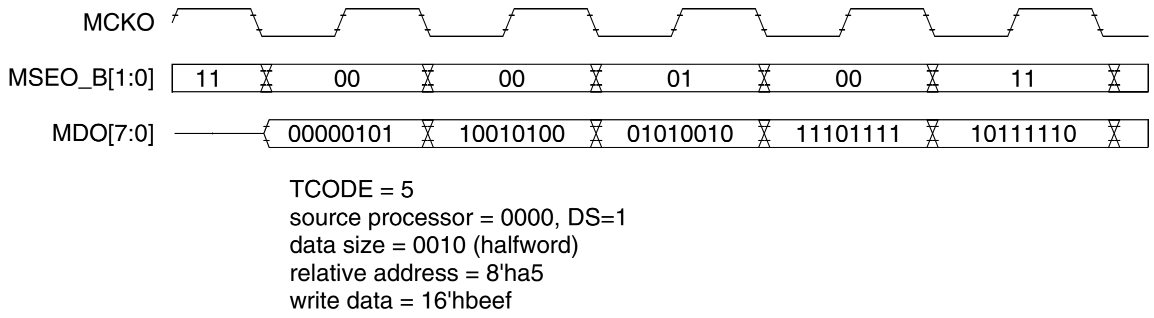


Figure 13-51. Data Trace — Data Write Message

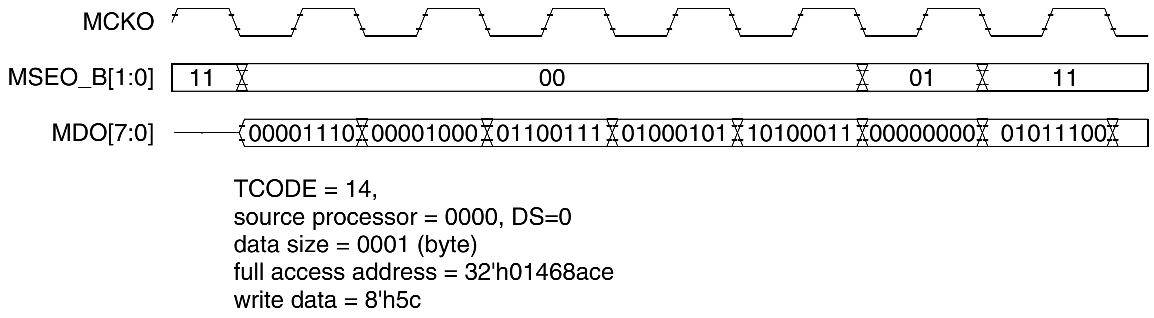


Figure 13-52. Data Trace — Data Read w/ Sync Message

13.13 Data Acquisition messaging

This section details the Data Acquisition mechanisms supported by the e200z759n3 Nexus 3 module. Data Acquisition Trace is implemented using Data Acquisition Trace Messages in accordance with IEEE-ISTO

5001 definitions. The control mechanism to export the data is different from the recommendations of the standard, however.

Data Acquisition Trace provides a convenient and flexible mechanism for the debugger to observe the architectural state of the machine through software instrumentation.

13.13.1 Data Acquisition ID Tag field

The DQTAG Tag field (DQTAG) is a 8-bit value specifying control or attribute information for the data included in the Data Acquisition Message. DQTAG is sampled from DEVENT$_{DQTAG}$ when a write to DDAM is performed via **mtspr** operations. The usage of the DQTAG is left to the discretion of the development tool to be used in whatever manner is deemed appropriate for the application.

13.13.2 Data Acquisition Data field

The Data Acquisition Data field (DQDATA) is the data captured from the DDAM write operation via **mtspr** operations. Leading zeros are omitted from the message.

13.13.3 Data Acquisition Trace event

For DQM, a dedicated SPR has been allocated (DDAM). It is expected that the general use case is to instrument the software and use **mtspr** operations to generate Data Acquisition Messages.

There is no explicit error response for failed accesses as a result of contention between an internal and external debugger. Software may be blocked or given ownership of DDAM and the DQTAG field of the DEVENT register via control in DBERC0 while in External Debug Mode. Hardware always has access to these registers. Refer to Section 12.3.4, Debug External Resource Control register (DBERC0), for more detail on DBERC0.

Reads from the Data Acquisition channel do not generate a Data Acquisition event and will return zeroes for the read data.

(1-32 bits)	(8 bits)	(4 bits)	(6 bits)
DQDATA	DQTAG	Src. Proc.	TCODE (000111)

Max length = 50 bits; Min length = 19 bits

Figure 13-53. Data Acquisition Message format

13.14 Watchpoint Trace Messaging

Enabling Watchpoint Messaging is done by setting the Watchpoint Trace Enable bit in the DC1 Register. Setting the individual Watchpoint sources is supported through the Zen Nexus1 module and the Performance Monitor unit. The Zen Nexus1 module is capable of setting multiple types of watchpoints. Please refer to the Debug chapter for details on Watchpoint initialization.

When watchpoints occur due to one or more asserted watchpoint event signals and Watchpoint Trace Messaging is enabled, a Watchpoint Trace message will be sent to the message queue to be messaged out. This message includes the watchpoint number indicating which watchpoint(s) caused the message. If more than one enabled watchpoint occurs in a single cycle, only one Watchpoint Trace message is generated and multiple bits of the watchpoint hit field will be set. The settings of the WMSK_{WEM} field control which watchpoints are enabled to generate watchpoint trace messages.

The occurrence of any of the Zen defined watchpoints can also be programmed to assert the Event Out (**nex_evto_b**) pin for one (1) period of the output clock (**nex_mcko**) based on settings in the DC2 and DC3 registers. See Table 13-39 for details on **nex_evto_b**.

Watchpoint information is messaged out in the following format:

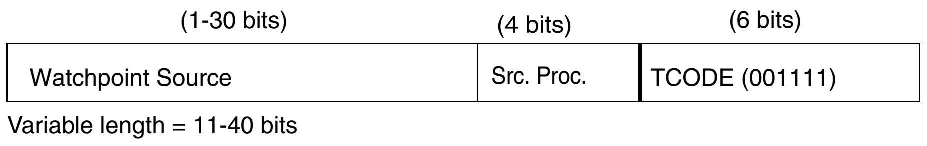


Figure 13-54. Watchpoint Message format

The Watchpoint Source message field will contain a '1' for each asserted watchpoint. Leading zeros are truncated.

Table 13-37. Watchpoint source encoding

Watchpoint source (1-30 bits)	Watchpoint description
000000000000000000000000000000	No Watchpoints enabled for Watchpoint Trace Messaging
*xxxxxxxxxxxxxxxxxxxxxxxxxxxxx*1	Watchpoint #0 enabled for WTM
*xxxxxxxxxxxxxxxxxxxxxxxxxxxx*1 *x*	Watchpoint #1 enabled for WTM
*xxxxxxxxxxxxxxxxxxxxxxxxxxx*1 *xx*	Watchpoint #2 enabled for WTM
*xxxxxxxxxxxxxxxxxxxxxxxxxx*1 *xxx*	Watchpoint #3 enabled for WTM
*xxxxxxxxxxxxxxxxxxxxxxxxx*1 *xxxx*	Watchpoint #4 enabled for WTM
*xxxxxxxxxxxxxxxxxxxxxxxx*1 *xxxxx*	Watchpoint #5 enabled for WTM
*xxxxxxxxxxxxxxxxxxxxxxx*1 *xxxxxx*	Watchpoint #6 enabled for WTM
*xxxxxxxxxxxxxxxxxxxxxx*1 *xxxxxxx*	Watchpoint #7 enabled for WTM
*xxxxxxxxxxxxxxxxxxxxx*1 *xxxxxxxx*	Watchpoint #8 enabled for WTM
*xxxxxxxxxxxxxxxxxxxx*1 *xxxxxxxxx*	Watchpoint #9 enabled for WTM
*xxxxxxxxxxxxxxxxxxx*1 *xxxxxxxxxx*	Watchpoint #10 enabled for WTM
*xxxxxxxxxxxxxxxxxx*1 *xxxxxxxxxxx*	Watchpoint #11 enabled for WTM
*xxxxxxxxxxxxxxxxx*1 *xxxxxxxxxxxx*	Watchpoint #12 enabled for WTM
*xxxxxxxxxxxxxxxx*1 *xxxxxxxxxxxxx*	Watchpoint #13 enabled for WTM
*xxxxxxxxxxxxxxx*1 *xxxxxxxxxxxxxx*	Watchpoint #14 enabled for WTM
*xxxxxxxxxxxxxx*1 *xxxxxxxxxxxxxxx*	Watchpoint #15 enabled for WTM
*xxxxxxxxxxxxx*1 *xxxxxxxxxxxxxxxx*	Watchpoint #16 enabled for WTM
*xxxxxxxxxxxx*1 *xxxxxxxxxxxxxxxxx*	Watchpoint #17 enabled for WTM
*xxxxxxxxxxx*1 *xxxxxxxxxxxxxxxxxx*	Watchpoint #18 enabled for WTM
*xxxxxxxxxx*1 *xxxxxxxxxxxxxxxxxxx*	Watchpoint #19 enabled for WTM
*xxxxxxxxx*1 *xxxxxxxxxxxxxxxxxxxx*	Watchpoint #20 enabled for WTM
*xxxxxxxx*1 *xxxxxxxxxxxxxxxxxxxxx*	Watchpoint #21 enabled for WTM
*xxxxxxx*1 *xxxxxxxxxxxxxxxxxxxxxx*	Watchpoint #22 enabled for WTM
*xxxxxx*1 *xxxxxxxxxxxxxxxxxxxxxxx*	Watchpoint #23 enabled for WTM
*xxxxx*1 *xxxxxxxxxxxxxxxxxxxxxxxx*	Watchpoint #24 enabled for WTM
*xxxx*1 *xxxxxxxxxxxxxxxxxxxxxxxxx*	Watchpoint #25 enabled for WTM
*xxx*1 *xxxxxxxxxxxxxxxxxxxxxxxxxx*	Watchpoint #26 enabled for WTM
*xx*1 *xxxxxxxxxxxxxxxxxxxxxxxxxxx*	Watchpoint #27 enabled for WTM
*x*1 *xxxxxxxxxxxxxxxxxxxxxxxxxxxx*	Watchpoint #28 enabled for WTM
1 *xxxxxxxxxxxxxxxxxxxxxxxxxxxxx*	Watchpoint #29 enabled for WTM

13.14.1 Watchpoint Timing Diagram (2 MDO / 1 MSEO configuration)

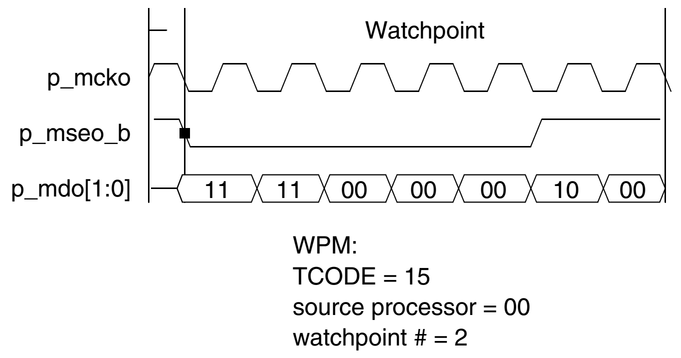


Figure 13-55. Watchpoint Message and Watchpoint Error Message

13.15 Nexus 3 read/write access to memory-mapped resources

The Read/Write access feature allows access to memory-mapped resources via the JTAG/OnCE port. The Read/Write mechanism supports single as well as block reads and writes to Zen AHB resources.

The Nexus 3 module is capable of accessing resources on the Zen system bus (AHB). Memory-mapped registers and other non-cached memory can be accessed via the standard memory map settings.

All accesses are setup and initiated by the Read/Write Access Control/Status Register (RWCS), as well as the Read/Write Access Address (RWA) and Read/Write Access Data Registers (RWD). Nexus 3 read/write accesses are run as privileged data non-cacheable, non-global accesses by default, and drive the **p_d_hprot[5:0]** bus access attributes to 6‘b000011 and the **p_d_gbl** access attribute to 0 accordingly. The $\text{RWCS}_{\text{ATTR}}$ field is provided to allow a portion of these default values to be modified when performing read or write accesses using the Nexus 3 Read/Write access mechanism.

Using the Read/Write Access Registers (RWCS/RWA/RWD), memory mapped Zen AHB resources can be accessed through Nexus 3. The following subsections describe the steps required to access memory-mapped resources.

NOTE

Read/Write Access can only access memory mapped resources when system reset is de-asserted and clocks are running.

Misaligned accesses are NOT supported in the zen Nexus3 module.

13.15.1 Single write Access

1. Initialize the Read/Write Access Address Register (RWA) through the access method outlined in Section 13.5, Nexus 3 register access via JTAG/OnCE. Configure as follows:
   a) Write Address → 32h’xxxxxxxx (write address)
2. Initialize the Read/Write Access Control/Status Register (RWCS) through the access method outlined in Section 13.5, Nexus 3 register access via JTAG/OnCE. Configure the bits as follows:
   a) Access Control (AC) → 1b’1 (to indicate start access)
   b) Map Select (MAP) → 3b’000 (primary memory map)
   c) Access Priority (PR) → 2b’00 (lowest priority)
   d) Read/Write (RW) → 1b’1 (write access)
   e) Word Size (SZ) → 3b’0xx (32-bit, 16-bit, 8-bit)
   f) Access Count (CNT) → 14h’0000 or 14h’0001(single access)

NOTE

Access Count (CNT) of 14’h0000 or 14’h0001 will perform a single access.

3. Initialize the Read/Write Access Data Register (RWD) through the access method outlined in Section 13.5, Nexus 3 register access via JTAG/OnCE. Configure as follows:
   a) Write Data → 32h’xxxxxxxx (write data)

4. The Nexus block will then arbitrate for the AHB system bus and transfer the data value from the data buffer RWD Register to the memory mapped address in the Read/Write Access Address Register (RWA). When the access has completed without error (ERR=1’b0), Nexus asserts the **nex_rdy_b** pin (see Table 13-39 for detail on **nex_rdy_b**) and clears the DV bit in the RWCS Register. This indicates that the device is ready for the next access.

NOTE

Only the **nex_rdy_b** pin as well as the DV and ERR bits within the RWCS provide Read/Write Access status to the external development tool.

13.15.2 Block write access

1. For a block write access, follow Steps 1, 2, and 3 outlined in Section 13.15.1, Single write Access, to initialize the registers, but using a value greater than one (14’h0001) for the CNT field in the RWCS Register.
2. The Nexus block will then arbitrate for the AHB system bus and transfer the first data value from the RWD Register to the memory mapped address in the Read/Write Access Address Register (RWA). When the transfer has completed without error (ERR=1’b0), the address from the RWA Register is incremented to the next word size (specified in the SZ field) and the number from the CNT field is decremented. Nexus will then assert the **nex_rdy_b** pin. This indicates that the device is ready for the next access.
3. Repeat Step 3 in Section 13.15.1, Single write Access, until the internal CNT value is zero (0). When this occurs, the DV bit within the RWCS will be cleared to indicate the end of the block write access.

NOTE

The actual RWA value as well as the CNT field within the RWCS are not changed when executing a block write access. The original values can be read by the external development tool at any time.

13.15.3 Single read access

1. Initialize the Read/Write Access Address Register (RWA) through the access method outlined in Section 13.5, Nexus 3 register access via JTAG/OnCE. Configure as follows:
   a) Read Address → 32h’xxxxxxxx (read address)
2. Initialize the Read/Write Access Control/Status Register (RWCS) through the access method outlined in Section 13.5, Nexus 3 register access via JTAG/OnCE. Configure the bits as follows:
   a) Access Control (AC) → 1b’1 (to indicate start access)
   b) Map Select (MAP) → 3b’000 (primary memory map)
   c) Access Priority (PR) → 2b’00 (lowest priority)
   d) Read/Write (RW) → 1b’0 (read access)
   e) Word Size (SZ) → 3b’0xx (32-bit, 16-bit, 8-bit)
   f) Access Count (CNT) → 14h’0000 or 14h’0001(single access)

NOTE

Access Count (CNT) of 14’h0000 or 14’h0001 will perform a single access.

3. The Nexus block will then arbitrate for the AHB system bus and the read data will be transferred from the AHB to the RWD Register. When the transfer is completed without error (ERR=1’b0), Nexus asserts the **nex_rdy_b** pin (see Table 13-39 for detail on **nex_rdy_b**) and sets the DV bit in the RWCS Register. This indicates that the device is ready for the next access.
4. The data can then be read from the Read/Write Access Data Register (RWD) through the access method outlined in Section 13.5, Nexus 3 register access via JTAG/OnCE.

NOTE

Only the **nex_rdy_b** pin as well as the DV and ERR bits within the RWCS provide Read/Write Access status to the external development tool.

13.15.4 Block read access

1. For a block read access, follow Steps 1 and 2 outlined in Section 13.15.3, Single read access, to initialize the registers, but using a value greater than one (14’h0001) for the CNT field in the RWCS Register.
2. The Nexus block will then arbitrate for the AHB system bus and the read data will be transferred from the AHB to the RWD Register. When the transfer has completed without error (ERR=1’b0), the address from the RWA Register is incremented to the next word size (specified in the SZ field) and the number from the CNT field is decremented. Nexus will then assert the **nex_rdy_b** pin. This indicates that the device is ready for the next access.
3. The data can then be read from the Read/Write Access Data Register (RWD) through the access method outlined in Section 13.5, Nexus 3 register access via JTAG/OnCE.
4. Repeat Steps 3 and 4 in Section 13.15.3, Single read access, until the CNT value is zero (0). When this occurs, the DV bit within the RWCS is set to indicate the end of the block read access.

NOTE

The data values must be shifted out 32-bits at a time LSB first (i.e. doubleword read = two word reads from the RWD).

NOTE

The actual RWA value as well as the CNT field within the RWCS are not changed when executing a block read access. The original values can be read by the external development tool at any time.

13.15.5 Error handling

The Nexus 3 module handles various error conditions as follows:

13.15.5.1 AHB read/write error

All address and data errors that occur on read/write accesses to the Zen AHB system bus will return a transfer error encoding on the **p_hresp[1:0]** signals. If this occurs:

1. The access is terminated without re-trying (AC bit is cleared)
2. The ERR bit in the RWCS Register is set
3. The Error Message is sent (TCODE = 8) indicating Read/Write Error

13.15.5.2 Access termination

The following cases are defined for sequences of the Read/Write protocol that differ from those described in the above sections.

1. If the AC bit in the RWCS Register is set to start Read/Write accesses and invalid values are loaded into the RWD and/or RWA, then an AHB access error may occur. This is handled as described above.
2. If a block access is in progress (all cycles not completed), and the RWCS Register is written, then the original block access is terminated at the boundary of the nearest completed access.
   a) If the RWCS is written with the AC bit set, the next Read/Write access will begin and the RWD can be written to/ read from.
   b) If the RWCS is written with the AC bit cleared, the Read/Write access is terminated at the nearest completed access. This method can be used to break (early terminate) block accesses.

13.15.6 Read/write access error message

The Read/Write Access Error Message is sent out when an AHB system bus access error (read or write) has occurred.

Error information is messaged out in the following format:

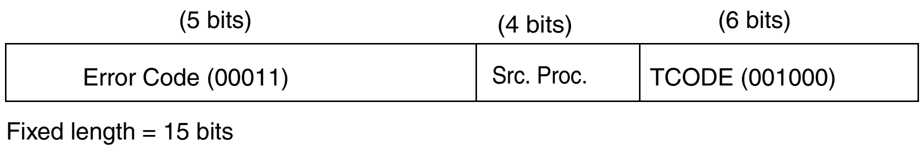


Figure 13-56. Error message format

13.16 Nexus 3 pin interface

This section details information regarding the Nexus 3 pins and pin protocol.

The Nexus 3 pin interface provides the function of transmitting messages from the messages queues to the external tools. It is also responsible for handshaking with the message queues.

13.16.1 Pins implemented

The Nexus 3 module implements an auxiliary port consisting of one (1) **nex_evti_b** and one (1) **nex_mseo_b** or two (2) **nex_mseo_b[1:0]**. It also implements a configurable number of **nex_mdo[n:0]** pins, (1) **nex_rdy_b** pin, (1) **nex_evto_b** pin, (4) **nex_wevto[3:0]** pins, and one (1) clock output pin (**nex_mcko**), as well as additional configuration pins described in Table 13-39. The output pins are synchronized to the Nexus 3 output clock (**nex_mcko**).

All Nexus 3 input functionality is controlled through the JTAG/OnCE port in compliance with IEEE 1149.1 (see Section 13.5, Nexus 3 register access via JTAG/OnCE, for details). The JTAG pins are incorporated as I/O to the Zen processor, and are further described in Section 12.4.3, JTAG/OnCE pins.

Table 13-38. JTAG pins for Nexus 3

JTAG pins	Input/ output	Description of JTAG pins (included in Zen Nexus 1)
j_tdo	O	The Test Data Output (**j_tdo**) pin is the serial output for test instructions and data. **j_tdo** is three-stateable and is actively driven in the "Shift-IR" and "Shift-DR" controller states. **j_tdo** changes on the falling edge of **j_tclk**.
j_tdi	I	The Test Data Input (**j_tdi**) pin receives serial test instruction and data. TDI is sampled on the rising edge of **j_tclk**.
j_tms	I	The Test Mode Select (**j_tms**) input pin is used to sequence the OnCE controller state machine. **j_tms** is sampled on the rising edge of **j_tclk**.
j_tclk	I	The Test Clock (**j_tclk**) input pin is used to synchronize the test logic, and control register access through the JTAG/OnCE port.
j_trst_b	I	The Test Reset (**j_trst_b**) input pin is used to asynchronously initialize the JTAG/OnCE controller.

The auxiliary pins are used to send and receive messages and are described in Table 13-39.

Table 13-39. Nexus 3 auxiliary pins

Auxiliary pins	Input/ output	Description of auxiliary pins
nex_mcko	O	Message Clock Out (**nex_mcko**) is a free running output clock to development tools for timing of **nex_mdo[n:0]** & **nex_mseo_b[1:0]** pin functions. **nex_mcko** is programmable through the DC1 Register.
nex_mdo[n:0]	O	Message Data Out (**nex_mdo[n:0])** are output pins used for OTM, BTM, and DTM. External latching of **nex_mdo[n:0]** shall occur on the rising edge of the Nexus3 clock (**nex_mcko**).
nex_mseo_b[1:0]	O	Message Start/End Out (**nex_mseo_b[1:0]**) are output pins that indicate when a message on the **nex_mdo[n:0]** pins has started, when a variable length packet has ended, and when the message has ended. External latching of **nex_mseo_b[1:0]** shall occur on the rising edge of the Nexus3 clock (**nex_mcko**). One or two pin MSEO functionality is determined at integration time per SOC implementation
nex_rdy_b	O	Ready (**nex_rdy_b**) is an output pin used to indicate to the external tool that the Nexus block is ready for the next Read/Write Access. If Nexus is enabled, this signal is asserted upon successful (without error) completion of an AHB system bus transfer (Nexus read or write) & is held asserted until the JTAG/OnCE state machine reaches the "Capture_DR" state. Upon exit from system reset or if Nexus is disabled, **nex_rdy_b** remains de-asserted

Table 13-39. Nexus 3 auxiliary pins (continued)

Auxiliary pins	Input/ output	Description of auxiliary pins
nex_evto_b	O	Event Out (**nex_evto_b**)is an output that, when asserted, indicates one of two events has occurred based on the EOC bits in the DC1 Register. **nex_evto_b** is held asserted for one (1) cycle of **nex_mcko**: 1) one (or more) watchpoints has occurred (from Nexus1) & EOC = 2'b00 2) debug mode was entered (jd_debug_b asserted from Nexus1) & EOC = 2'b01
nex_evti_b	I	Event In (**nex_evti_b**) is an input that, when asserted, will initiate one of two events based on the EIC bits in the DC1 Register (if the Nexus module is enabled at reset): 1) Program Trace & Data Trace synchronization messages (provided Program Trace & Data Trace are enabled & EIC = 2'b00). 2) Debug request to Zen Nexus1 module (provided EIC = 2'b01 and this feature is implemented).
nex_wevto[3:0]	O	Watchpoint Event Out 3:0 (**nex_wevto[3:0]**) are outputs that, when asserted, indicate one or more watchpoint events has occurred based on the settings in the DC2 and DC3 registers. **nex_wevto[3:0]** is held asserted for one (1) cycle of **nex_mcko**.
nex_ext_src_id[0:3]	I	nex_ext_src_id[0:3] is used to provide the SRC field value used in each message. These pins are tied to a predetermined value at SoC integration time

The Nexus auxiliary port arbitration pins are used when the Nexus 3 module is implemented in a multi-Nexus SoC that shares a single auxiliary output port. The arbitration is controlled by an SoC level Nexus Port Control module (NPC). Refer to Section 13.18, Auxiliary port arbitration, for detail on Nexus port arbitration.

Table 13-40. Nexus port arbitration signals

Nexus port arbitration pins	Input/ output	Description of arbitration pins
nex_aux_req[1:0]	O	Nexus Auxiliary Request (**nex_aux_req[1:0]**) output signals indicate to an SoC level Nexus arbiter a request for access to the shared Nexus auxiliary port in a multi-Nexus implementation. The priority encodings are determined by how many messages are currently in the message queues (see Table 13-42).
nex_aux_busy	O	Nexus Auxiliary Busy (**nex_aux_busy**) is an output signal to an SoC level Nexus arbiter indicating that the Nexus 3 module is currently transmitting its message after being granted the Nexus auxiliary port.
npc_aux_grant	I	Nexus Auxiliary Grant (**npc_aux_grant**) is an input from the SoC level Nexus Port Controller (NPC) that the auxiliary port has been granted to the Nexus 3 module to transmit its message.
ext_multi_nex_sel	I	Multi-Nexus Select (**ext_multi_nex_sel**) is a static signal indicating that the Nexus 3 module is implemented within a multi-Nexus environment. If set, port control and arbitration is controlled by the SoC level arbitration module (NPC).

13.16.2 Pin protocol

The protocol for the Zen processor transmitting messages via the auxiliary pins is accomplished with the MSEO pin function outlined in Table 13-41. Both single and dual pin cases are shown.

nex_mseo_b[1:0] is used to signal the end of variable-length packets, and not fixed length packets. **nex_mseo_b[1:0]** is sampled on the rising edge of the Nexus 3 clock (**nex_mcko**).

Table 13-41. MSEO pin(s) protocol

nex_mseo_b function	Single nex_mseo_b data (serial)	Dual nex_mseo_b[1:0] data
Start of message	1-1-0	11-00
End of message	0-1-1-(more 1's)	00 (or 01)-11-(more 1's)
End of variable length packet	0-1-0	00-01
Message transmission	0's	00's
Idle (no message)	1's	11's

Figure 13-57 illustrates the state diagram for single pin MSEO transfers.

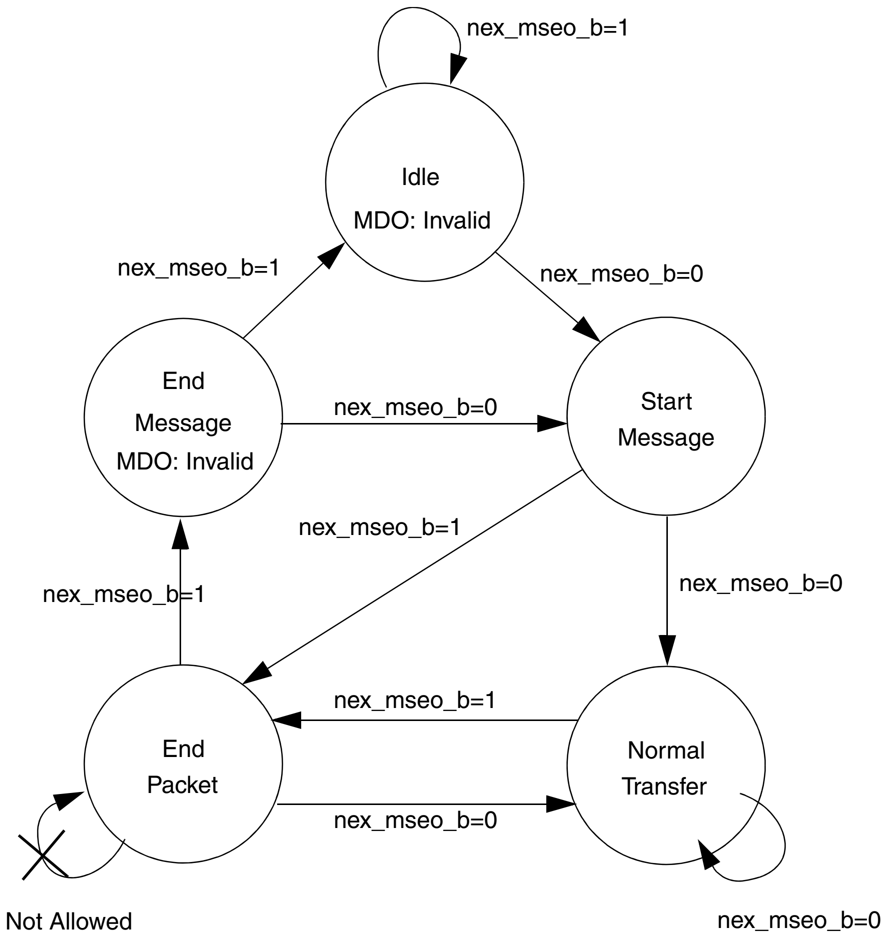


Figure 13-57. Single pin MSEO transfers

Note that the "End Message" state does not contain valid data on the **nex_mdo[n:0]** pins. Also, It is not possible to have two consecutive "End Packet" messages. This implies the minimum packet size for a variable length packet is 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 '1's on the **nex_mseo_b** pin before the actual end of message.

Figure 13-58 illustrates the state diagram for dual pin MSEO transfers.

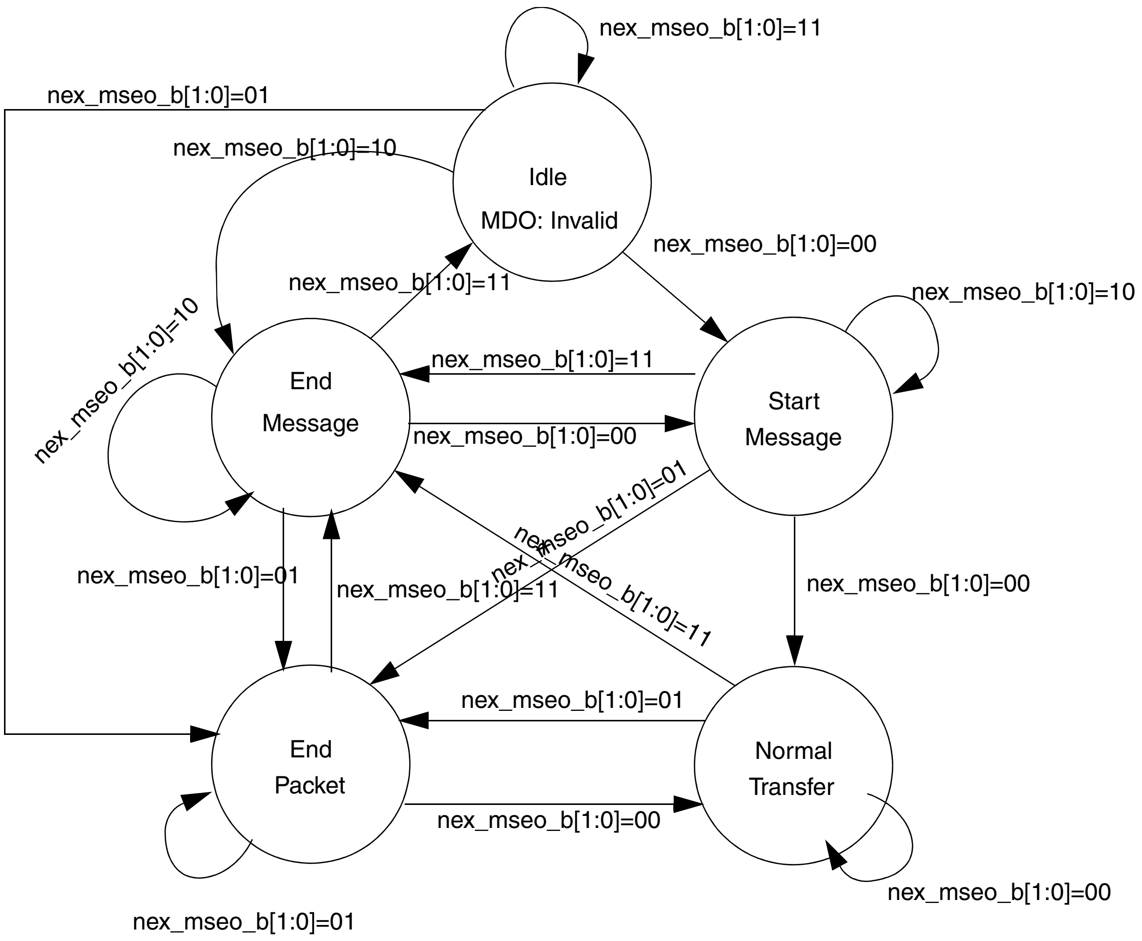


Figure 13-58. Dual pin MSEO transfers

The dual pin MSEO option is more robust that the single pin option. Termination of the current message may immediately be followed by the start of the next message on the consecutive clocks. An extra clock to end the message is not necessary as with the one MSEO pin option. The dual pin option also allows for consecutive "End Packet" states. This can be an advantage when small, variable sized packets are transferred.

NOTE

The "End Message" state may also indicate the end of a variable-length packet as well as the end of the message when using the dual pin option.

13.17 Rules for output messages

Zen based Class 3-compliant embedded processors must provide messages via the auxiliary port in a consistent manner as described below:

- A variable-sized packet within a message must end on a port boundary.
- A variable-sized packet may start within a port boundary only when following a fixed length packet. (If two variable-sized packets end and start on the same clock, it is impossible to know which bit is from the last packet and which bit is from the next packet.)
- Whenever a variable-length packet is sized such that it does not end on a port boundary, it is necessary to extend and zero fill the remaining bits after the highest-order bit so that it can end on a port boundary.

  For example, if the **nex_mdo[n:0]** port is 2 bits wide, and the unique portion of an indirect address TCODE is 5 bits, then the remaining 1 bit of **nex_mdo[n:0]** must be packed with a 0.

13.18 Auxiliary port arbitration

In a multi-Nexus environment, the Nexus 3 module must arbitrate for the shared Nexus port at the SoC level.The request scheme is implemented as a 2-bit request with various levels of priority. The priority levels are defined in Table 13-42 below. The Nexus 3 module will receive a 1-bit grant signal (**npc_aux_grant**) from the SoC level arbiter. When a grant is received, the Nexus 3 module will begin transmitting its message following the protocol outlined in Section 13.16.2, Pin protocol. The Nexus 3 module will maintain control of the port, by asserting the **nex_aux_busy** signal, until the $\overline{\text{MSEO}}$ state machine reaches the "End Message" state.

Table 13-42. MDO request encodings

Request Level	MDO Request Encoding (nex_aux_req[1:0])	Condition of Queue
No Request	00	No message to send
Low Priority	01	Message queue less than 1/2 full
—	10	Reserved
High Priority	11	Message queue 1/2 full or more

13.19 Examples

The following are examples of Program Trace and Data Trace Messages.

Table 13-43 illustrates an example Indirect Branch Message with 2 MDO / 1MSEO configuration. Table 13-44 illustrates the same example with an 8 MDO / 2 MSEO configuration.

Note that T0 and S0 are the least significant bits where:

- Tx = TCODE number (fixed)
- Sx = Source processor (fixed)
- MAP = Address Space Value (IS)

- Ix = Number of instructions (variable)
- Ax = Unique portion of the address (variable)

Note that during clock 13, the **nex_mdo[n:0]** pins are ignored in the single MSEO case.

Table 13-43. Indirect branch message example (2 MDO / 1 MSEO)

Clock	nex_mdo[1:0]		nex_mseo_b	State
0	X	X	1	Idle (or end of last message)
1	T1	T0	0	Start Message
2	T3	T2	0	Normal Transfer
3	T5	T4	0	Normal Transfer
4	S1	S0	0	Normal Transfer
5	S3	S2	0	Normal Transfer
6	I0	MAP	0	Normal Transfer
7	I2	I1	0	Normal Transfer
8	I4	I3	1	End Packet
9	A1	A0	0	Normal Transfer
10	A3	A2	0	Normal Transfer
11	A5	A4	0	Normal Transfer
12	A7	A6	1	End Packet
13	0	0	1	End Message
14	T1	T0	0	Start Message

Table 13-44. Indirect branch message example (8 MDO / 2 MSEO)

Clock	nex_mdo[7:0]								nex_mseo_b[1:0]		State
0	X	X	X	X	X	X	X	X	1	1	Idle (or end of last message)
1	S1	S0	T5	T4	T3	T2	T1	T0	0	0	Start Message
2	I4	I3	I2	I1	I0	MAP	S3	S2	0	1	End Packet
3	A7	A6	A5	A4	A3	A2	A1	A0	1	1	End Packet/End Message
4	S1	S0	T5	T4	T3	T2	T1	T0	0	0	Start Message

Table 13-45 & Table 13-46 illustrate examples of Direct Branch Messages: one with 2 MDO / 1 MSEO, and one with 8 MDO / 2 MSEO.

Note that T0 and I0 are the least significant bits where:

- Tx = TCODE number (fixed)
- Sx = Source processor (fixed)
- Ix = Number of Instructions (variable)

Table 13-45. Direct branch message example (2 MDO / 1 MSEO)

Clock	nex_mdo[1:0]		nex_mseo_b	State
0	X	X	1	Idle (or end of last message)
1	T1	T0	0	Start Message
2	T3	T2	0	Normal Transfer
3	T5	T4	0	Normal Transfer
4	S1	S0	0	Normal Transfer
5	S3	S2	0	Normal Transfer
6	I1	I0	1	End Packet
7	0	0	1	End Message

Table 13-46. Direct branch message example (8 MDO / 2 MSEO)

Clock	nex_mdo[7:0]								nex_mseo_b[1:0]		State
0	X	X	X	X	X	X	X	X	1	1	Idle (or end of last message)
1	S1	S0	T5	T4	T3	T2	T1	T0	0	0	Start Message
2	0	0	0	0	I1	I0	S3	S2	1	1	End Packet/End Message
3	S1	S0	T5	T4	T3	T2	T1	T0	0	0	Start Message

Table 13-47 illustrates an example Data Write Message with 8 MDO / 1 MSEO configuration, and Table 13-48 illustrates the same DWM with 8 MDO / 2 MSEO configuration.

Note that T0, A0, D0 are the least significant bits where:

- Tx = TCODE number (fixed)
- Sx = Source processor (fixed)
- MAP = Address Space Value (DS)
- Zx = Data size (fixed)
- Ax = Unique portion of the address (variable)
- Dx = Write data (variable— 8-, 16-, or 32-bit)

Table 13-47. Data Write Message Example (8 MDO / 1 MSEO)

Clock	nex_mdo[7:0]								nex_mseo_b	State
0	X	X	X	X	X	X	X	X	1	Idle (or end of last message)
1	S1	S0	T5	T4	T3	T2	T1	T0	0	Start Message
2	A0	Z3	Z2	Z1	Z0	DS	S3	S2	1	End Packet
3	D7	D6	D5	D4	D3	D2	D1	D0	0	Normal Transfer
4	0	0	0	0	0	0	0	0	1	End Packet
5	0	0	0	0	0	0	0	0	1	End Message

Table 13-48. Data write message example (8 MDO / 2 MSEO)

Clock	nex_mdo[7:0]								nex_mseo_b[1:0]		State
0	X	X	X	X	X	X	X	X	1	1	Idle (or end of last message)
1	S1	S0	T5	T4	T3	T2	T1	T0	0	0	Start Message
2	A0	Z3	Z2	Z1	Z0	DS	S3	S2	0	1	End Packet
3	D7	D6	D5	D4	D3	D2	D1	D0	1	1	End Packet/ End Message

13.20 Electrical characteristics

For all electrical characteristics related to Zen and Nexus 3 operation, please refer to the appropriate “Zen Integration Guide”.

13.21 IEEE 1149.1 (JTAG) RD/WR sequences

This section contains example JTAG/OnCE sequences used to access resources.

13.21.1 JTAG sequence for accessing internal Nexus registers

Table 13-49. Accessing internal Nexus 3 registers via JTAG/OnCE

Step #	TMS pin	Description
1	1	IDLE → SELECT-DR_SCAN
2	0	SELECT-DR_SCAN → CAPTURE-DR (Nexus Command Register value loaded in shifter)
3	0	CAPTURE-DR → SHIFT-DR
4	0	(7) TCK clocks issued to shift in direction (rd/wr) bit and first 6 bits of Nexus reg. addr.
5	1	SHIFT-DR → EXIT1-DR (7th bit of Nexus reg. shifted in)
6	1	EXIT1-DR → UPDATE-DR (Nexus shifter is transferred to Nexus Command Register)
7	1	UPDATE-DR → SELECT-DR_SCAN

Table 13-49. Accessing internal Nexus 3 registers via JTAG/OnCE (continued)

Step #	TMS pin	Description
8	0	SELECT-DR_SCAN → CAPTURE-DR (Register value is transferred to Nexus shifter)
9	0	CAPTURE-DR → SHIFT-DR
10	0	(31) TCK clocks issued to transfer register value to TDO pin while shifting in TDI value
11	1	SHIFT-DR → EXIT1-DR (MSB of value is shifted in/out of shifter)
12	1	EXIT1-DR → UPDATE -DR (if access is write, shifter is transferred to register)
13	0	UPDATE-DR → RUN-TEST/IDLE (transfer complete - Nexus controller to Reg. Select state)

13.21.2 JTAG sequence for read access of memory-mapped resources

Table 13-50. Accessing memory-mapped resources (reads)

Step #	TCLK clocks	Description
1	13	Nexus command = write to Read/Write Access Address Register (RWA)
2	37	Write RWA (initialize starting read address - data input on TDI)
3	13	Nexus Command = write to Read/Write Control/Status Register (RWCS)
4	37	Write RWCS (initialize read access mode and CNT value - data input on TDI)
5	—	Wait for falling edge of **nex_rdy_b** pin
6	13	Nexus command = read Read/Write Access Data Register (RWD)
7	37	Read RWD (data output on TDO)
8	—	If CNT > 0, go back to Step #5

13.21.3 JTAG sequence for write access of memory-mapped resources

Table 13-51. Accessing memory-mapped resources (writes)

Step #	TCLK clocks	Description
1	13	Nexus command = write to Read/Write Access Control/Status Register (RWCS)
2	37	Write RWCS (initialize write access mode and CNT value - data input on TDI)
3	13	Nexus command = write to Read/Write Address Register (RWA)
4	37	Write RWA (initialize starting write address - data input on TDI)
5	13	Nexus command = read Read/Write Access Data Register (RWD)
6	37	Write RWD (data output on TDO)
7	—	Wait for falling edge of **nex_rdy_b** pin
8	—	If CNT > 0, go back to Step #5

Chapter 14 External Core Complex Interfaces

This chapter describes the external interfaces to the e200z759n3 core complex. Signal descriptions as well as the data transfer protocols are documented in the following subsections.

The external interfaces encompass control and data signals supporting instruction and data transfers, support for interrupts, including vectored interrupt logic, reset support, power management interface signals, debug event signals, Time Base control and status information, processor state information, Nexus 1/3 / OnCE / JTAG interface signals, and a Test interface.

The e200z759n3 core complex is being designed to support an interchangeable BIU in order to support multiple system bus interface definitions. BIU support is planned for several standards including AMBA AHB 2.v6, AMBA AXI, etc. This chapter will focus on AMBA AHB 2.v6 initially, with additional sections added for other interfaces. Only a single interface standard will be present in any e200z759n3 instance. The specific module is selected at the synthesis stage.

The memory portion of the Zen core interface is comprised of a pair of 64-bit wide standard AHB 2.v6 system buses, one for instructions and the other for data. The data memory interface supports read and write transfers of 8, 16, 24, 32, and 64 bits, supports misaligned transfers, supports true big- and little-endian operating modes, and operates in a pipelined fashion. The instruction memory interface supports read transfers of 16, 32, and 64 bits, supports misaligned transfers, supports true big- and little-endian operating modes, and operates in a pipelined fashion.

The memory interface supported by the BIUs is based on the AHB 2.v6 definition. Additional sideband signals have been added to support additional control functions, and are described in this chapter.

NOTE

The AHB bit and byte ordering reflect a natural little-endian ordering, as used by the AMBA documentation. The e200z759n3 BIU will automatically perform the necessary byte lane conversions to support big-endian transfers. Memories and peripheral devices/interfaces should be wired according to byte lane addresses defined in Section 14.2.5, Byte lane specification, and Table 14-10.

Single-beat and misaligned transfers are supported for Cache-Inhibited read and write cycles, and write-buffer writes. Burst transfers (doubleword aligned) of four doublewords are supported for cache linefill and copyback operations.

Misaligned accesses are supported with one or more transfers to an interface. If an access is misaligned, but is contained within an aligned 64-bit doubleword, the core performs a single transfer, and the memory interface is responsible for delivering (reads) or accepting (writes) the data corresponding to the size and byte enable signals aligned according to the low order three address bits. If an access is misaligned and crosses a 64-bit boundary, the BIU will perform a pair of transfers beginning at the effective address for the first transfer, along with appropriate byte enables, and for the second transfer the address is incremented to the next 64-bit boundary, and the size and byte enable signals are driven to correspond to the number of remaining bytes to be transferred.

14.1 Signal index

This section contains an index of the e200z759n3 signals.

The following prefixes are used for e200z759n3 signal mnemonics:

- **m** denotes master clock and reset signals
- **p** denotes processor or core-related signals
- **j** denotes JTAG mode signals
- **jd** denotes JTAG and Debug mode signals
- **ipt** denotes Scan and Test Mode signals
- **nex** denotes Nexus signals

NOTE

The “_b” suffix denotes an active low signal. Signals without the active-low suffix are active high.

Figure 14-1 and Figure 14-2 groups core bus and control signals by function.

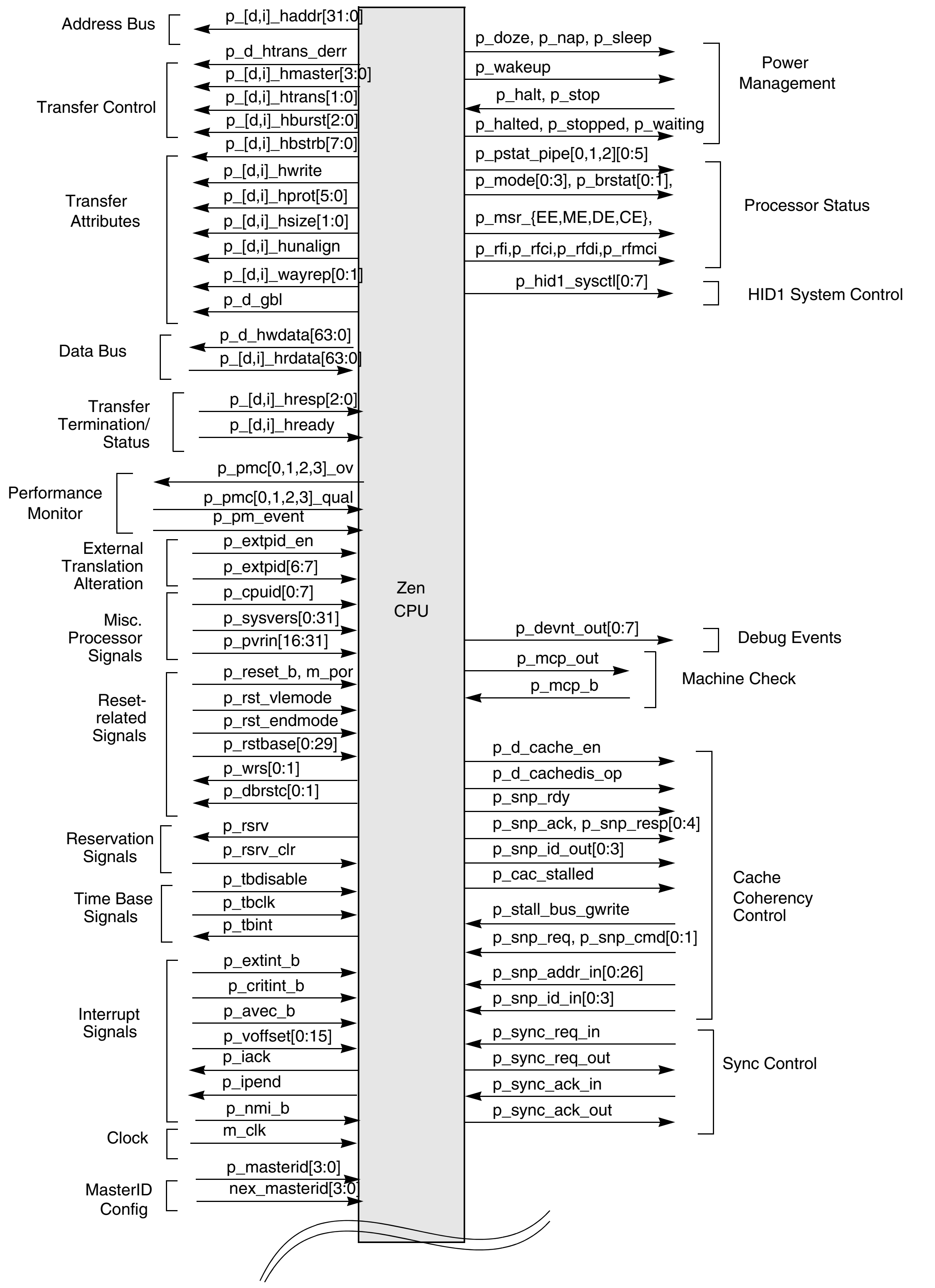


Figure 14-1. Zen signal groups - part 1

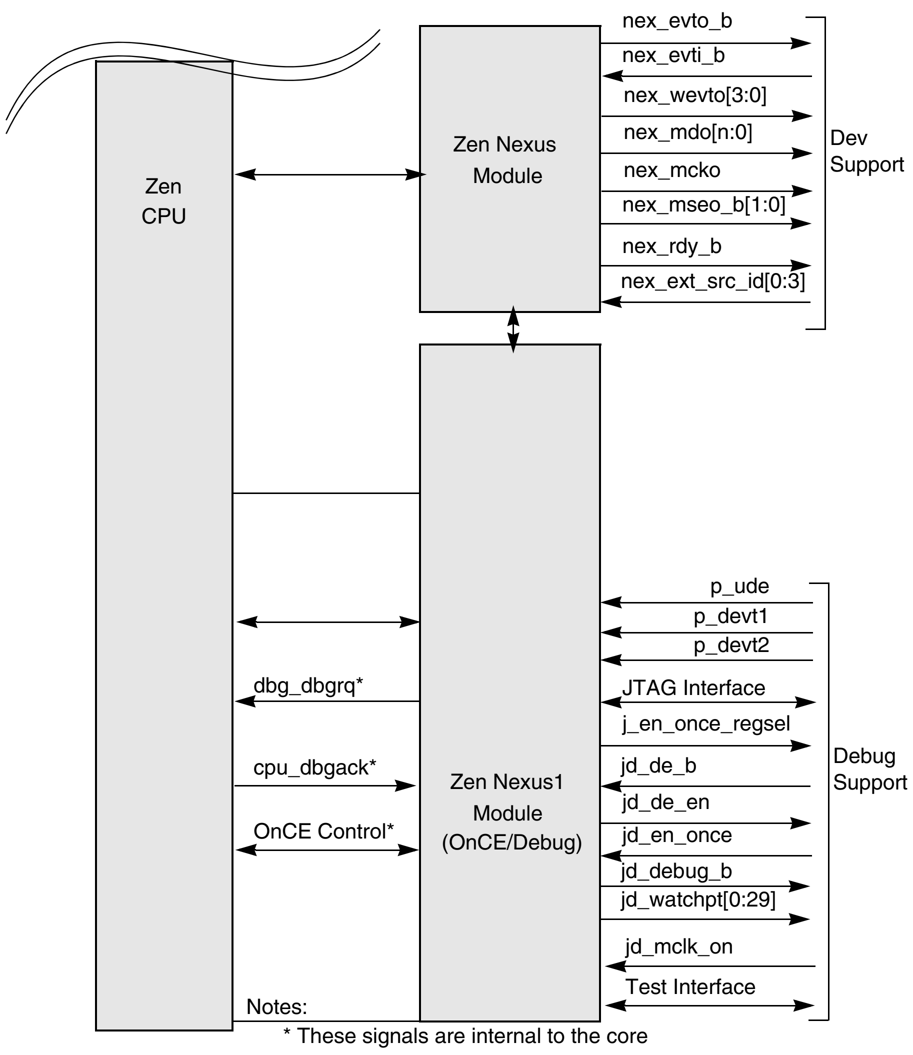


Figure 14-2. Zen signal groups - part 2

Table 14-1 shows e200z759n3 signal function and type, signal definition, and reset value. Signals are presented in functional groups.

Table 14-1. Interface signal definitions

Signal name	Type	Reset value	Definition
Clock and reset-related signals			
m_clk	I		Global system clock
m_por	I		Power-on reset
p_reset_b	I		Processor reset input
p_wrs[0:1]	O		Processor watchdog reset status outputs
p_dbrstc[0:1]	O		Processor debug reset control outputs
p_rstbase[0:29]	I		Reset exception handler base address

Table 14-1. Interface signal definitions (continued)

Signal name	Type	Reset value	Definition
p_rst_endmode	I		Reset endian mode select
p_rst_vlemode	I		Reset VLE mode select, value to be loaded into TLB entry 0 on reset.
Memory interface signals			
p_d_hmaster[3:0], p_i_hmaster[3:0]	O	-	Master ID
p_d_haddr[31:0], p_i_haddr[31:0]	O	-	Address buses
p_d_hwrite, p_i_hwrite*	O	0	Write signal (always driven low for p_i_hwrite)
p_d_hprot[5:0], p_i_hprot[5:0]	O	-	Protection Codes
p_d_htrans[1:0], p_i_htrans[1:0]	O	-	Transfer Type
p_d_htrans_derr	O	-	Transfer Data Parity error indicator (push errors)
p_d_hburst[2:0], p_i_hburst[2:0]	O	-	Burst Type
p_d_hsize[1:0], p_d_hsize[1:0]	O	-	Transfer Size
p_d_hunalign, p_i_hunalign	O	-	Indicates the current data access is a misaligned access.
p_d_gbl	O	-	Indicates the current access is marked as a globally coherent access.
p_d_hbstrb[7:0], p_i_hbstrb[7:0]	O	0	Byte strobes
p_d_hrdata[63:0], p_i_hrdata[63:0]	I		Read data buses
p_d_hwdata[63:0]	O	-	Write data bus
p_d_hready, p_i_hready	I		Transfer Ready
p_d_hresp[2:0], p_i_hresp[1:0]	I		Transfer Response
p_d_wayrep[0:1] p_i_wayrep[0:1]	0		Way replacement Indicates the cache way being replaced by a burst read linefill.
p_d_ahb_clken, p_i_ahb_clken	I		AHB Clock enable
Master ID configuration signals			
p_masterid[3:0]	I	-	CPU Master ID configuration
nex_masterid[3:0]	I	-	Nexus Master ID configuration
Sync control interface signals			
p_sync_req_in	I	-	Sync Request Input
p_sync_ack_in	I	-	Sync Acknowledge Input
p_sync_req_out	O	0	Sync Request Output
p_sync_ack_out	O	0	Sync Acknowledge Output

Table 14-1. Interface signal definitions (continued)

Signal name	Type	Reset value	Definition
Coherency control interface signals			
p_snp_req	I	-	Snoop Request
p_snp_cmd[0:1]	I	-	Snoop Command
p_snp_addr_in[0:31]	I	-	Snoop Address Input (bit 0 is MSB)
p_snp_id_in[0:3]	I	-	Snoop ID Input
p_stall_bus_gwrite	I	-	Stall External Bus Global Writes
p_snp_rdy	O	0	Snoop Ready
p_snp_ack	O	0	Snoop Acknowledge
p_snp_resp[0:4]	O	0	Snoop Response
p_snp_id_out[0:3]	O	-	Snoop ID Output
p_cac_stalled	O	0	CPU cache access Stalled
p_d_cache_en	O	0	Data cache enabled/disabled state
p_d_cachedis_op	O	0	Data cache disable operation in progress
Interrupt interface signals			
p_extint_b	I		External Input interrupt request
p_critint_b	I		Critical Input interrupt request
p_nmi_b	I		Non-Maskable Interrupt input request
p_avec_b	I		Autovector request Use internal interrupt vector offset
p_voffset[0:15]	I		Interrupt vector offset for vectored interrupts
p_iack	O	0	Interrupt Acknowledge. Indicates an interrupt is being acknowledge.
p_ipend	O	0	Interrupt Pending. Indicates an interrupt is pending internally.
p_mcp_b	I		Machine Check input request
External translation alteration signals			
p_extpid_en	I	-	External PID enable input
p_extpid[6:7]	I	-	External PID[6:7] input
Time base signals			
p_tbint	O	0	Time Base Interrupt
p_tbdisable	I	-	Time Base Disable input
p_tbclk	I	-	Time Base Clock input

Table 14-1. Interface signal definitions (continued)

Signal name	Type	Reset value	Definition
Misc. CPU signals			
p_cpuid[0:7]	I		CPU ID input
p_sysvers[0:31]	I		System Version inputs (for SVR)
p_pvrin[16:31]	I		Inputs for PVR
p_pid0[0:7]	O	0	PID0[24:31] outputs
p_pid0_updt	O	0	PID0 update status
p_hid1_sysctl[0:7]	O	0	HID1[16:23] outputs
CPU reservation signals			
p_rsrv	O	0	Reservation status
p_rsrv_clr	I		Clear Reservation flag
CPU state signals			
p_mode[0:3]	O	0	Indicates processor global status
p_pstat_pipe0[0:5], p_pstat_pipe1[0:5]	O	0	Indicates processor status for each pipe
p_brstat[0:1]	O	0	Indicates Branch prediction status
p_msr_EE, p_msr_DE, p_msr_CE, p_msr_ME	O	0	Reflect the values of these MSR bits
p_rfi, p_rfci, p_rfdi, p_rfmci	O	0	Reflect the execution of the corresponding instruction
p_mcp_out	O	0	Indicates a machine check has occurred
p_doze	O	0	Indicates low-power doze mode of operation
p_nap	O	0	Indicates low-power nap mode of operation
p_sleep	O	0	Indicates low-power sleep mode of operation
p_wakeup	O	0	Indicates to external clock control module to enable clocks and exit from low-power mode
p_halt	I		CPU halt request
p_halted	O	0	CPU halted
p_stop	I		CPU stop request
p_stopped	O	0	CPU stopped
p_waiting	O	0	CPU waiting
CPU performance monitor signals			
p_pm_event	I	-	Performance Monitor Event input
p_pmc0_ov	O	0	Performance Monitor Counter 0 OV bit

Table 14-1. Interface signal definitions (continued)

Signal name	Type	Reset value	Definition
p_pmc1_ov	O	0	Performance Monitor Counter 1 OV bit
p_pmc2_ov	O	0	Performance Monitor Counter 2 OV bit
p_pmc3_ov	O	0	Performance Monitor Counter 3 OV bit
p_pmc0_qual	I		Performance Monitor Counter 0 trigger qualifier input
p_pmc1_qual	I		Performance Monitor Counter 1 trigger qualifier input
p_pmc2_qual	I		Performance Monitor Counter 2 trigger qualifier input
p_pmc3_qual	I		Performance Monitor Counter 3 trigger qualifier input
CPU debug event signals			
p_ude	I		Unconditional Debug Event
p_devt1	I		Debug Event 1 input
p_devt2	I		Debug Event 2 input
p_devnt_out[0:7]	O	0	Debug Event outputs
Debug/emulation support signals (Nexus 1/OnCE)			
jd_en_once	I		Enable full OnCE operation
jd_debug_b	O	1	Indicates processor has entered debug session
jd_de_b	I		Debug request
jd_de_en	O	0	Active -high output enable for DE_b open-drain IO cell
jd_mclk_on	I		Indicates the system clock controller is actively toggling **m_clk**
jd_watchpt[0:29]	O	0	Indicate a watchpoint has occurred
Development support signals (Nexus 3)			
nex_mcko	O		Nexus 3 Clock Output
nex_rdy_b	O		Nexus 3 Ready Output
nex_evto_b	O		Nexus 3 Event-Out Output
nex_wevto[3:0]	O		Nexus 3 Watchpoint Event-Out Output
nex_evti_b	I		Nexus 3 Event-In Input
nex_mdo[n:0]	O		Nexus 3 Message Data Output
nex_mseo_b[1:0]	O		Nexus 3 Message Start/End Output
JTAG-related signals			

Table 14-1. Interface signal definitions (continued)

Signal name	Type	Reset value	Definition
j_trst_b	I		JTAG test reset from pad
j_tclk	I		JTAG test clock from pad
j_tms	I		JTAG test mode select from pad
j_tdi	I		JTAG test data input from pad
j_tdo	O	0	JTAG test data out to master controller or pad
j_tdo_en	O	0	Enables TDO output buffer
j_tst_log_rst	O	0	Indicates Test-Logic-Reset state of JTAG controller
j_capture_ir	O	0	Indicates Capture_IR state of JTAG controller
j_update_ir	O	0	Indicates Update_IR state of JTAG controller
j_shift_ir	O	0	Indicates Shift_IR state of JTAG controller
j_capture_dr	O	0	Indicates parallel test data register load state of JTAG controller
j_shift_dr	O	0	Indicates the TAP controller is in shift DR state
j_update_gp_reg	O	0	Updates JTAG controller test data register
j_rti	O	0	JTAG controller run-test-idle state
j_key_in	I		Input for providing data to be shifted out during Shift_IR state when jd_en_once is negated
j_en_once_regsel	O	0	external Enable Once register select
j_nexus_regsel	O	0	external Nexus register select
j_lsrl_regsel	O	0	external LSRL register select
j_gp_regsel[0:9]	O	0	General-purpose external JTAG register select
j_id_sequence[0:1]	I		JTAG ID Register (2 MSBs of sequence field)
j_id_version[0:3]	I		JTAG ID Register Version Field
j_serial_data	I		Serial data from external JTAG registers
Test primary input/output signals			
Test Control Interface[1]			Test Mode determination
Scan Test Interface[1]			Scan Configuration and Testing
Memory BIST Interface[1]			Memory BIST Configuration and Testing

[1] Please refer to the e200z759n3 Test Guide for information on the Test signals.

14.2 Signal descriptions

The following paragraphs provide descriptions of the signals.

14.2.1 e200z759n3 processor clock (m_clk)

The **m_clk** input is the synchronous clock source for the e200z759n3 processor core.

Since e200z759n3 is designed for static operation, **m_clk** can be gated off to lower power dissipation (e.g., during low-power stopped states).

14.2.2 Reset-related signals

Zen supports several reset input signals for the CPU and JTAG/OnCE control logic: **m_por**, **p_reset_b**, and **j_trst_b**. The reset domains have been partitioned such that the CPU **p_reset_b** signal does not affect JTAG/OnCE logic and **j_trst_b** does not affect processor logic. It is possible and desirable to access OnCE registers while the processor is running or in reset. Alternatively, it is also possible and desirable to assert **j_trst_b** and clear the JTAG/OnCE logic without affecting the state of the processor.

The synchronization logic between the processor and debug module requires an assertion of either **j_trst_b** or **m_por** during initial processor power-up reset in order to ensure proper operation. If the pin associated with the **j_trst_b** input is designed with a pull-up resistor and left floating, then assertion of **m_por** is required during the initial power-on processor reset. Similarly, for those systems 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. Once a power-up reset has been achieved, the two resets can be asserted independently.

The watchdog reset status output signals **p_wrs[0:1]** are also provided, which can be conditionally asserted by watchdog time-outs, and the debug reset control outputs **p_dbrstc[0:1]** can be asserted by debug control settings in DBCR0.

A set of input signals (**p_rstbase[0:29], p_rst_endmode, p_rst_vlemode**) are provided to relocate the reset exception handler to allow for flexible placement of boot code, and to select the default endian mode and VLE mode of the CPU out of reset.

These signals are described in detail in the following sub-sections.

14.2.2.1 Power-on reset (m_por)

The **m_por** signal is the power-on reset input for the e200z759n3 processor. This signal serves the following purposes:

1. **m_por** is "ORed" with the **j_trst_b** function and the resulting signal clears the JTAG TAP controller and associated registers as well as the OnCE state machine. This is an asynchronous clear with a short assertion time requirement.
2. **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.

14.2.2.2 Reset (p_reset_b)

The **p_reset_b** input is the active-low reset input for the e200z759n3 processor. **p_reset_b** is treated as an asynchronous input and is sampled by the clock control logic in the e200z759n3 debug module.

14.2.2.3 Watchdog reset status (p_wrs[0:1])

The **p_wrs[0:1]** outputs are active-high reset output status signals from the e200z759n3 core that reflect the value of the $\mathrm{TSR}_{\mathrm{WRS}}$ status field. **p_wrs[0:1]** are conditionally asserted by the Watchdog Timer (Section 2.4.8, Timer Control Register (TCR), and Section 2.4.9, Timer Status Register (TSR)).

14.2.2.4 Debug reset control (p_dbrstc[0:1])

The **p_dbrstc[0:1]** outputs are active-high reset output control signals from the e200z759n3 core that reflect the value of the $\mathrm{DBCR0}_{\mathrm{RST}}$ status field. **p_dbrstc[0:1]** are conditionally asserted by the Debug control logic (Section 12.3.3.1, Debug Control Register 0 (DBCR0)).

14.2.2.5 Reset base (p_rstbase[0:29])

The **p_rstbase[0:29]** inputs are provided to allow system integrators to be able to specify/relocate the base address of the reset exception handler. These inputs are used to form the upper 30 bits of the instruction access following negation of reset, which is used to fetch the initial instruction of the reset exception handler. These bits should be driven to a value corresponding to the desired boot memory device in the system. These inputs must remain stable in a window beginning two clocks prior to the negation of reset and extending into the cycle in which the reset vector fetch is initiated. These inputs are also used by the MMU during reset to form a default TLB entry 0 for translation of the reset vector fetch.

The initial instruction fetch will occur to the location [**p_rstbase[0:29]**] || 2’b00.

14.2.2.6 Reset endian mode (p_rst_endmode)

The **p_rst_endmode** input is used by the MMU during reset to form the ‘E’ bit of the default TLB entry 0 for translation of the reset vector fetch. A low logic level on this signal will cause the resultant entry ‘E’ bit to set to ‘0’, indicating a big-endian page. A high logic level on this signal will cause the resultant entry ‘E’ bit to set to ‘1’, indicating a little-endian page.

14.2.2.7 Reset VLE Mode (p_rst_vlemode)

The **p_rst_vlemode** input is used by the MMU during reset to form the ‘VLE’ bit of the default TLB entry 0 for translation of the reset vector fetch. A low logic level on this signal will cause the resultant entry ‘VLE’ bit to set to ‘0’, indicating a BookE page. A high logic level on this signal will cause the resultant entry ‘VLE’ bit to set to ‘1’, indicating a VLE page.

14.2.2.8 JTAG/OnCE reset (j_trst_b)

The **j_trst_b** signal (referred to in the *IEEE 1149.1 JTAG Specification* as the TRST* signal) is an asynchronous reset with a short assertion time requirement. It is “ORed” with the **m_por** function and the resulting signal clears the OnCE TAP controller and associated registers as well as the OnCE state machine.

14.2.3 Address and data buses

Dual instruction and data interfaces are provided by the e200z759n3. They are described together, with appropriate differences denoted.

14.2.3.1 Address bus (p_d_haddr[31:0], p_i_haddr[31:0])

These outputs provide the address for a bus transfer. Per the AHB definition, **p_[d,i]_haddr[31]** is the MSB and **p_[d,i]_haddr[0]** is the LSB.

14.2.3.2 Read data bus (p_d_hrdata[63:0], p_i_hrdata[63:0])

These inputs provide data to the e200z759n3 on read transfers. The read data bus can transfer 8, 16, 24, 32, or 64 bits of data per bus transfer. Instruction transfers will not use the 8-bit and 24-bit capability. Per the AHB definition, **p_[d,i]_hrdata[63]** is the MSB and **p_hrdata[0]** is the LSB. Table 14-2 shows the relationship of byte addresses to read data bus signals.

Table 14-2. p_hrdata[63:0] byte address mappings

Memory byte address	Wired to p_[d,i]_hrdata bits
000	7:0
001	15:8
010	23:16
011	31:24
100	39:32
101	47:40
110	55:48
111	63:56

14.2.3.3 Write data bus (p_d_hwdata[63:0])

These outputs transfer data from the e200z759n3 on write transfers. The write data bus can transfer 8, 16, 24, 32, or 64 bits of data per bus transfer. Per the AHB definition, **p_d_hwdata[63]** is the MSB and **p_d_hwdata[0]** is the LSB. Figure 14-3 shows the relationship of byte addresses to write data bus signals.

Table 14-3. p_d_hwdata[63:0] byte address mappings

Memory byte address	Wired to p_d_hwdata bits
000	7:0
001	15:8
010	23:16
011	31:24
100	39:32
101	47:40

Table 14-3. p_d_hwdata[63:0] byte address mappings

Memory byte address	Wired to p_d_hwdata bits
110	55:48
111	63:56

14.2.4 Transfer attribute signals

The following paragraphs describe the transfer attribute signals, which provide additional information about the bus transfer cycle. Transfer attributes are driven with address at the beginning of a bus transfer.

14.2.4.1 Transfer type (p_d_htrans[1:0], p_i_htrans[1:0])

The processor drives these signals to indicate the current transfer type. Table 14-4 shows **p_[d,i]_htrans[1:0]** encoding.

Table 14-4. p_[d,i]_htrans[1:0] transfer type encoding

p_[d,i]_htrans[1]	p_[d,i]_htrans[0]	Access type
0	0	IDLE - No data transfer is required.
0	1	BUSY - Master is busy, burst transfer continues. (encoding not used by Zen Z7).
1	0	NONSEQ - indicates the first transfer of a burst, or a single transfer. Address and control signals are unrelated to the previous transfer.
1	1	SEQ - indicates the continuation of a burst. Address and control signals are related to the previous transfer. Control signals are the same, Address has been incremented by the size of the data transferred (optionally wrapped).

If the **p_[d,i]_htrans[1:0]** encoding is not IDLE or BUSY, a transfer is being requested. e200z759n3 does not utilize the BUSY encoding, and will not present this type of transfer to a bus slave. Slaves must terminate IDLE transfers with a zero wait-state OKAY response and ignore the (non-existent) transfer.

14.2.4.2 Write (p_d_hwrite, p_i_hwrite)

This output signal defines the data transfer direction for the current bus cycle. A high (logic one) level indicates a write cycle, and a low (logic zero) level indicates a read cycle. For **p_i_hwrite**, the signal is internally driven low for all instruction AHB transfers.

14.2.4.3 Transfer size (p_d_hsize[1:0], p_i_hsize[1:0])

The **p_[d,i]_hsize[1:0]** signals indicate the data size for a bus transfer. Table 14-5 shows the definitions of the **p_[d,i]_hsize[1:0]** encodings. For misaligned transfers, the transfer size may indicate a size larger than the requested size to ensure that all asserted byte strobes are contained within the “container” defined by **p_[d,i]_hsize[1:0]**. Refer to Table 14-11 and Table 14-12 for **p_[d,i]_hsize[1:0]** encodings used for aligned and misaligned transfers.

Table 14-5. p_[d,i]_hsize[1:0] transfer size encoding

p_[d,i]_hsize[1:0]	Transfer size
00	Byte
01	Halfword (2 bytes)
10	Word (4 bytes)
11	Doubleword (8 bytes)

14.2.4.4 Burst type (p_d_hburst[2:0], p_i_hburst[2:0])

The **p_[d,i]_hburst[2:0]** signals indicate the burst type for a bus transfer. Table 14-6 shows the definitions of the **p_[d,i]_hburst[2:0]** encodings.

Table 14-6. p_[d,i]_hburst[2:0] burst type encoding

p_hburst[2:0]	Burst type
000	SINGLE — No burst, single beat only
001	INCR — Incrementing burst of unspecified length — Unused
010	WRAP4 — 4-beat wrapping burst
011	INCR4 — 4-beat incrementing burst — Unused
100	WRAP8 — 8-beat wrapping burst — Unused
101	INCR8 — 8-beat incrementing burst — Unused
110	WRAP16 — 16-beat wrapping burst — Unused
111	INCR16— 16-beat incrementing burst — Unused

The e200z759n3 will only utilize SINGLE and WRAP4 burst types. In addition, all WRAP4 bursts are of doubleword size aligned to doubleword boundaries.

14.2.4.5 Protection control (p_d_hprot[5:0], p_i_hprot[5:0])

The e200z759n3 drives the **p_[d,i]_hprot[5:0]** signals to indicate the type of access for the current bus cycle. **p_[d,i]_hprot[0]** indicates instruction/data, **p_[d,i]_hprot[1]** indicates user/supervisor. **p_[d,i]_hprot[5]** indicates whether the access is Exclusive (i.e. for a ***lbarx, lharx, lwarx, stbcx., sthcx.,*** or **stwcx.** instruction). **p_[d,i]_hprot[4:2]** (Allocate, Cacheable, Bufferable) are used to indicate particular cache attributes for the access and are driven to default values based on settings in the memory management unit.

Table 14-7 shows the definitions of the **p_d_hprot[5:0]** signals.

Table 14-7. p_d_hprot[5:0] protection control encoding

p_hprot[5]	p_hprot[4]	p_hprot[3]	p_hprot[2]	p_hprot[1]	p_hprot[0]	Transfer type
—	—	—	—	0	1	User mode access
—	—	—	—	1	1	Supervisor mode access
—	0	0	0	—	1	Cache-inhibited

Table 14-7. p_d_hprot[5:0] protection control encoding (continued)

p_hprot[5]	p_hprot[4]	p_hprot[3]	p_hprot[2]	p_hprot[1]	p_hprot[0]	Transfer type
—	0	0	1	—	1	Guarded, not cache-inhibited
—	0	1	0	—	1	Reserved
—	0	1	1	—	1	Reserved
—	1	0	0	—	1	Reserved
—	1	0	1	—	1	Reserved
—	1	1	0	—	1	Cacheable, writethrough
—	1	1	1	—	1	Cacheable, writeback
0	—	—	—	—	1	Not exclusive
1	—	—	—	—	1	Exclusive access

Table 14-8 shows the definitions of the **p_i_hprot[5:0]** signals.

Table 14-8. p_i_hprot[5:0] protection control encoding

p_hprot[5]	p_hprot[4]	p_hprot[3]	p_hprot[2]	p_hprot[1]	p_hprot[0]	Transfer type
0	—	—	—	0	0	User mode access
0	—	—	—	1	0	Supervisor mode access
0	0	0	0	—	0	Cache-inhibited
0	0	0	1	—	0	Reserved
0	0	1	0	—	0	Reserved
0	0	1	1	—	0	Reserved
0	1	0	0	—	0	Reserved
0	1	0	1	—	0	Reserved
0	1	1	0	—	0	Cacheable
0	1	1	1	—	0	Reserved

Note that all signals are provided on both I and D ports, although they will not all change state. (ex. p_d_hprot0 is always high, etc.).

The e200z759n3 maps the *PowerISA 2.06* storage attributes to the AHB data port **hprot** signals in the manner described in Table 14-9:

Table 14-9. Mapping of access attributes to p_d_hprot[4:2] protection control

[I]	[G]	[W]	p_hprot[4]	p_hprot[3]	p_hprot[2]	Transfer type
0	0	0	1	1	1	Cacheable, writeback
0	0	1	1	1	0	Cacheable, writethrough
0	1	—	0	0	1	Guarded, not cache-inhibited

Table 14-9. Mapping of access attributes to p_d_hprot[4:2] protection control (continued)

[I]	[G]	[W]	p_hprot[4]	p_hprot[3]	p_hprot[2]	Transfer type
1	—	—	0	0	0	Cache-inhibited
—	—	—	0	0	1	Buffered store, page marked guarded
—	—	—	1	1	0	Buffered store and page marked writethrough, and non-guarded
—	—	—	1	1	1	Buffered store and page marked copyback, and non-guarded

For buffered stores, **p_d_hprot[1]** is driven with the user/supervisor mode attribute associated with the store at the time it was buffered.

14.2.4.6 Transfer data error (p_d_htrans_derr)

The **p_d_htrans_derr** control signal is driven during bus transfers on the data interface to indicate a data cache data array parity error has occurred for a cache push (copyback) operation, and the data corresponding to this address is not valid due to a parity error. This signal is driven valid with address and attribute timing. System logic may monitor this output and perform any desired recovery activity. This signal will only be asserted during a copyback operation for those beats for which the corresponding data has a parity error.

14.2.4.7 Globally coherent access — (p_d_gbl)

The **p_d_gbl** control signal is driven during bus transfers on the data interface to indicate whether the memory access is marked by the MMU 'M' page attribute as globally coherent. This signal is driven valid with address and attribute timing, and remain valid for all beats of a burst access. This signal reflects the value of the "M" (memory coherence required) attribute for the page associated with the access, except for dirty line pushes to memory. For those accesses, it is negated.

14.2.4.8 Cache way replacement (p_d_wayrep[0:1], p_i_wayrep[0:1])

The **p_[d,i]_wayrep[0:1]** control signals are driven valid during cache line fills to indicate which way of the cache is being replaced. These signals are driven valid with address and attribute timing, and remain valid for all beats of the burst read. These signals are undefined on all other transfer types.

14.2.5 Byte lane specification

Read transactions transfer from 1 to 8 bytes of data on the **p_[d,i]_hrdata[63:0]** bus. The byte lanes involved in the transfer are determined by the starting byte number specified by the lower address bits in conjunction with the transfer size and byte strobes. Addressing of the byte lanes is shown big-endian (left to right) regardless of the endian mode of the e200z759n3 core. The byte of memory corresponding to address 0 is connected to B0 (**p_[d,i]_h{r,w}data[7:0]**) and the byte of memory corresponding to address 7 is connected to B7 (**p_[d,i]_h{r,w}data[63:56]**). The CPU internally permutes read data as required for the endian mode of the current access. Misaligned transfers are indicated with the **p_[d,i]_hunalign** signal to indicate that byte strobes do not correspond exactly to size and low-order address bits.

14.2.5.1 Unaligned access (p_d_hunalign, p_i_hunalign)

The **p_[d,i]_hunalign** output signal indicates that the current access is a misaligned access. This signal is asserted for misaligned data accesses, and for misaligned instruction accesses from VLE pages. Normal BookE instruction pages are always aligned. The timing of this signal is approximately the same as address timing. When **p_[d,i]_hunalign** is asserted, the **p_[d,i]_hbstrb[7:0]** byte strobe signals will indicate the selected bytes involved in the current portion of the misaligned access, which may not include all bytes defined by the size and low-order address signals. Aligned transfers also assert the byte strobes, but in a manner corresponding to size and low order address bits.

14.2.5.2 Byte strobes (p_d_hbstrb[7:0], p_i_hbstrb[7:0])

The **p_[d,i]_hbstrb[7:0]** byte strobe signals indicate the selected bytes involved in the current transfer. For a misaligned access, the current transfer may not include all bytes defined by the size and low-order address signals. For aligned transfers, the byte strobe signals will correspond to the bytes defined by the size and low-order address signals. Table 14-3 shows the relationship of byte addresses to the byte strobe signals.

Table 14-10. p_[d,i]_hbstrb[7:0] to byte address mappings

Memory byte address	Wired to p_h{r,w}data bits	Corresponding byte strobe signal
000	7:0	p_[d,i]_hbstrb[0]
001	15:8	p_[d,i]_hbstrb[1]
010	23:16	p_[d,i]_hbstrb[2]
011	31:24	p_[d,i]_hbstrb[3]
100	39:32	p_[d,i]_hbstrb[4]
101	47:40	p_[d,i]_hbstrb[5]
110	55:48	p_[d,i]_hbstrb[6]
111	63:56	p_[d,i]_hbstrb[7]

Table 14-11 lists all of the data transfer permutations. Note that misaligned data requests which cross a 64-bit boundary are broken up into two separate bus transactions, and the address value and the size encoding for the first transfer is not modified. The table is arranged in a big-endian fashion, but the active lanes are the same regardless of the endian-mode of the access. The e200z759n3 performs the proper byte routing internally based on endianness.

Table 14-11. Byte strobe assertion for transfers

Program size and byte offset	A(2:0)	HSIZE[1:0]	Data bus byte strobes								HUNALIGN
			B0	B1	B2	B3	B4	B5	B6	B7	
Byte @000	0 0 0	0 0	X	—	—	—	—	—	—	—	0
Byte @001	0 0 1	0 0	—	X	—	—	—	—	—	—	0
Byte @010	0 1 0	0 0	—	—	X	—	—	—	—	—	0
Byte @011	0 1 1	0 0	—	—	—	X	—	—	—	—	0
Byte @100	1 0 0	0 0	—	—	—	—	X	—	—	—	0
Byte @101	1 0 1	0 0	—	—	—	—	—	X	—	—	0
Byte @110	1 1 0	0 0	—	—	—	—	—	—	X	—	0
Byte @111	1 1 1	0 0	—	—	—	—	—	—	—	X	0
Half @000	0 0 0	0 1	X	X	—	—	—	—	—	—	0
Half @001	0 0 1	1 0#	—	X	X	—	—	—	—	—	1
Half @010	0 1 0	0 1	—	—	X	X	—	—	—	—	0
Half @011	0 1 1	1 1#	—	—	—	X	X	—	—	—	1
Half @100	1 0 0	0 1	—	—	—	—	X	X	—	—	0
Half @101	1 0 1	1 0#	—	—	—	—	—	X	X	—	1
Half @110	1 1 0	0 1	—	—	—	—	—	—	X	X	0
Half @111 (2 bus transfers)	1 1 1 0 0 0	0 1* 0 0	— X	— —	— —	— —	— —	— —	— —	X —	1 0
Word @000	0 0 0	1 0	X	X	X	X	—	—	—	—	0
Word @001	0 0 1	1 1#	—	X	X	X	X	—	—	—	1
Word @010	0 1 0	1 1#	—	—	X	X	X	X	—	—	1
Word @011	0 1 1	1 1#	—	—	—	X	X	X	X	—	1
Word @100	1 0 0	1 0	—	—	—	—	X	X	X	X	0
Word @101 (2 bus transfers)	1 0 1 0 0 0	1 0* 0 0	— X	— —	— —	— —	— —	X —	X —	X —	1 0
Word @110 (2 bus transfers)	1 1 0 0 0 0	1 0* 0 1	— X	— X	— —	— —	— —	— —	X —	X —	1 0
Word @111 (2 bus transfers)	1 1 1 0 0 0	10* 1 0	— X	— X	— X	— —	— —	— —	— —	X —	1 1
Doubleword @000	0 0 0	1 1	X	X	X	X	X	X	X	X	0
Doubleword @001 (2 bus transfers)	0 0 1 0 0 0	1 1* 0 0	— X	X —	X —	X —	X —	X —	X —	X —	1 0
Doubleword @010 (2 bus transfers)	0 1 0 0 0 0	1 1* 0 1	— X	— X	X —	X —	X —	X —	X —	X —	1 0

Table 14-11. Byte strobe assertion for transfers (continued)

Program size and byte offset	A(2:0)	HSIZE[1:0]	Data bus byte strobes								HUNALIGN
			B0	B1	B2	B3	B4	B5	B6	B7	
Doubleword @011 (2 bus transfers)	0 1 1 0 0 0	1 1* 1 0#	— X	— X	— X	X —	X —	X —	X —	X —	1 1
Doubleword @100 (2 bus transfers)	1 0 0 0 0 0	1 1* 1 0	— X	— X	— X	— X	X —	X —	X —	X —	1 0
Doubleword @101 (2 bus transfers)	1 0 1 0 0 0	1 1* 1 1#	— X	— X	— X	— X	— X	X —	X —	X —	1 1
Doubleword @110 (2 bus transfers)	1 1 0 0 0 0	1 1* 1 1#	— X	— X	— X	— X	— X	— X	X —	X —	1 1
Doubleword @111 (2 bus transfers)	1 1 1 0 0 0	1 1* 1 1#	— X	— X	— X	— X	— X	— X	— X	X —	1 1

Table Notes:
"X" indicates byte lanes involved in the transfer; Other lanes will contain driven but unused data.
# These misaligned transfers drive size according to the size of the power of two aligned "container" in which the byte strobes are asserted.
* These misaligned cases drive request size according to the size specified by the load or store instruction.

Table 14-12 shows the final layout in memory for data transferred from a 64-bit GPR containing the bytes 'A B C D E F G H' to memory. Misaligned accesses that cross a doubleword boundary are broken into a pair of accesses by the CPU.

Table 14-12. Big- and little-endian memory storage

Program size and byte offset	A(3:0)	HSIZE(1:0)	Even Double Word — 0								0dd Double Word — 1							
			B0	B1	B2	B3	B4	B5	B6	B7	B0	B1	B2	B3	B4	B5	B6	B7
Byte @0000	0 0 0 0	0 0	H	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Byte @0001	0 0 0 1	0 0	—	H	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Byte @0010	0 0 1 0	0 0	—	—	H	—	—	—	—	—	—	—	—	—	—	—	—	—
Byte @0011	0 0 1 1	0 0	—	—	—	H	—	—	—	—	—	—	—	—	—	—	—	—
Byte @0100	0 1 0 0	0 0	—	—	—	—	H	—	—	—	—	—	—	—	—	—	—	—
Byte @0101	0 1 0 1	0 0	—	—	—	—	—	H	—	—	—	—	—	—	—	—	—	—
Byte @0110	0 1 1 0	0 0	—	—	—	—	—	—	H	—	—	—	—	—	—	—	—	—
Byte @0111	0 1 1 1	0 0	—	—	—	—	—	—	—	H	—	—	—	—	—	—	—	—
Byte @1000	1 0 0 0	0 0	—	—	—	—	—	—	—	—	H	—	—	—	—	—	—	—
Byte @1001	1 0 0 1	0 0	—	—	—	—	—	—	—	—	—	H	—	—	—	—	—	—
Byte @1010	1 0 1 0	0 0	—	—	—	—	—	—	—	—	—	—	H	—	—	—	—	—
Byte @1011	1 0 1 1	0 0	—	—	—	—	—	—	—	—	—	—	—	H	—	—	—	—

Table 14-12. Big- and little-endian memory storage (continued)

Program size and byte offset	A(3:0)	HSIZE(1:0)	Even Double Word — 0								0dd Double Word — 1							
			B0	B1	B2	B3	B4	B5	B6	B7	B0	B1	B2	B3	B4	B5	B6	B7
Byte @1100	1 1 0 0	0 0	—	—	—	—	—	—	—	—	—	—	—	—	H	—	—	—
Byte @1101	1 1 0 1	0 0	—	—	—	—	—	—	—	—	—	—	—	—	—	H	—	—
Byte @1110	1 1 1 0	0 0	—	—	—	—	—	—	—	—	—	—	—	—	—	—	H	—
Byte @1111	1 1 1 1	0 0	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	H
B. E. Half @0000	0 0 0 0	0 1	G	H	—	—	—	—	—	—	—	—	—	—	—	—	—	—
B. E. Half @0001	0 0 0 1	1 0[#]	—	G	H	—	—	—	—	—	—	—	—	—	—	—	—	—
B. E. Half @0010	0 0 1 0	0 1	—	—	G	H	—	—	—	—	—	—	—	—	—	—	—	—
B. E. Half @0011	0 0 1 1	1 1[#]	—	—	—	G	H	—	—	—	—	—	—	—	—	—	—	—
B. E. Half @0100	0 1 0 0	0 1	—	—	—	—	G	H	—	—	—	—	—	—	—	—	—	—
B. E. Half @0101	0 1 0 1	1 0[#]	—	—	—	—	—	G	H	—	—	—	—	—	—	—	—	—
B. E. Half @0110	0 1 1 0	0 1	—	—	—	—	—	—	G	H	—	—	—	—	—	—	—	—
B. E. Half @0111	0 1 1 1	0 1	—	—	—	—	—	—	—	G	—	—	—	—	—	—	—	—
	1 0 0 0	0 0	—	—	—	—	—	—	—	—	H	—	—	—	—	—	—	—
B. E. Half @1000	1 0 0 0	0 1	—	—	—	—	—	—	—	—	G	H	—	—	—	—	—	—
B. E. Half @1001	1 0 0 1	1 0[#]	—	—	—	—	—	—	—	—	—	G	H	—	—	—	—	—
B. E. Half @1010	1 0 1 0	0 1	—	—	—	—	—	—	—	—	—	—	G	H	—	—	—	—
B. E. Half @1011	1 0 1 1	1 1[#]	—	—	—	—	—	—	—	—	—	—	—	G	H	—	—	—
B. E. Half @1100	1 1 0 0	0 1	—	—	—	—	—	—	—	—	—	—	—	—	G	H	—	—
B. E. Half @1101	1 1 0 1	1 0[#]	—	—	—	—	—	—	—	—	—	—	—	—	—	G	H	—
B. E. Half @1110	1 1 1 0	0 1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	G	H
B. E. Half @1111	1 1 1 1	0 1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	G
	0 0 0 0 (next dword)	0 0	H	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—

Table 14-12. Big- and little-endian memory storage (continued)

Program size and byte offset	A(3:0)	HSIZE(1:0)	Even Double Word — 0								0dd Double Word — 1							
			B0	B1	B2	B3	B4	B5	B6	B7	B0	B1	B2	B3	B4	B5	B6	B7
L E. Half @0000	0 0 0 0	0 1	H	G	—	—	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Half @0001	0 0 0 1	1 0#	—	H	G	—	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Half @0010	0 0 1 0	0 1	—	—	H	G	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Half @0011	0 0 1 1	1 1#	—	—	—	H	G	—	—	—	—	—	—	—	—	—	—	—
L. E. Half @0100	0 1 0 0	0 1	—	—	—	—	H	G	—	—	—	—	—	—	—	—	—	—
L. E. Half @0101	0 1 0 1	1 0#	—	—	—	—	—	H	G	—	—	—	—	—	—	—	—	—
L. E. Half @0110	0 1 1 0	0 1	—	—	—	—	—	—	H	G	—	—	—	—	—	—	—	—
L. E. Half @0111	0 1 1 1	0 1	—	—	—	—	—	—	—	H	—	—	—	—	—	—	—	—
	1 0 0 0	0 0	—	—	—	—	—	—	—	—	G	—	—	—	—	—	—	—
L. E. Half @1000	1 0 0 0	0 1	—	—	—	—	—	—	—	—	H	G	—	—	—	—	—	—
L. E. Half @1001	1 0 0 1	1 0#	—	—	—	—	—	—	—	—	—	H	G	—	—	—	—	—
L. E. Half @1010	1 0 1 0	0 1	—	—	—	—	—	—	—	—	—	—	H	G	—	—	—	—
L. E. Half @1011	1 0 1 1	1 1#	—	—	—	—	—	—	—	—	—	—	—	H	G	—	—	—
L. E. Half @1100	1 1 0 0	0 1	—	—	—	—	—	—	—	—	—	—	—	—	H	G	—	—
L. E. Half @1101	1 1 0 1	1 0#	—	—	—	—	—	—	—	—	—	—	—	—	—	H	G	—
L. E. Half @1110	1 1 1 0	0 1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	H	G
L. E. Half @1111	1 1 1 1	0 1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	H
	+0 0 0 0 (next dword)	0 0	G	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
B. E. Word @0000	0 0 0 0	1 0	E	F	G	H	—	—	—	—	—	—	—	—	—	—	—	—
B. E. Word @0001	0 0 0 1	1 1#	—	E	F	G	H	—	—	—	—	—	—	—	—	—	—	—

Table 14-12. Big- and little-endian memory storage (continued)

Program size and byte offset	A(3:0)	HSIZE(1:0)	Even Double Word — 0								0dd Double Word — 1							
			B0	B1	B2	B3	B4	B5	B6	B7	B0	B1	B2	B3	B4	B5	B6	B7
B. E. Word @0010	0 0 1 0	1 1#	—	—	E	F	G	H	—	—	—	—	—	—	—	—	—	—
B. E. Word @0011	0 0 1 1	1 1#	—	—	—	E	F	G	H	—	—	—	—	—	—	—	—	—
B. E. Word @0100	0 1 0 0	0 1 0	—	—	—	—	E	F	G	H	—	—	—	—	—	—	—	—
B. E. Word @0101	0 1 0 1	1 0	—	—	—	—	—	E	F	G	—	—	—	—	—	—	—	—
	1 0 0 0	0 0	—	—	—	—	—	—	—	—	H	—	—	—	—	—	—	—
B. E. Word @0110	0 1 1 0	1 0	—	—	—	—	—	—	E	F	—	—	—	—	—	—	—	—
	1 0 0 0	0 1	—	—	—	—	—	—	—	—	G	H	—	—	—	—	—	—
B. E. Word @0111	0 1 1 1	1 0	—	—	—	—	—	—	—	E	—	—	—	—	—	—	—	—
	1 0 0 0	1 0	—	—	—	—	—	—	—	—	F	G	H	—	—	—	—	—
B. E. Word @1000	1 0 0 0	1 0	—	—	—	—	—	—	—	—	E	F	G	H	—	—	—	—
B. E. Word @1001	1 0 0 1	1 1#	—	—	—	—	—	—	—	—	—	E	F	G	H	—	—	—
B. E. Word @1010	1 0 1 0	1 1#	—	—	—	—	—	—	—	—	—	—	E	F	G	H	—	—
B. E. Word @1011	1 0 1 1	1 1#	—	—	—	—	—	—	—	—	—	—	—	E	F	G	H	—
B. E. Word @1100	1 1 0 0	1 0	—	—	—	—	—	—	—	—	—	—	—	—	E	F	G	H
B. E. Word @1101	1 1 0 1	1 0	—	—	—	—	—	—	—	—	—	—	—	—	—	E	F	G
	+0 0 0 0 (next dword)	0 0	H	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
B. E. Word @1110	1 1 1 0	1 0	—	—	—	—	—	—	—	—	—	—	—	—	—	—	E	F
	+0 0 0 0 (next dword)	0 1	G	H	—	—	—	—	—	—	—	—	—	—	—	—	—	—
B. E. Word @1111	1 1 1 1	1 0	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	E
	+0 0 0 0 (next dword)	1 0	F	G	H	—	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Word @0000	0 0 0 0	1 0	H	G	F	E	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Word @0001	0 0 0 1	1 1#	—	H	G	F	E	—	—	—	—	—	—	—	—	—	—	—

Table 14-12. Big- and little-endian memory storage (continued)

Program size and byte offset	A(3:0)	HSIZE(1:0)	Even Double Word — 0								0dd Double Word — 1							
			B0	B1	B2	B3	B4	B5	B6	B7	B0	B1	B2	B3	B4	B5	B6	B7
L. E. Word @0010	0 0 1 0	1 1#	—	—	H	G	F	E	—	—	—	—	—	—	—	—	—	—
L. E. Word @0011	0 0 1 1	1 1#	—	—	—	H	G	F	E	—	—	—	—	—	—	—	—	—
L. E. Word @0100	0 1 0 0	1 0	—	—	—	—	H	G	F	E	—	—	—	—	—	—	—	—
L. E. Word @0101	0 1 0 1	1 0	—	—	—	—	—	H	G	F	—	—	—	—	—	—	—	—
	1 0 0 0	0 0	—	—	—	—	—	—	—	—	E	—	—	—	—	—	—	—
L. E. Word @0110	0 1 1 0	1 0	—	—	—	—	—	—	H	G	—	—	—	—	—	—	—	—
	1 0 0 0	0 1	—	—	—	—	—	—	—	—	F	E	—	—	—	—	—	—
L. E. Word @0111	0 1 1 1	1 0	—	—	—	—	—	—	—	H	—	—	—	—	—	—	—	—
	1 0 0 0	1 0	—	—	—	—	—	—	—	—	G	F	E	—	—	—	—	—
L. E. Word @1000	1 0 0 0	1 0	—	—	—	—	—	—	—	—	H	G	F	E	—	—	—	—
L. E. Word @1001	1 0 0 1	1 1#	—	—	—	—	—	—	—	—	—	H	G	F	E	—	—	—
L. E. Word @1010	1 0 1 0	1 1#	—	—	—	—	—	—	—	—	—	—	H	G	F	E	—	—
L. E. Word @1011	1 0 1 1	1 1#	—	—	—	—	—	—	—	—	—	—	—	H	G	F	E	—
L. E. Word @1100	1 1 0 0	1 0	—	—	—	—	—	—	—	—	—	—	—	—	H	G	F	E
L. E. Word @1101	1 1 0 1	1 0	—	—	—	—	—	—	—	—	—	—	—	—	—	H	G	F
	+0 0 0 0 (next dword)	0 0	E	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Word @1110	1 1 1 0	1 0	—	—	—	—	—	—	—	—	—	—	—	—	—	—	H	G
	+0 0 0 0 (next dword)	0 1	F	E	—	—	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Word @1111	1 1 1 1	1 0	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	H
	+0 0 0 0 (next dword)	1 0	G	F	E	—	—	—	—	—	—	—	—	—	—	—	—	—
B.E. Doubleword @0000	0 0 0 0	1 1	A	B	C	D	E	F	G	H	—	—	—	—	—	—	—	—

Table 14-12. Big- and little-endian memory storage (continued)

Program size and byte offset	A(3:0)	HSIZE(1:0)	Even Double Word — 0								0dd Double Word — 1							
			B0	B1	B2	B3	B4	B5	B6	B7	B0	B1	B2	B3	B4	B5	B6	B7
B. E. Doubleword @0001	0 0 0 1	1 1	—	A	B	C	D	E	F	G	—	—	—	—	—	—	—	—
	1 0 0 0 (next dword)	0 0	—	—	—	—	—	—	—	—	H	—	—	—	—	—	—	—
B. E. Doubleword @0010	0 0 1 0	1 1	—	—	A	B	C	D	E	F	—	—	—	—	—	—	—	—
	1 0 0 0 (next dword)	0 1	—	—	—	—	—	—	—	—	G	H	—	—	—	—	—	—
B. E. Doubleword @0011	0 0 1 1	1 1	—	—	—	A	B	C	D	E	—	—	—	—	—	—	—	—
	1 0 0 0 (next dword)	1 0#	—	—	—	—	—	—	—	—	F	G	H	—	—	—	—	—
B. E. Doubleword @0100	0 1 0 0	1 1	—	—	—	—	A	B	C	D	—	—	—	—	—	—	—	—
	1 0 0 0 (next dword)	1 0	—	—	—	—	—	—	—	—	E	F	G	H	—	—	—	—
B. E. Doubleword @0101	0 1 0 1	1 1	—	—	—	—	—	A	B	C	—	—	—	—	—	—	—	—
	1 0 0 0 (next dword)	1 1#	—	—	—	—	—	—	—	—	D	E	F	G	H	—	—	—
B. E. Doubleword @0110	0 1 1 0	1 1	—	—	—	—	—	—	A	B	—	—	—	—	—	—	—	—
	1 0 0 0 (next dword)	1 1#	—	—	—	—	—	—	—	—	C	D	E	F	G	H	—	—
B. E. Doubleword @0111	0 1 1 1	1 1	—	—	—	—	—	—	—	A	—	—	—	—	—	—	—	—
	1 0 0 0 (next dword)	1 1#	—	—	—	—	—	—	—	—	B	C	D	E	F	G	H	—
B.E. Doubleword @1000	1 0 0 0	1 1	—	—	—	—	—	—	—	—	A	B	C	D	E	F	G	H
B. E. Doubleword @1001	1 0 0 1	1 1	—	—	—	—	—	—	—	—	—	A	B	C	D	E	F	G
	+0 0 0 0 (next dword)	0 0	H	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
B. E. Doubleword @1010	1 0 1 0	1 1	—	—	—	—	—	—	—	—	—	—	A	B	C	D	E	F
	+ 0 0 0 0 (next dword)	0 1	G	H	—	—	—	—	—	—	—	—	—	—	—	—	—	—

Table 14-12. Big- and little-endian memory storage (continued)

Program size and byte offset	A(3:0)	HSIZE(1:0)	Even Double Word — 0								0dd Double Word — 1							
			B0	B1	B2	B3	B4	B5	B6	B7	B0	B1	B2	B3	B4	B5	B6	B7
B. E. Doubleword @1011	1 0 1 1	1 1	—	—	—	—	—	—	—	—	—	—	—	A	B	C	D	E
	+0 0 0 0 (next dword)	1 0#	F	G	H	—	—	—	—	—	—	—	—	—	—	—	—	—
B. E. Doubleword @1100	1 1 0 0	1 1	—	—	—	—	—	—	—	—	—	—	—	—	A	B	C	D
	+0 0 0 0 (next dword)	1 0	E	F	G	H	—	—	—	—	—	—	—	—	—	—	—	—
B. E. Doubleword @1101	1 1 0 1	1 1	—	—	—	—	—	—	—	—	—	—	—	—	—	A	B	C
	+0 0 0 0 (next dword)	1 1#	D	E	F	G	H	—	—	—	—	—	—	—	—	—	—	—
B. E. Doubleword @1110	1 1 1 0	1 1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	A	B
	+0 0 0 0 (next dword)	1 1#	C	D	E	F	G	H	—	—	—	—	—	—	—	—	—	—
B. E. Doubleword @1111	1 1 1 1	1 1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	A
	+0 0 0 0 (next dword)	1 1#	B	C	D	E	F	G	H	—	—	—	—	—	—	—	—	—
L.E. Doubleword @0000	0 0 0 0	1 1	H	G	F	E	D	C	B	A	—	—	—	—	—	—	—	—
L. E. Doubleword @0001	0 0 0 1	1 1	—	H	G	F	E	D	C	B	—	—	—	—	—	—	—	—
	1 0 0 0 (next dword)	0 0	—	—	—	—	—	—	—	—	A	—	—	—	—	—	—	—
L. E. Doubleword @0010	0 0 1 0	1 1	—	—	H	G	F	E	D	C	—	—	—	—	—	—	—	—
	1 0 0 0 (next dword)	0 1	—	—	—	—	—	—	—	—	B	A	—	—	—	—	—	—
L. E. Doubleword @0011	0 0 1 1	1 1	—	—	—	H	G	F	E	D	—	—	—	—	—	—	—	—
	1 0 0 0 (next dword)	1 0#	—	—	—	—	—	—	—	—	C	B	A	—	—	—	—	—
L. E. Doubleword @0100	0 1 0 0	1 1	—	—	—	—	H	G	F	E	—	—	—	—	—	—	—	—
	1 0 0 0 (next dword)	1 0	—	—	—	—	—	—	—	—	D	C	B	A	—	—	—	—

Table 14-12. Big- and little-endian memory storage (continued)

Program size and byte offset	A(3:0)	HSIZE(1:0)	Even Double Word — 0								0dd Double Word — 1							
			B0	B1	B2	B3	B4	B5	B6	B7	B0	B1	B2	B3	B4	B5	B6	B7
L. E. Doubleword @0101	0 1 0 1	1 1	—	—	—	—	—	H	G	F	—	—	—	—	—	—	—	—
	1 0 0 0 (next dword)	1 1#	—	—	—	—	—	—	—	—	E	D	C	B	A	—	—	—
L. E. Doubleword @0110	0 1 1 0	1 1	—	—	—	—	—	—	H	G	—	—	—	—	—	—	—	—
	1 0 0 0 (next dword)	1 1#	—	—	—	—	—	—	—	—	F	E	D	C	B	A	—	—
L. E. Doubleword @0111	0 1 1 1	1 1	—	—	—	—	—	—	—	H	—	—	—	—	—	—	—	—
	1 0 0 0 (next dword)	1 1#	—	—	—	—	—	—	—	—	G	F	E	D	C	B	A	—
L.E. Doubleword @1000	0 0 0 0	1 1	—	—	—	—	—	—	—	—	H	G	F	E	D	C	B	A
L. E. Doubleword @1001	1 0 0 1	1 1	—	—	—	—	—	—	—	—	—	H	G	F	E	D	C	B
	+0 0 0 0 (next dword)	0 0	A	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Doubleword @1010	1 0 1 0	1 1	—	—	—	—	—	—	—	—	—	—	H	G	F	E	D	C
	+0 0 0 0 (next dword)	0 1	B	A	—	—	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Doubleword @1011	1 0 1 1	1 1	—	—	—	—	—	—	—	—	—	—	—	H	G	F	E	D
	+0 0 0 0 (next dword)	1 0#	C	B	A	—	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Doubleword @1100	1 1 0 0	1 1	—	—	—	—	—	—	—	—	—	—	—	—	H	G	F	E
	+0 0 0 0 (next dword)	1 0	D	C	B	A	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Doubleword @1101	1 1 0 1	1 1	—	—	—	—	—	—	—	—	—	—	—	—	—	H	G	F
	+0 0 0 0 (next dword)	1 1#	E	D	C	B	A	—	—	—	—	—	—	—	—	—	—	—
L. E. Doubleword @1110	1 1 1 0	1 1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	H	G
	+0 0 0 0 (next dword)	1 1#	F	E	D	C	B	A	—	—	—	—	—	—	—	—	—	—

Table 14-12. Big- and little-endian memory storage (continued)

Program size and byte offset	A(3:0)	HSIZE(1:0)	Even Double Word — 0								0dd Double Word — 1							
			B0	B1	B2	B3	B4	B5	B6	B7	B0	B1	B2	B3	B4	B5	B6	B7
L. E. Doubleword @1111	1 1 1 1	1 1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	H
	+ 0 0 0 0 (next dword)	1 1#	G	F	E	D	C	B	A	—	—	—	—	—	—	—	—	—

Table Notes:
Assumes a 64-bit GPR contains ‘A B C D E F G H’
# These misaligned transfers drive size according to the size of the power of two aligned “container” in which the byte strobes are asserted.

14.2.6 Transfer control signals

The following paragraphs describe the transfer control signals.

14.2.6.1 Transfer ready (p_d_hready, p_i_hready)

The **p_[d,i]_hready** input signal indicates completion of a requested transfer operation. An external device asserts **p_[d,i]_hready** to terminate the transfer. The **p_[d,i]_hresp[2:0]** signals indicate status of the transfer.

14.2.6.2 Transfer response (p_d_hresp[2:0], p_i_hresp[1:0])

The **p_d_hresp[2:0]** and **p_i_hresp[1:0]** signals indicate status of a terminating transfer on the respective interfaces. Table 14-13 shows the definitions of the **p_d_hresp[2:0]** and **p_i_hresp[1:0]** encodings.

Table 14-13. p_d_hresp[2:0] transfer response encoding

p_d_hresp[2:0]	Response type
000	OKAY — transfer terminated normally
001	ERROR — transfer terminated abnormally
010	Reserved (RETRY not supported in AHB—Lite protocol)
011	Reserved (SPLIT not supported in AHB—Lite protocol)
100	XFAIL — Exclusive store failed (**stwcx.** did not completed successfully)
101	Reserved
110	Reserved
111	Reserved

Table 14-14. p_i_hresp[1:0] transfer response encoding

p_i_hresp[1:0]	Response type
00	OKAY — transfer terminated normally
01	ERROR — transfer terminated abnormally
10	Reserved (RETRY not supported in AHB-Lite protocol)
11	Reserved (SPLIT not supported in AHB-Lite protocol)

The ERROR and XFAIL responses are required to be two cycle responses. In this case, the ERROR or XFAIL responses must be signaled one cycle prior to assertion of **p_[d,i]_hready**, and must remain unchanged during the cycle **p_[d,i]_hready** is asserted.

The XFAIL response will be signaled to the CPU via the **p_d_xfail_b** internal signal. See Section 15.2.3.17, Store exclusive failure (p_d_xfail_b).

14.2.6.3 Bus stall global write request (p_stall_bus_gwrite)

The active-high **p_stall_bus_gwrite** signal is provided to request that new bus activity for global writes (writes with the “M” page attribute set) be stalled (postponed) for a period of time. When asserted, no new transfer requests will be generated for global writes following initiation and completion of all currently requested and outstanding accesses. This signal is provided to allow for control over global write access initiation to prevent overruns or overflows of external agents that observe or act upon bus transfers, but are not actually addressed slaves. One particular use of this throttling mechanism is to prevent overflow of the snoop (coherency) fifo in another CPU, or a trace fifo present in the system.

When asserted, no new global write transaction request will be generated, although a pending transaction (p_htrans != IDLE) awaiting completion of an outstanding transaction will still be taken and performed (unless an error response is received for the current outstanding transaction and the pending transaction is canceled).

14.2.7 AHB clock enable signals

The following paragraphs describe the AHB clock enable signals. These inputs are used to qualify the processor **m_clk** edges used for AHB output signal state updates and AHB input signal sampling for the memory interfaces. This allows for system AHB interfaces that run at sub-multiples of the **m_clk** frequency. These signals do not affect non-AHB interface signals.

14.2.7.1 Instruction AHB clock enable (p_i_ahb_clken)

The **p_i_ahb_clken** input signal is used to qualify the rising edges of m_clk on which the input signals **p_i_hready**, **p_i_hresp[1:0]** and **p_i_hrdata[63:0]** are sampled. (Note that by definition, **p_i_hrdata[63:0]** sampling is also qualified by the recognized assertion of **p_i_hready**, per the AHB protocol). When driven low, no sampling of these signals occurs, since **m_clk** is gated at the sampling logic. The **p_i_ahb_clken** input signal is also used to qualify the rising edges of **m_clk** on which the output signals **p_i_haddr[31:0]**, **p_i_hbstrb[7:0]**, **p_i_hburst[1:0]**, **p_i_hmaster[3:0]**, **p_i_hprot[5:0]**, **p_i_hsize[1:0]**, **p_i_htrans[1:0]**, and **p_i_hunalign** change state (by definition, in conjunction with the

p_i_hready input per the AHB protocol). The **p_i_ahb_clken** signal should normally be driven (change state) off the falling edge of **m_clk** to ensure the proper setup and hold times surrounding the **m_clk** high period. It must remain stable throughout the duration of **m_clk** high. This signal is ***not*** internally synchronized. It should be tied high when operating the data AHB at **m_clk** frequency. The integration guide defines the required setup time before **m_clk** rises and hold time after **m_clk** falls.

14.2.7.2 Data AHB clock enable (p_d_ahb_clken)

The **p_d_ahb_clken** input signal is used to qualify the rising edges of **m_clk** on which the input signals **p_d_hready**, **p_d_hresp[2:0]**, and **p_d_hrdata[63:0]** are sampled. (Note that by definition, **p_d_hrdata[63:0]** sampling is also qualified by the recognized assertion of **p_d_hready**, per the AHB protocol). When driven low, no sampling of these signals occurs, since **m_clk** is gated at the sampling logic. The **p_d_ahb_clken** input signal is also used to qualify the rising edges of **m_clk** on which the output signals **p_d_haddr[31:0]**, **p_d_hbstrb[7:0]**, **p_d_hburst[1:0]**, **p_d_hmaster[3:0]**, **p_d_hprot[5:0]**, **p_d_hsize[1:0]**, **p_d_htrans[1:0]**, **p_d_hunalign**, **p_d_hwdata[63:0]**, and **p_d_hwrite** change state (by definition, in conjunction with the **p_d_hready** input per the AHB protocol). The **p_d_ahb_clken** signal should normally be driven (change state) off the falling edge of **m_clk** to ensure the proper setup and hold times surrounding the **m_clk** high period. It must remain stable throughout the duration of **m_clk** high. This signal is ***not*** internally synchronized. It should be tied high when operating the data AHB at **m_clk** frequency. The integration guide defines the required setup time before **m_clk** rises and hold time after **m_clk** falls.

14.2.8 Master ID configuration signals

The following paragraphs describe the master ID configuration signals. These inputs are used to drive the **p_[d,i]_hmaster[3:0]** outputs when a bus cycle is active.

14.2.8.1 CPU master ID (p_masterid[3:0])

The **p_masterid[3:0]** input signals configure the master ID for the CPU. These values are driven on the **p_[d,i]_hmaster[3:0]** outputs for a CPU-initiated bus cycle.

14.2.8.2 Nexus master ID (nex_masterid[3:0])

The **nex_masterid[3:0]** input signals configure the master ID for the Nexus 3 unit. These values are driven on the **p_d_hmaster[3:0]** outputs for a Nexus 3 initiated bus cycle.

14.2.9 Coherency control signals

The following paragraphs describe the signals that control the Cache Coherency hardware functions. Examples of operation are provided in Section 14.3.4, Cache coherency interface operation.

14.2.9.1 Snoop ready (p_snp_rdy)

This active-high output signal indicates that the CPU is ready to accept a new snoop request. When asserted, it indicates that a new snoop cycle may be requested via the **p_snp_req** input during the following two clock cycles. When this signal is negated, a new snoop request will not be accepted after the next clock cycle, even if **p_snp_req** is asserted. This signal is asserted when the internal snoop queue contains two or more available entries for a new snoop request if a request is pending, or if three or more entries are available and no request is pending. The protocol is designed to prevent unnecessary transitions of the **p_snp_rdy** signal, as well as to support using **p_snp_rdy** to affect the **p_stall_bus_gwrite** input of another CPU to prevent queue overruns.

14.2.9.2 Snoop request (p_snp_req)

This active-high input signal indicates that the CPU should perform a new snoop request operation. When asserted, it indicates that a new snoop cycle is being requested based on additional information provided on the **p_snp_cmd[0:1]**, **p_snp_addr[0:26]**, and **p_snp_id_in[0:3]** input signals. A new snoop request will be ignored if the **p_snp_rdy** signal was negated two clock cycles earlier, even if **p_snp_req** is asserted.

14.2.9.3 Snoop command input (p_snp_cmd_in[0:1])

These input signals provide a command indicator for a snoop request. The command value is stored in the snoop queue along with the snoop address and snoop ID value. Table 14-15 shows the definitions of the **p_snp_cmd[0:1]** encodings.

Table 14-15. p_snp_cmd[0:1] snoop command encoding

p_snp_cmd[0:1]	Response type
00	Null — no status bit operation performed, lookup is performed (queue entry allocated)
01	INV — invalidate matching cache entry (queue entry allocated)
10	SYNC — synchronize snoop queue (queue entry allocated). p_snp_addr[0:26] is unused.
11	Reserved — do not use

The NULL command is used for testing of interface handshaking and other status gathering purposes. The NULL command performs a snoop lookup operation, but performs no actual cache tag or status modifications (even in the presence of tag parity or EDC errors). The INV command causes a snoop lookup and subsequent invalidation of a matching cache line. The SYNC command causes the snoop queue to be emptied with highest priority relative to CPU requests.

14.2.9.4 Snoop request ID input (p_snp_id_in[0:3])

These input signals provide an identifier value for a snoop request. The identifier value is stored in the snoop queue along with the snoop address and snoop command, and is only used by the CPU to be reflected on the **p_snp_id_out[0:3]** outputs when a snoop cycle is subsequently acknowledged via the **p_snp_ack** output.

14.2.9.5 Snoop address input (p_snp_addr_in[0:26])

These input signals provide the address value for a snoop request. The address value is stored in the snoop queue along with the snoop ID value and snoop command, and is used by the CPU to perform a cache line lookup when a snoop cycle is subsequently performed to the cache from the queue. The snoop address signals are used to index the cache and perform a tag compare with the physical cache tags. These inputs are not translated, thus they reflect the physical addresses of cached memory.

14.2.9.6 Snoop acknowledge (p_snp_ack)

This active high output signal is used to acknowledge that a previous snoop command request has been performed. When asserted, the signal indicates that the **p_snp_id_out[0:3]** and **p_snp_resp[0:4]** outputs are valid, and reflect the result of a completed snoop command.

14.2.9.7 Snoop request ID output (p_snp_id_out[0:3])

These output signals provide an ID value for a snoop request. The ID value is the value of **p_snp_id_in[0:3]** that was stored in the snoop queue along with the snoop address and snoop command during a previous snoop command request, and are only used by the CPU to be reflected on the **p_snp_id_out[0:3]** outputs when a snoop command is subsequently acknowledged via the **p_snp_ack** output.

14.2.9.8 Snoop response (p_snp_resp[0:4])

These output signals provide a response indicator for a processed snoop command request. The command value is stored in the snoop queue along with the snoop address and snoop tag value. Table 14-15 shows the definitions of the **p_snp_resp[0:4]** encodings.

Table 14-16. p_snp_resp[0:4] snoop response encoding

p_snp_resp[0:4][1]	Response type
000cc	NULL — no operation performed or no matching cache entry
001cc	Reserved
010cc	ERROR — Error in processing a snoop request due to TAG parity error. For NULL commands, a tag parity error occurred and no hit to a tag without error occurred. No modification of cache entries, no machine check generated internally. For INV commands, possible invalidation of locked line with tag parity error occurred, or dirty line left valid with tag parity error. Machine check generated internally.
01100	SYNC — Sync completed, snoop queue synchronized
100cc	HIT Clean— matching unlocked cache entry found
101cc	HIT Dirty— matching unlocked dirty cache entry found
110cc	HIT Locked — matching clean locked cache entry found
111cc	HIT Dirty Locked — matching dirty locked cache entry found

[1] cc — Number of collapsed requests
00—No collapsing
01— Two requests combined
10— Three requests combined
11— Four requests combined

14.2.9.9 Cache stalled (p_cac_stalled)

The active-high **p_cac_stalled** output signal is used to indicate that a CPU access to the data cache is stalled due to a snoop access to the cache. This signal may be monitored by system logic to determine the impact of snooping on CPU performance and to adjust the rate of snoops accordingly to minimize or distribute stall cycles.

14.2.9.10 Data cache enabled (p_d_cache_en)

The active-high **p_d_cache_en** output signal is used to indicate that the data cache is enabled or disabled. When disabled, no snoop lookups are performed, and a default Null response is given for snoop requests. This signal may be monitored by system logic to cancel pending snoop requests, or to manage a directory ownership or snoop filter by noting when the cache has been disabled and enabled. This signal reflects the state of the $L1CSR0_{DCE}$ control bit.

14.2.10 Memory synchronization control signals

The following paragraphs describe the signals that comprise the Memory Synchronization control functions. Examples of operation are shown in Section 14.3.3, Memory synchronization control operation.

14.2.10.1 Synchronization request in (p_sync_req_in)

This active-high input signal indicates that a synchronization operation is being requested by system logic. Assertion of this signal causes the CPU to empty the snoop queue of all valid entries present at the time the **p_sync_req_in** input was asserted. including any valid snoop command request accepted on the same clock cycle. This is a heavyweight synchronization operation that may affect system performance. This signal should remain asserted until acknowledged via assertion of the **p_sync_ack_out** signal, otherwise proper synchronization of all queues is not guaranteed. This signal is allowed to negate early however, if an interrupt causes a synchronization operation request to be aborted. Early negation does not guarantee that synchronization will not be completed as requested however.

This signal is not sampled during the Stopped low power state, thus it will not be acknowledged unless the Stopped state is exited and the signal is still seen asserted. SoC logic should ensure that no undesired system delay or deadlock can occur due to this behavior in the stopped state.

14.2.10.2 Synchronization request acknowledge out (p_sync_ack_out)

This active-high output signal indicates that a synchronization operation being requested by system logic via the **p_sync_req_in** input has completed. Assertion of this signal occurs after the CPU has emptied the snoop queue of all valid entries present at the time the **p_sync_req_in** input was asserted, including any valid snoop command request accepted on the same clock cycle.

This signal is qualified with the sampled value of **p_sync_req_in**, and will negate the cycle following negation of **p_sync_req_in**. If **p_sync_req_in** is negated prior to assertion of **p_sync_ack_out**, **p_sync_ack_out** will not be asserted.

During the Stopped state, **p_sync_ack_out** will remain negated. SoC logic must be aware of this and handle any synchronization request handshaking required to prevent a deadlock condition when another CPU attempts to execute a synchronization instruction and handshake a synchronization operation.

14.2.10.3 Synchronization request out (p_sync_req_out)

This active-high output signal indicates that a synchronization operation is being requested by the CPU. Assertion of this signal occurs during execution of an **msync** and mbar with MO = 0 or 1 (not for mbar w/MO=2) instruction by the CPU after it has suspended instruction and data fetches and emptied the store buffer.

This signal remains asserted until acknowledged via assertion of the **p_sync_ack_in** signal, unless a pending interrupt occurs. In this case the synchronization operation will be aborted and restarted at a later time, and the **p_sync_req_output** will be negated.

14.2.10.4 Synchronization request acknowledge in (p_sync_ack_in)

This active-high input signal indicates that a synchronization operation being requested by the assertion of **p_sync_req_out** by the CPU has completed.

This signal is sampled beginning with the clock cycle following assertion of **p_sync_req_out**, and will cause negation of **p_sync_req_out** the cycle after it is recognized as asserted. This signal should be negated the cycle after **p_sync_req_out** negates. This signal is ignored during the clock cycle that **p_sync_req_out** is initially asserted.

14.2.11 Interrupt signals

The following paragraphs describe the signals that control the interrupt functions. Interrupt request inputs **p_extint_b** and **p_critint_b** to the core are level sensitive, not edge-triggered, thus the interrupt controller module must keep the interrupt request as well as the **p_voffset** or **p_avec_b** inputs (as appropriate) asserted until the interrupt is serviced to guarantee that the CPU core recognizes the request. Once a request is generated, there is no guarantee the CPU will not recognize the interrupt request even if the request is later removed Interrupt requests must be held stable to avoid spurious responses. The interrupt inputs **p_nmi_b** and **p_mcp_b** are transition sensitive as described in Section 14.2.11.8, Machine check (p_mcp_b), and Section 14.2.11.3, Non-maskable input interrupt request (p_nmi_b).

14.2.11.1 External input interrupt request (p_extint_b)

This active-low signal provides the External Input interrupt request to the e200z759n3 core. **p_extint_b** is masked by the MSR[EE] bit. This signal is not internally synchronized by the e200z759n3 core, thus it must meet setup and hold time constraints relative to **m_clk** when the e200z759n3 core clock is running. This signal is level sensitive and must remain asserted to be guaranteed to be recognized.

14.2.11.2 Critical input interrupt request (p_critint_b)

This active-low signal provides the Critical Input interrupt request to the e200z759n3 core. **p_critint_b** is masked by the MSR[CE] bit. This signal is not internally synchronized by the e200z759n3 core, thus it must meet setup and hold time constraints relative to **m_clk** when the e200z759n3 core clock is running. This signal is level sensitive and must remain asserted to be guaranteed to be recognized.

14.2.11.3 Non-maskable input interrupt request (p_nmi_b)

This active-low, transition sensitive signal provides a non-maskable interrupt request to the e200z759n3 core. This signal is not internally synchronized by the e200z759n3 core, thus it must meet setup and hold time constraints to **m_clk** when the e200z759n3 core clock is running. The **p_nmi_b** input is sampled on two consecutive **m_clk** periods to detect a transition from the negated to the asserted state. Initiation of exception processing for the NMI will be internally qualified with this transition. Note that when the core is halted or stopped without clocks, transitions on this signal will not be immediately detected, but the **p_ipend** and **p_wakeup** signals will be asserted to indicate to system logic that an interrupt is pending and so the clocks should be started, and the **p_halt** and **p_stop** inputs should be negated in order for the interrupt to be processed.

14.2.11.4 Interrupt pending (p_ipend)

This active-high signal indicates that an asserted **p_extint_b, p_critint_b**, or **p_nmi_b** interrupt request input, or an enabled Timer facility interrupt (Watchdog, Fixed-Interval, or Decrementer) has been recognized internally by the core and is enabled by the appropriate bit in the MSR (**p_nmi_b** is never masked), and is asserted combinationally from the qualified interrupt request inputs as well as when the $MCSR_{NMI}$ syndrome bit is set. The **p_ipend** signal can be used to signal other bus masters or a bus arbiter that an interrupt condition is pending. External power management logic can use this output to control operation of the core and other logic or may use the **p_wakeup** signal similarly. Actual handling of the interrupt request may be delayed due to higher priority exceptions; assertion of **p_ipend** does not mean that exception processing for the interrupt has begun. The **p_nmi_b** input will affect the **p_ipend** signal slightly differently; the **p_ipend** output will assert any time the **p_nmi_b** input is asserted or whenever the $MCSR_{NMI}$ syndrome bit is set.

14.2.11.5 Autovector (p_avec_b)

This active-low signal is asserted with either the **p_extint_b** or **p_critint_b** interrupt request to request use of the internal IVOR4 or IVOR0 registers for obtaining an exception vector offset. If this signal is negated when a **p_extint_b** or **p_critint_b** interrupt is requested, an external vector offset is taken from the **p_voffset[0:15]** input signals. This signal is level sensitive and must remain asserted to be guaranteed to be recognized. This signal must be driven to a valid state during each clock cycle that either **p_extint_b** or **p_critint_b** is asserted.

14.2.11.6 Interrupt vector offset (p_voffset[0:15])

These input signals provide a vector offset to be used when exception processing begins for an incoming interrupt request. These signals are sampled along with the **p_extint_b** and **p_critint_b** interrupt request inputs, and must be driven to a valid value when either of these signals is asserted unless the **p_avec_b** signal is also asserted. If **p_avec_b** is asserted, these inputs are not used. The **p_voffset[0:15]** signals correspond to bits 16:31 of the IVOR registers. **p_voffset[0:11]** are used in forming the exception handler address, and **p_voffset[12:15]** are reserved and should be driven low. The **p_voffset[0:15]** signals are level sensitive and must remain asserted to be guaranteed to be recognized correctly. In addition, these signals must be asserted concurrently with the **p_extint_b** and **p_critint_b** inputs when used.

14.2.11.7 Interrupt vector acknowledge (p_iack)

The **p_iack** output signal provide an interrupt vector acknowledge indicator to allow external interrupt controllers to be informed when a critical input or external input interrupt is being processed. The **p_iack** signal will be asserted after the cycle in which the **p_avec_b** and **p_voffset[0:15]** signals are sampled in preparation for exception processing. See Figure 14-41 and Figure 14-42 for timing diagrams of operation.

14.2.11.8 Machine check (p_mcp_b)

This active-low, transition sensitive signal provides a Machine Check interrupt request to the e200z759n3 core. **p_mcp_b** is masked by the HID0[EMCP] bit. This signal is not internally synchronized by the e200z759n3 core, thus it must meet setup and hold time constraints to **m_clk** when the e200z759n3 core clock is running. The **p_mcp_b** input is sampled on two consecutive **m_clk** periods to detect a transition from the negated to the asserted state. Note that when the core is halted or stopped without clocks, transitions on this signal will not be immediately detected, so it must be held asserted until it can be recognized with the **m_clk** running.

The **p_mcp_b** signal is sampled while the e200z759n3 core is in debug mode or is in the waiting, halted, or stopped power management states if the **m_clk** is running. See Section 14.2.17.1, Processor waiting (p_waiting), Section 14.2.17.3, Processor halted (p_halted), and Section 14.2.17.5, Processor stopped (p_stopped).

14.2.12 External translation alteration signals

The following paragraphs describe the external translation alteration interface signals. A description of operation is provided in Section 10.11, External translation alterations for realtime systems.

14.2.12.1 External PID enable (p_extpid_en)

The active-high **p_extpid_en** input signal is used to enable the external translation alteration interface. Enabling of the dynamic mapping capability is controlled by asserting the **p_extpid_en** control input. This input is sampled with the rising edge of the clock, and when asserted, allows for the dynamic remapping capability to be used.

14.2.12.2 External PID in (p_extpid[6:7])

The active-high **p_extpid[6:7]** input signals are used to provide the PID[6:7] comparison values for certain TLB entries. These signals are qualified with the assertion of **p_extpid_en**.

14.2.13 Timer facility signals

The following sub-sections describe the processor signals associated with the Timer Facilities (Time Base, Watchdog, Fixed-interval and Decrementer).

14.2.13.1 Timer disable (p_tbdisable)

The active-high **p_tbdisable** input signal is used to disable the internal Time Base and Decrementer counters. When this signal is asserted, Time Base and Decrementer updates are frozen. When this signal is negated, Time Base and Decrementer updates are unaffected. This signal may be used to freeze the state of the Time Base and Decrementer during low power or debug operation. This signal is not internally synchronized by the e200z759n3 core, thus it must meet setup and hold time constraints relative to **m_clk** when the e200z759n3 core clock is running, as well as to **p_tbclk** when selected as an alternate clock source for the Time Base.

14.2.13.2 Timer external clock (p_tbclk)

The active-high **p_tbclk** input signal is used as an alternate clock source for the Time Base and Decrementer counters. Selection of this clock is made using the HID0[SEL_TBCLK] control bit (see Section 2.4.11, Hardware Implementation Dependent Register 0 (HID0)). This clock source must be synchronous to the **m_clk** input, and cannot exceed 50% of the **m_clk** frequency. This signal must be driven such that it changes state on the falling edge of **m_clk**.

14.2.13.3 Timer interrupt status (p_tbint)

The active-high **p_tbint** output signal is used to indicate that an internal timer facility unit is generating an interrupt request (TSR[WIS]=1 and TCR[WIE]=1 and MSR[CE]=1, or TSR[DIS]=1 and TCR[DIE]=1 and MSR[EE]=1, or TSR[FIS]=1 and TCR[FIE]=1 and MSR[CE]=1). This signal may be used to exit low power operation, or for other system purposes.

14.2.14 Processor reservation signals

The following sub-sections describe processor reservation signals associated with the **lbarx**, **lharx**, **lwarx**, **stbcx.**, **sthcx.**, and **stwcx.** instructions.

14.2.14.1 CPU reservation status (p_rsrv)

The active-high **p_rsrv** output signal is used to indicate that a reservation has been established by the execution of a load and reserve (**lbarx**, **lharx**, **lwarx**) instruction. This signal is set following the successful completion of a load and reserve instruction. This signal will remain set until the reservation has been cleared. (Refer to Section 3.5, Memory synchronization and reservation instructions). This signal is provided as a status indicator for specialized system applications only.

14.2.14.2 CPU reservation clear (p_rsrv_clr)

The active-high **p_rsrv_clr** input signal is used to clear a reservation that has been previously established. External reservation management logic may use this signal to implement reservation management policies that are outside of the scope of the CPU. (Refer to Section 3.5, Memory synchronization and reservation instructions). This signal may be asserted independently of any bus transfer.

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

The **p_d_xfail_b** signal is provided to indicate success/failure of a **stbcx.**, **sthcx.**, or **stwcx.** instruction as part of bus transfer termination using the XFAIL **p_d_hresp[2:0]** encoding. See Section 15.2.3.17, Store exclusive failure (p_d_xfail_b), for more detail on **p_d_xfail_b**.

14.2.15 Miscellaneous processor signals

The following paragraph describes several miscellaneous processor signals.

14.2.15.1 CPU ID (p_cpuid[0:7])

The active-high **p_cpuid[0:7]** input signals are used to provide an identity for a particular processor. These inputs are reflected in the Processor ID Register (Section 2.4.2, Processor ID Register (PIR)) following reset. These inputs are intended to remain in a static condition and are not internally synchronized.

14.2.15.2 PID0 outputs (p_pid0[0:7])

The active-high **p_pid0[0:7]** output signals are used to provide the current process ID in the Process ID Register 0 (PID0). These outputs correspond to the low order eight bits of PID0.

14.2.15.3 PID0 update (p_pid0_updt)

The active-high **p_pid0_updt** signal is used to indicate that the Process ID Register 0 (PID0) is being updated by a **mtspr** instruction. This output will assert during the clock cycle the **p_pid0[0:7]** outputs are changing.

14.2.15.4 System version (p_sysvers[0:31])

The active-high **p_sysvers[0:31]** input signals are used to provide a version number for the particular system incorporating a e200z759n3 CPU. These inputs are reflected in the System Version Register (Section 2.4.4, System Version Register (SVR)). These inputs are intended to remain in a static condition and are not internally synchronized.

14.2.15.5 Processor version (p_pvrin[16:31])

The active-high **p_pvrin[16:31]** input signals are used to provide a portion of the version number for a particular e200z759n3 CPU. These inputs are reflected in the Processor Version Register (Section 2.4.3, Processor Version Register (PVR)). These inputs are intended to remain in a static condition and are not internally synchronized.

14.2.15.6 HID1 system control (p_hid1_sysctl[0:7])

The active-high **p_hid1_sysctl[0:7]** output signals are used to provide a set of control output signals external to the CPU via values written to the HID1 special purpose register. These outputs change state following the rising edge of **m_clk**, and may need synchronization depending on actual use. See Section 2.4.12, Hardware Implementation Dependent Register 1 (HID1).

14.2.15.7 Debug event outputs (p_devnt_out[0:7])

The active-high **p_devnt_out[0:7]** output signals are used to provide a single-clock pulse based on the values written to the DEVNT field of the DEVENT debug register. These outputs correspond to the low order eight bits of DEVENT. Note that **p_devnt_out[0]** corresponds to the low order bit, not the MSB of the DEVNT field.

14.2.16 Processor state signals

The following sub-sections describe processor internal state signals.

14.2.16.1 Processor mode (p_mode[0:3])

These signals indicate the global processor execution status. The timing is synchronous with **m_clk**. Table 14-18 shows **p_mode[0:3]** encoding.

Table 14-17. Processor mode encoding

p_mode[0:3]				Internal processor mode
0	0	0	0	Execution stalled
0	0	0	1	Execute exception
0	0	1	0	Instruction squashed
0	0	1	1	Normal processing
0	1	0	0	Processor in Halted state
0	1	0	1	Processor in Stopped state
0	1	1	0	Processor in Debug mode[1]
0	1	1	1	Reserved
1	0	0	0	Processor in Waiting state

[1] As reflected on the **cpu_dbgack** internal state signal

14.2.16.2 Processor execution pipeline status (p_pstat_pipe0[0:5], p_pstat_pipe1[0:5])

These signals indicate the internal execution pipeline status. The timing is synchronous with the **m_clk**, so the indicated status may not apply to a current bus transfer. Pipe0 corresponds to the oldest instruction in the pipeline, pipe1 to the next to oldest instruction. Table 14-18 shows **p_pstat_pipe{0,1}[0:5]** encodings.

Table 14-18. Processor execution pipeline status encoding[1]

p_pstat_pipe{0,1}[0:5]						Processor pipeline status
0	0	0	0	s	m	Complete instruction[2,3]
0	0	0	1	0	0	Complete **lmw, stmw, e_lmw, e_stmw, e_lmvgprw, e_stmvgprw, e_lmvsprw, e_stmvsprw, e_lmv[c,d,mc,]srrw, e_stmv[c,d,mc,]srrw**
0	0	0	1	0	1	Complete **e_lmw**, or **e_stmw**
0	0	1	0	0	0	Complete **isync**
0	0	1	0	1	1	Complete **se_isync**
0	0	1	1	0	m	Complete **lbarx, lharx, lwarx, stbcx., sthcx.**, or **stwcx.**[4]
0	1	0	0	0	m	Complete **evsel** with condition false for both elements
0	1	0	1	0	m	Complete **evsel** with condition false for high element and true for low element
0	1	1	0	0	m	Complete **evsel** with condition true for high element and false for low element
0	1	1	1	0	m	Complete **evsel** with condition true for both elements
1	0	0	0	0	0	Complete branch instruction **bc**, **bcl**, **bca**, **bcla, b, bl, ba, bla** resolved as not taken
1	0	0	0	0	1	Complete branch instruction **e_bc**, **e_bcl**, **e_b, e_bl** resolved as not taken
1	0	0	0	1	1	Complete branch instruction **se_bc**, **se_b, se_bl** resolved as not taken
1	0	0	1	0	0	Complete branch instruction **bc**, **bcl**, **bca**, **bcla, b, bl, ba, bla** resolved as taken
1	0	0	1	0	1	Complete branch instruction **e_bc**, **e_bcl**, **e_b, e_bl** resolved as taken
1	0	0	1	1	1	Complete branch instruction **se_bc**, **se_b, se_bl** resolved as taken
1	0	1	0	0	0	Complete **bclr**, **bclrl, bcctr**, **bcctrl** resolved as not taken
1	0	1	1	0	0	Complete **bclr**, **bclrl, bcctr**, **bcctrl** resolved as taken
1	0	1	1	1	1	Complete **se_blr**, **se_blrl, se_bctr**, **se_bctrl** (always taken)
1	1	0	0	0	m	Complete **isel** with condition false
1	1	0	1	0	m	Complete **isel** with condition true
1	1	1	0	x	x	No instruction completed
1	1	1	1	0	0	Complete **rfi**, **rfci**, **rfdi**, or **rfmci**
1	1	1	1	1	1	Complete **se_rfi**, **se_rfci**, **se_rfdi**, or **se_rfmci**

[1] All encodings that do not appear in the table are reserved

[2] Except **rfi, rfci, rfdi, rfmci, lmw, stmw, lbarx, lharx, lwarx, stbcx., sthcx., stwcx., isync, isel, se_rfi, se_rfci, se_rfdi, se_rfmci, e_lmw, e_stmw, se_isel,** and Change of Flow Instructions

[3] s — instruction size, 0=32-bit, 1=16-bit

m — 0 for BookE page, 1 for VLE page

[4] m — 0 for BookE page, 1 for VLE page

14.2.16.3 Branch prediction status (p_brstat[0:1])

These signals indicate the status of a branch prediction prefetch. Branch prediction prefetches are performed for Branch Target Buffer hits with predict taken status to accelerate branches. The timing is synchronous with the **m_clk**, so the indicated status may not apply to a current bus transfer. Table 14-19 shows **p_brstat[0:1]** encoding.

Table 14-19. Branch prediction status encoding

p_brstat[0:1]		Branch prediction status
0	x	Default (no branch predicted taken prefetch)
1	0	Branch predicted taken prefetch resolved as not taken
1	1	Branch predicted taken prefetch resolved as taken

14.2.16.4 Processor exception enable MSR values (p_msr_EE, p_msr_CE, p_msr_DE, p_msr_ME)

These active-high output signals reflect the state of the corresponding MSR[EE,CE,DE,ME] bits. They may be used by external system logic to determine the set of enabled exceptions. These signals change state on execution of a **mtmsr**, **rfi**, **rfci**, **rfdi**, **rfmci**, **se_rfi**, **se_rfci**, **se_rfdi**, **se_rfmci**, **wrtee**, or **wrteei** instruction, or during exception processing where one or more bits may be cleared during the exception processing sequence.

14.2.16.5 Processor return from interrupt (p_rfi, p_rfci, p_rfdi, p_rfmci)

These active-high output signals reflect the state of the processor when executing a return from interrupt class instruction. The signals are asserted for one clock during the execution of the corresponding **rfi**, **rfci**, **rfdi**, **rfmci**, **se_rfi**, **se_rfci**, **se_rfdi**, or **se_rfmci** instruction. They may be used by external system logic to determine the execution state of one or more nested or un-nested interrupt exception handlers, and may be used to provide hardware assist to external interrupt controllers, or priority elevation mechanisms. In conjunction with the interrupt acknowledge and exception enable outputs, an external state machine may track the entry and exit status of handlers for various classes and priorities of interrupts.

14.2.16.6 Processor machine check (p_mcp_out)

The active-high **p_mcp_out** output signal is asserted by the processor when a machine check condition has caused an “Async Mchk” or “Error Report” type syndrome bit to be set in the Machine Check Syndrome register. Refer to Section 2.4.7, Machine Check Syndrome Register (MCSR).

14.2.17 Power management control signals

The following signals are provided for power management or other control functions by external control logic.

14.2.17.1 Processor waiting (p_waiting)

The active-high **p_waiting** output signal is used to indicate that the processor has entered the Waiting state (Section 9.1.2, Waiting state).

14.2.17.2 Processor halt request (p_halt)

The active-high **p_halt** input signal is used to request the processor to enter the Halted state (Section 9.1.3, Halted state).

14.2.17.3 Processor halted (p_halted)

The active-high **p_halted** output signal is used to indicate that the processor has entered the Halted state (Section 9.1.3, Halted state).

14.2.17.4 Processor stop request (p_stop)

The active-high **p_stop** input signal is used to request the processor to enter the Stopped state (Section 9.1.4, Stopped state).

14.2.17.5 Processor stopped (p_stopped)

The active-high **p_stopped** output signal is used to indicate that the processor has entered the Stopped state (Section 9.1.4, Stopped state).

14.2.17.6 Low-power mode signals (p_doze, p_nap, p_sleep)

The active-high **p_doze**, **p_nap**, and **p_sleep** output signals are asserted by the processor to reflect the settings of the HID0[DOZE], HID0[NAP], and HID0[SLEEP] control bits when the MSR[WE] bit is set.

These outputs may assert for one or more clock cycles. External logic can detect the asserted edge or level of these signals to determine which low-power mode has been requested and then place the e200z759n3 core and peripherals in a low-power consumption state. The **p_wakeup** signal can be monitored to determine when to end the low-power condition.

The e200z759n3 core can be placed in a low-power state by forcing the **m_clk** input to a quiescent state, and brought out of low-power state by re-enabling **m_clk**. The Time Base facilities may be separately enabled or disabled using combinations of the Timer Facility control signals described in Section 14.2.13, Timer facility signals.

14.2.17.7 Wakeup (p_wakeup)

The active-high **p_wakeup** output signal should be used by external logic to remove the e200z759n3 core and system logic from a low-power state. It also is used to indicate to the system clock controller that the **m_clk** input should be re-enabled for debug purposes. This signal is asynchronous to the system clock and should be synchronized to the system clock domain to avoid hazards.

p_wakeup asserts whenever:

- A valid pending interrupt is detected by the core
- A request to enter debug mode is made by setting the DR bit in the OnCE control register (OCR) or via the assertion of the **jd_de_b** or **p_ude** input signals.
- The processor is in a debug session and the **jd_debug_b** output is asserted

- A request to enable the **m_clk** input has been made by setting the WKUP bit in the OnCE control register
- The **p_nmi_b** input is asserted or the $MCSR_{NMI}$ syndrome bit is set

p_wakeup (or other system state) should be monitored to determine when to release the processor (and system if applicable) from a low-power state.

14.2.18 Performance monitor signals

The following interface signals are for the Performance Monitor unit.

14.2.18.1 Performance monitor event (p_pm_event)

The active-high **p_pm_event** input signal is used to signal a performance monitor counted event. Selection of this event is described in Section 8.7, Event selection. This signal is not internally synchronized by the e200z759n3 core, thus it must meet setup and hold time constraints relative to **m_clk** when the e200z759n3 core clock is running. This signal is both level and transition sensitive.

14.2.18.2 Performance monitor counter 0 overflow state (p_pmc0_ov)

The active-high **p_pmc0_ov** output signal is used to reflect the state of the performance monitor counter 0 OV bit ($PMC0_{OV}$) described in Section 8.3.9, Performance Monitor Counter registers (PMC0–PMC3).

14.2.18.3 Performance monitor counter 1 overflow state (p_pmc1_ov)

The active-high **p_pmc1_ov** output signal is used to reflect the state of the performance monitor counter 1 OV bit ($PMC1_{OV}$) described in Section 8.3.9, Performance Monitor Counter registers (PMC0–PMC3).

14.2.18.4 Performance monitor counter 2 overflow state (p_pmc2_ov)

The active-high **p_pmc2_ov** output signal is used to reflect the state of the performance monitor counter 2 OV bit ($PMC2_{OV}$) described in Section 8.3.9, Performance Monitor Counter registers (PMC0–PMC3).

14.2.18.5 Performance monitor counter 3 overflow state (p_pmc3_ov)

The active-high **p_pmc3_ov** output signal is used to reflect the state of the performance monitor counter 3 OV bit ($PMC3_{OV}$) described in Section 8.3.9, Performance Monitor Counter registers (PMC0–PMC3).

14.2.18.6 Performance monitor counter 3 qualifier inputs (p_pmc[0,1,2,3]_qual)

The active-high **p_pmc[0,1,2,3]_qual** input signals are used to provided additional triggering control means for the respective performance monitor counters. Triggering control is described in Section 8.3.7, Performance Monitor Local Control B Registers (PMLCb0–PMLCb3).

14.2.19 Debug event input signals

The following interface signals are provided to signal debug events to the e200z759n3 core.

14.2.19.1 Unconditional debug event (p_ude)

The active-high **p_ude** input signal is used to request an unconditional debug event. This event is described in detail in Section 12.2.13, Unconditional debug event. This signal is not internally synchronized by the e200z759n3 core, thus it must meet setup and hold time constraints relative to **m_clk** when the e200z759n3 core clock is running. This signal is level sensitive and must be held asserted until acknowledged by software, or, when external debug mode is enabled, by assertion of the **jd_debug_b** output to be guaranteed to be recognized. In addition, only a transition from the negated state to the asserted state of the **p_ude** signal will cause an event to occur. The level on this signal is used however to cause assertion of the **p_wakeup** output.

14.2.19.2 External debug event 1 (p_devt1)

The active-high **p_devt1** input signal is used to request an external debug event. This event is described in detail in Section 12.2.12, External debug event. This signal is not internally synchronized by the e200z759n3 core, thus it must meet setup and hold time constraints relative to **m_clk** when the e200z759n3 core clock is running. If the e200z759n3 core clock is disabled, this signal will not be recognized. In addition, only a transition from the negated state to the asserted state of the **p_devt1** signal will cause an event to occur. It is intended to signal e200z759n3-related events that are generated while the CPU is active.

14.2.19.3 External debug event 2 (p_devt2)

The active-high **p_devt2** input signal is used to request an external debug event. This event is described in detail in Section 12.2.12, External debug event. This signal is not internally synchronized by the e200z759n3 core, thus it must meet setup and hold time constraints relative to **m_clk** when the e200z759n3 core clock is running. If the e200z759n3 core clock is disabled, this signal will not be recognized. In addition, only a transition from the negated state to the asserted state of the **p_devt2** signal will cause an event to occur. It is intended to signal e200z759n3-related events that are generated while the CPU is active.

14.2.20 Debug event output signals (p_devnt_out[0:7])

The active-high **p_devnt_out[0:7]** output signals are used to provide a single-clock pulse based on the values written to the DEVNT field of the DEVENT debug register. These outputs correspond to the low order eight bits of DEVENT. Note that **p_devnt_out[0]** corresponds to the low order bit, not the MSB of the DEVNT field.

14.2.21 Debug/emulation (Nexus 1/ OnCE) support signals

The following interface signals are provided to assist in implementing an On-Chip Emulation capability with a controller external to the e200z759n3 core.

Table 14-20. e200z759n3 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

14.2.21.1 OnCE enable (jd_en_once)

The OnCE enable signal **jd_en_once** is used to enable the OnCE controller to allow certain instructions and operations to be executed. Assertion of this signal will enable the full OnCE command set, as well as operation of control signals and OnCE Control register functions. When this signal is disabled, only the Bypass, ID and Enable_OnCE commands are executed by the OnCE unit, and all other commands default to a “Bypass” command. The OnCE Status register (OSR) is not visible when OnCE operation is disabled. In addition, OnCE Control register (OCR) functions are disabled, as is the operation of the **jd_de_b** input. Secure systems may choose to leave this signal negated until a security check has been performed. Other systems should tie this signal asserted to enable full OnCE operation. The **j_en_once_regsel** and **j_key_in** signals are provided to assist external logic performing security checks. Refer to Section 14.2.23.15, Enable OnCE register select (j_en_once_regsel), for a description of the **j_en_once_regsel** output signal, and to Section 14.2.23.19, Key data in (j_key_in), for a description of the **j_key_in** input signal.

The **jd_en_once** input must only change state during the Test-Logic-Reset, Run-Test/Idle, or Update_DR TAP states. A new value will take affect after one additional **j_tclk** cycle of synchronization.

14.2.21.2 Debug session (jd_debug_b)

The **jd_debug_b** active-low output signal is asserted when the processor first enters into debug mode. It remains asserted for the duration of a debug session.

NOTE

A debug session includes single-step operations (Go+NoExit OnCE commands). That is, **jd_debug_b** remains asserted during OnCE single-step executions.

This signal is provided to allow system resources to be aware that access is occurring for debug purposes, thus allowing certain resource side effects to be frozen or otherwise controlled. Examples might include FIFO state change control, control of side-effects of register or memory accesses, etc. Refer to Section 12.4.5.3, e200z759n3 OnCE debug output (jd_debug_b), for additional information on this signal.

14.2.21.3 Debug request (jd_de_b)

This signal is the debug mode request input. This signal is not internally synchronized by the e200z759n3 core, thus it must meet setup and hold time constraints relative to **j_tclk.** To be recognized, it must be held asserted for a minimum of two **j_tclk** periods, and the **jd_en_once** input must be in the asserted state. **jd_de_b** is synchronized to **m_clk** in the debug module before being sent to the processor (two clocks).

This signal is normally the input from the top-level **DE_b** open-drain bidirectional I/O cell. Refer to Section 12.4.5.2, OnCE debug request/event (jd_de_b, jd_de_en), for additional information on this signal.

14.2.21.4 DE_b active high output enable (jd_de_en)

This output signal is an active-high enable for the top-level **DE_b** open-drain bidirectional I/O cell. This signal is asserted for three **j_tclk** periods upon processor entry into debug mode. Refer to Section 12.4.5.2, OnCE debug request/event (jd_de_b, jd_de_en), for additional information on this signal.

14.2.21.5 Processor clock on (jd_mclk_on)

This active-high input signal is driven by system level clock control logic to indicate that the processor's **m_clk** input is active. This signal is synchronized to **j_tclk** and provided as a status bit in the OnCE Status register.

14.2.21.6 Watchpoint events (jd_watchpt[0:29])

The **jd_watchpt[0:29]** active-high output signals are used to indicate that a watchpoint has occurred. Each debug address compare function (IAC1-8, DAC1-2), and Debug Counter event (DCNT1-2) is capable of triggering a watchpoint output, and in addition DEVNT-, DTC-based, and Performance Monitor watchpoints are supported. Refer to Section 12.5, Watchpoint support, for the signal assignments of each watchpoint source.

14.2.22 Development support (Nexus 3) signals

The following interface signals are provided to assist in implementing a real-time development tool capability with a controller external to the e200z759n3 core.

Table 14-21. e200z759n3 development support (Nexus) signals

Signal	Type	Description
nex_mcko	O	Nexus clock output
nex_rdy_b	O	Nexus ready output
nex_evto_b	O	Nexus event-out output
nex_wevto[3:0]	O	Nexus event-out output
nex_evti_b	I	Nexus event-in input
nex_mdo[n:0]	O	Nexus message data output
nex_mseo_b[1:0]	O	Nexus message start/end output
nex_ext_src_id[0:3]	I	Nexus SRC ID input

14.2.23 JTAG support signals

Table 14-22 details the primary JTAG interface signals. These signals are usually connected directly to device pins (except for **j_tdo**, which needs tri-state and edge support logic). However, this may not be the case when JTAG TAP controllers are concatenated together.

Table 14-22. JTAG primary interface signals

Signal name	Type	Description
j_trst_b	I	JTAG test reset
j_tclk	I	JTAG test clock
j_tms	I	JTAG test mode select
j_tdi	I	JTAG test data input
j_tdo	O	Test data out to master controller or pad
j_tdo_en[1]	O	Enables TDO output buffer

[1] j_tdo_en is asserted when the TAP controller is in the shift_dr or shift_ir state.

14.2.23.1 JTAG/OnCE serial input (j_tdi)

Data and commands are provided to the OnCE controller through the **j_tdi** pin. Data is latched on the rising edge of the **j_tclk** serial clock. Data is shifted into the OnCE serial port least significant bit (LSB) first.

14.2.23.2 JTAG/OnCE serial clock (j_tclk)

The **j_tclk** pin supplies the serial clock to the OnCE control block. The serial clock provides pulses required to shift data and commands into and out of the OnCE serial port. (Data is clocked into the OnCE on the rising edge and is clocked out of the OnCE serial port on the rising edge.) The debug serial clock frequency must be no greater than 50% of the processor clock frequency.

14.2.23.3 JTAG/OnCE serial output (j_tdo)

Serial data is read from the OnCE block through the **j_tdo** pin. Data is always shifted out the OnCE serial port least significant bit (LSB) first. When data is clocked out of the OnCE serial port, **j_tdo** changes on the rising edge of **j_tclk**. The **j_tdo** output signal is always driven.

An external system-level TDO pin may be tri-stateable and should be actively driven in the shift-IR and shift-DR controller states. The **j_tdo_en** signal is supplied to indicate when an external TDO pin should be enabled and is asserted during the shift-IR and shift-DR controller states. In addition, for IEEE1149 compliance, the system level pin should change state on the falling edge of TCLK.

14.2.23.4 JTAG/OnCE test mode select (j_tms)

The **j_tms** input is used to cycle through states in the OnCE Debug Controller. Toggling the **j_tms** pin while clocking with **j_tclk** controls transitions through the TAP state controller.

14.2.23.5 JTAG/OnCE test reset (j_trst_b)

The **j_trst_b** input is used to externally reset the OnCE controller by placing it in the Test-Logic-Reset state.

Table 14-23 details additional signals that may be used to support external JTAG data registers using the e200z759n3 TAP controller.

Table 14-23. JTAG signals used to support external registers

Signal Name	Type	Description
j_tst_log_rst	O	Indicates the TAP controller is in the Test-Logic-Reset state
j_rti	O	JTAG controller run-test/idle state
j_capture_ir	O	Indicates the TAP controller is in the capture IR state
j_shift_ir	O	Indicates the TAP controller is in shift IR state
j_update_ir	O	Indicates the TAP controller is in update IR state
j_capture_dr	O	Indicates the TAP controller is in the capture DR state
j_shift_dr	O	Indicates the TAP controller is in shift DR state
j_update_gp_reg	O	Updates JTAG controller general-purpose data register
j_gp_regsel[0:9]	O	General-purpose external JTAG register select
j_en_once_regsel	O	External enable OnCE register select
j_key_in	I	Serial data from external key logic
j_nexus_regsel	O	External Nexus register select
j_lsrl_regsel	O	External LSRL register select
j_serial_data	I	Serial data from external JTAG register(s)

14.2.23.6 Test-Logic-Reset (j_tst_log_rst)

This signal indicates the TAP controller is in the Test-Logic-Reset state.

14.2.23.7 Run-Test/Idle (j_rti)

This signal indicates the TAP controller is in the Run-Test/Idle state.

14.2.23.8 Capture IR (j_capture_ir)

This signal indicates the TAP controller is in the Capture_IR state.

14.2.23.9 Shift IR (j_shift_ir)

This signal indicates the TAP controller is in the Shift_IR state.

14.2.23.10 Update IR (j_update_ir)

This signal indicates the TAP controller is in the Update_IR state.

14.2.23.11 Capture DR (j_capture_dr)

This signal indicates the TAP controller is in the Capture_DR state.

14.2.23.12 Shift DR (j_shift_dr)

This signal indicates the TAP controller is in the Shift_DR state.

14.2.23.13 Update DR w/write (j_update_gp_reg)

This signal indicates the TAP controller is in the Update_DR state and that the R/W bit in the OnCE Command register is low (write command). The **j_gp_regsel[0:9]** signals should be monitored to see which register, if any, needs to be updated.

14.2.23.14 Register select (j_gp_regsel)

The outputs shown in Table 14-24 are a decode of the REGSEL[0:6] field in the OnCE Command Register (OCMD). They are used to specify which external general purpose JTAG register to access via the e200z759n3 TAP controller.

Table 14-24. JTAG general purpose register select decoding

Signal name	Type	Description
j_gp_regsel[0]	O	REGSEL[0:6]=7’h70
j_gp_regsel[1]	O	REGSEL[0:6]=7’h71
j_gp_regsel[2]	O	REGSEL[0:6]=7’h72
j_gp_regsel[3]	O	REGSEL[0:6]=7’h73
j_gp_regsel[4]	O	REGSEL[0:6]=7’h74
j_gp_regsel[5]	O	REGSEL[0:6]=7’h75
j_gp_regsel[6]	O	REGSEL[0:6]=7’h76
j_gp_regsel[7]	O	REGSEL[0:6]=7’h77
j_gp_regsel[8]	O	REGSEL[0:6]=7’h78
j_gp_regsel[9]	O	REGSEL[0:6]=7’h79

14.2.23.15 Enable OnCE register select (j_en_once_regsel)

The **j_en_once_regsel** output is asserted when a decode of the REGSEL[0:6] field in the OnCE Command Register (OCMD) indicates an external Enable_OnCE register is selected (0b1111110 encoding) for access via the e200z759n3 TAP controller. This control signal may be used by external security logic to assist in controlling the **jd_enable_once** input signal. The external Enable_OnCE register should be muxed onto the **j_serial_data** input (Refer to Section 14.2.23.18, Serial data (j_serial_data)). During the Shift_DR state, **j_serial_data** is supplied to the **j_tdo** output.

14.2.23.16 External Nexus register select (j_nexus_regsel)

The **j_nexus_regsel** output is asserted when a decode of the REGSEL[0:6] field in the OnCE Command Register (OCMD) indicates an external Nexus register is selected (0b1111100 encoding) for access via the e200z759n3 TAP controller.

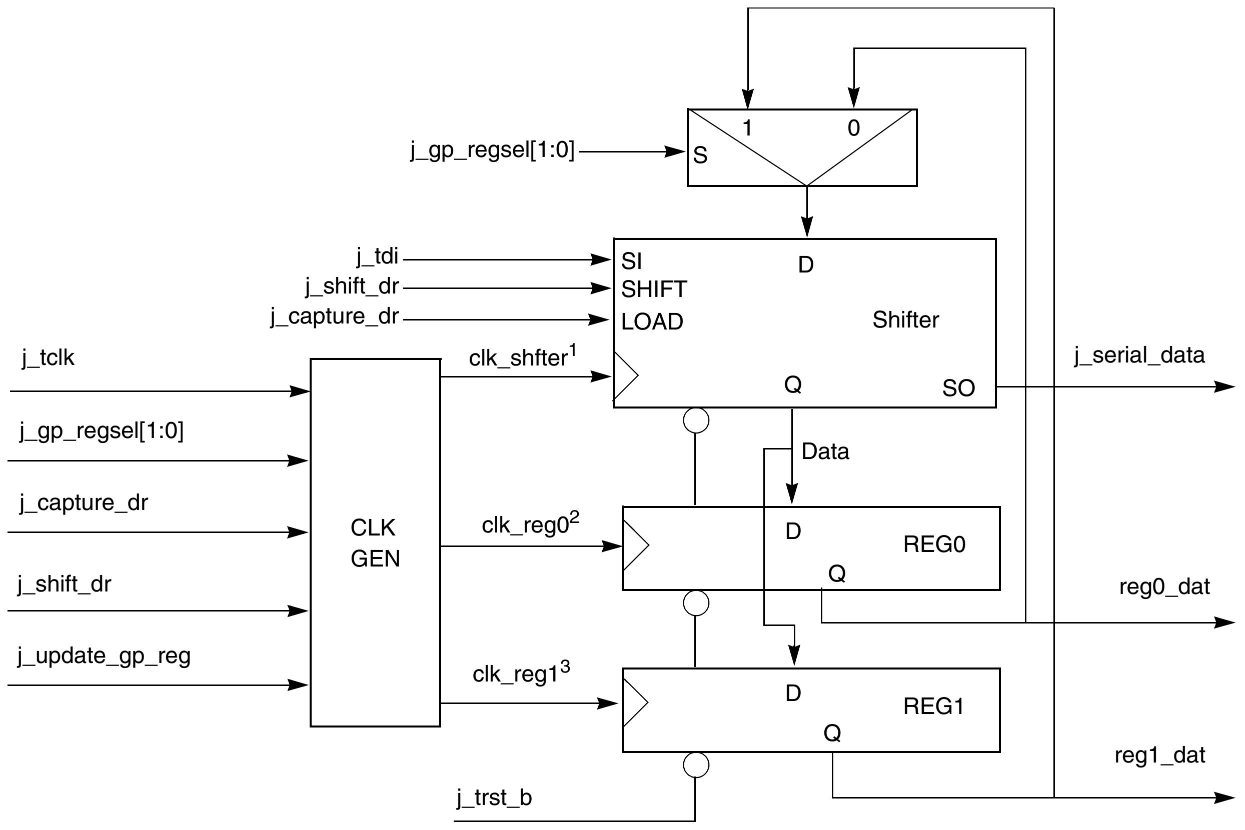
14.2.23.17 External LSRL register select (j_lsrl_regsel)

The **j_lsrl_regsel** output is asserted when a decode of the REGSEL[0:6] field in the OnCE Command Register (OCMD) indicates an external LSRL register is selected (0b1111101 encoding) for access via the e200z759n3 TAP controller.

14.2.23.18 Serial data (j_serial_data)

This input signal receives serial data from external JTAG registers. All external registers share this one serial output back to the core, therefore it must be muxed using the **j_gp_regsel[0:9]**, **j_lsrl_regsel**, and **j_en_once_regsel** signals. The data is internally routed to **j_tdo**.

Figure 14-3 shows one example of how an external JTAG register set (2) could be designed using the inputs and outputs provided and by the JTAG primary inputs themselves. The main components are a clock generation unit, a JTAG shifter (load, shift, hold, clr), the registers (load, hold, clr), and an input mux to the shifter for the serial output back to the e200z759n3 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 14-3. Example external JTAG register design

14.2.23.19 Key data in (j_key_in)

This input signal receives serial data from logic to indicate a key or other value to be scanned out in the Shift_IR state when the current value in the IR is the Enable_OnCE instruction. This input is provided to assist in implementing security logic outside of the e200z759n3 that conditionally asserts **jd_en_once**. During the Shift_IR state, when **jd_en_once** is negated, this input is sampled on the rising edge of **j_tclk**, and after a two clock delay the data is internally routed to **j_tdo**. This allows provision of a key value via the **j_tdo** output following a transition from Capture_IR to Shift_IR. The key value is provided via the **j_key_in** input.

14.2.24 JTAG ID signals

Table 14-25 shows the JTAG ID register unique to Freescale as specified by the *IEEE 1149.1 JTAG Specification*. Note that bit 31 is the MSB of this register.

Table 14-25. JTAG register ID fields

Bit field	Type	Description	Value
[31:28]	Variable	Version number	Variable
[27:22]	Fixed	Design center number (ZEN)	6’b011111
[21:12]	Variable	Sequence number	Variable
[11:1]	Fixed	Freescale manufacturer ID	11’b00000001110
0	Fixed	JTAG ID register identification bit	1’b1

The e200z759n3 core shifts out a “1” as the first bit on **j_tdo** if the Shift_DR state is entered directly from the test-logic-reset state. This is per the JTAG specification and informs any JTAG controller that an ID register exists on the part. The e200z759n3 JTAG ID register is accessed by writing the OCMD (OnCE Command Register) with the value 7’h02 in the REGSEL[0:6] field.

The JTAG ID bit, manufacturer ID field, and design center number are fixed by the JTAG Consortium and/or Freescale. The version numbers and the two most significant bits (MSBs) of the sequence number are variable and brought out to external ports. The lower eight bits of the sequence number are variable and strapped internally to track variations in processor deliverables.

Table 14-26 shows the inputs to the JTAG ID register that are input ports on the e200z759n3 core. These bits are provided for a customer to track revisions of a device using the e200z759n3 core.

Table 14-26. JTAG ID register inputs

Signal name	Type	Description
j_id_sequence[0:1]	I	JTAG ID register (2 MSBs of sequence field)
j_id_version[0:3]	I	JTAG ID register version field

14.2.24.1 JTAG ID sequence (j_id_sequence[0:1])

The **j_id_sequence[0:1]** inputs correspond to the two MSBs of the 10-bit sequence number in the JTAG ID register. These inputs are normally static. They are provided for the customer for further component variation identification.

14.2.24.2 JTAG ID sequence (j_id_sequence[2:9])

The **j_id_sequence[2:9]** field is internally strapped to track variations in processor and module deliverables. Each e200z759n3 deliverable has a unique sequence number. Additionally, each revision of these modules can be identified by unique sequence numbers.

14.2.24.3 JTAG ID version (j_id_version[0:3])

The **j_id_version[0:3]** inputs correspond to the 4-bit version number in the JTAG ID register. These inputs are normally static. They are provided to the customer for strapping in order to facilitate easy identification of component variants.

14.2.25 Test signals

Please refer to the *e200z759n3 Test Guide* for information on Test signals.

14.3 Timing diagrams

14.3.1 AHB clock enable and the internal HCLK

The CPU generates an internal HCLK to control AHB signal input sampling and output transitions based on the internal **m_clk** and the **p_[i,d]_ahb_clken** signals. The following diagrams show the relationships of these signals and the resulting HCLK. Note that since no AHB signals are sampled or change state on the falling edge of HCLK, the duty cycle is not an issue.

Figure 14-4 shows an example of a free-running half-speed HCLK relative to **m_clk**.

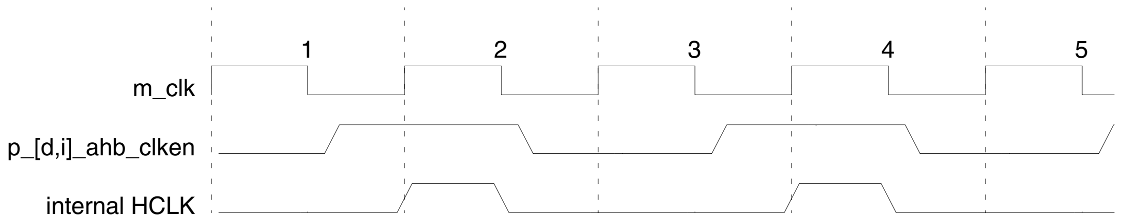


Figure 14-4. AHB clock enable operation — 1

Figure 14-5 shows an example of a free-running 1/3 speed HCLK relative to **m_clk**.

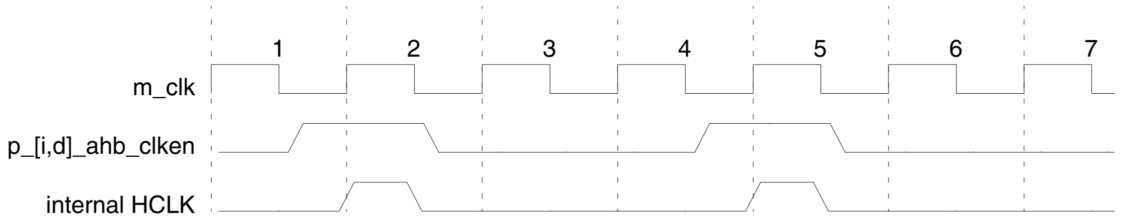


Figure 14-5. AHB clock enable operation — 2

Figure 14-6 shows an example of a non-periodic HCLK, used for power reduction, relative to **m_clk**.

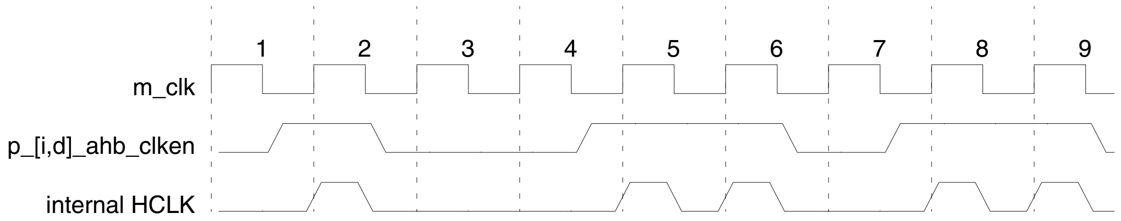


Figure 14-6. AHB clock enable operation —3

14.3.2 Processor instruction/data transfers

Transfer of data between the core and peripherals involves the address bus, data busses, and control and attribute signals. The address and data buses are parallel, non-multiplexed buses, supporting byte, halfword, three byte, word, and doubleword transfers. All bus input and output signals are sampled and driven with respect to the rising edge of the **m_clk** signal. The core moves data on the bus by issuing control signals and using a handshake protocol to ensure correct data movement.

The memory interface operates in a pipelined fashion to allow additional access time for memory and peripherals. AHB transfers consist of an address phase that lasts only a single cycle, followed by the data phase, which may last for one or more cycles depending on the state of the **p_hready** signal.

Read transfers consist of a request cycle, where address and attributes are driven along with a transfer request, and one or more memory access cycles to perform accesses and return data to the CPU for alignment, sign or zero extension, and forwarding.

Write transfers consist of a request cycle, where address and attributes are driven along with a transfer request, and one or more data drive cycles where write data is driven and external devices accept write data for the access.

Access requests are generated in an overlapped fashion in order to support sustained single cycle transfers. Up to two access requests may be in progress at any one cycle, one access outstanding and a second in the pending request phase.

Access requests are assumed to be accepted as long as there are no accesses in progress, or if an access in progress is terminated during the same cycle a new request is active (**p_hready** asserted). Once an access has been accepted, the BIU is free to change the current request at any time, even if part of a burst transfer.

The local memory control logic is responsible for proper pipelining and latching of all interface signals to initiate memory accesses.

The system hardware can use the **p_hresp[2:0]** signals to signal that the current bus cycle has an error when a fault is detected, using the ERROR response encoding. ERROR assertion requires a two cycle response. In the first cycle of the response, the **p_hresp[2:0]** signals are driven to indicate ERROR and **p_hready** must be negated. During the following cycle, the ERROR response must continue to be driven, and **p_hready** must be asserted. When the core recognizes a bus error condition for an access at the end of the first cycle of the two cycle error response, a subsequent pending access request may be removed by the BIU driving the **p_htrans[2:0]** signals to the IDLE state in the second cycle of the two cycle error response. Not all pending requests will be removed however.

When a bus cycle is terminated with a bus error, the core can enter storage error exception processing immediately following the bus cycle, or it can defer processing the exception.

The instruction prefetch mechanism requests instruction words from the instruction memory unit before it is ready to execute them. If a bus error occurs on an instruction fetch, the core does not take the exception until it attempts to use the instruction. Should an intervening instruction cause a branch, or should a task switch occur, the storage error exception for the unused access does not occur. A bus error termination for any write access or read access that reference data specifically requested by the execution unit causes the core to begin exception processing.

NOTE

In the following diagrams showing AHB operations, note that the HCLK signal is that of the AHB bus, i.e. **m_clk** qualified by **p_[i,d]_ahb_clken**

14.3.2.1 Basic read transfer cycles

During a read transfer, the core receives data from a memory or peripheral device. Figure 14-7 illustrates functional timing for basic read transfers. Clock-by-clock descriptions of activity in Figure 14-7 follows:

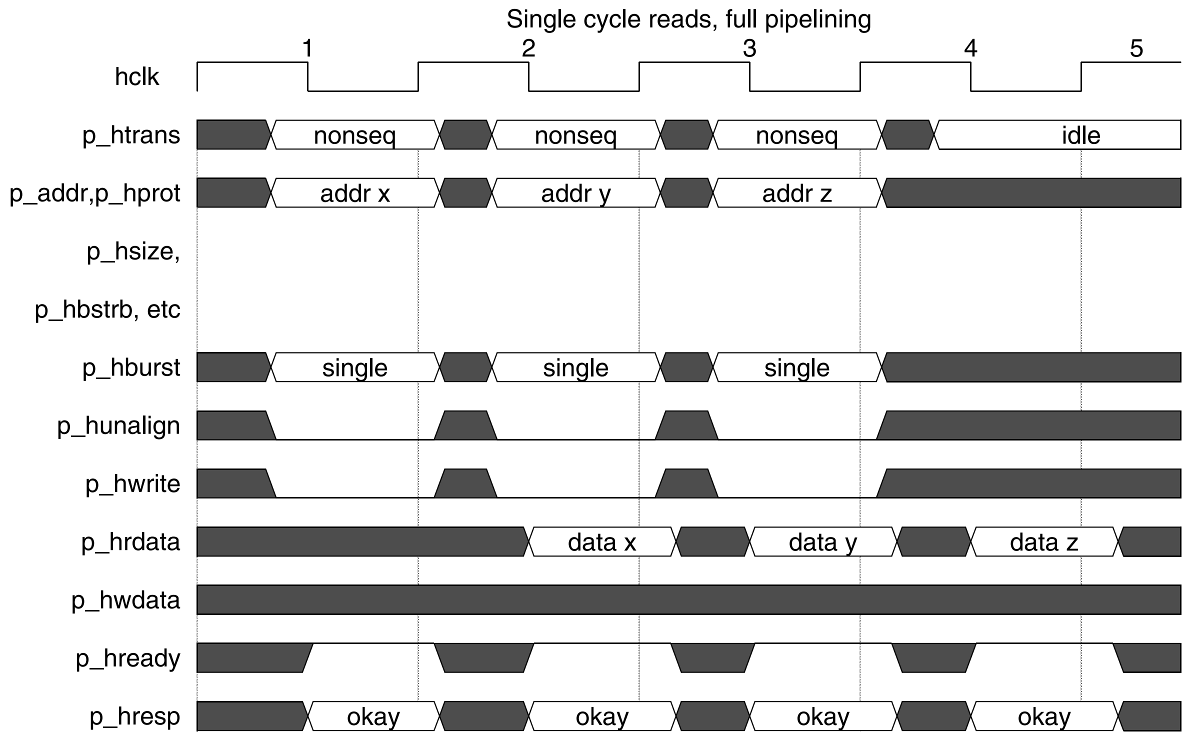


Figure 14-7. Basic read transfers

Clock 1 (C1)

The first read transfer starts in clock cycle 1. During C1, the core places valid values on the address bus and transfer attributes. The burst type (**p_hburst[2:0]**), protection control (**p_hprot[5:0]**), and transfer type (**p_htrans[1:0]**) attributes identify the specific access type. The transfer size attributes (**p_hsize[1:0]**) indicates the size of the transfer. The byte strobes (**p_hbstrb[7:0]**) are driven to indicate active byte lanes. The write (**p_hwrite**) signal is driven low for a read cycle.

The core asserts transfer request (**p_htrans=** NONSEQ) during C1 to indicate that a transfer is being requested. Since the bus is currently idle, (0 transfers outstanding), the first read request to $addr_x$ is considered *taken* at the end of C1. The default slave drives 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 one or more bytes of memory. Read data from the slave device is provided on the **p_hrdata** inputs. The slave device responds by asserting **p_hready** to indicate the cycle is completing and drives an OKAY response.

Another read transfer request is made during C2 to $addr_y$ (**p_htrans** = NONSEQ), and since the access to $addr_x$ is completing, it is considered *taken* at the end of C2.

Clock 3 (C3):

During C3, the addr$_y$ memory access takes place using the address and attribute values that were driven during C2 to enable reading of one or more bytes of memory. Read data from the slave device for addr$_y$ is provided on the **p_hrdata** inputs. The slave device responds by asserting **p_hready** to indicate the cycle is completing and drives an OKAY response.

Another read transfer request is made during C3 to addr$_z$ (**p_htrans** = NONSEQ), and since the access to addr$_y$ is completing, it is considered *taken* at the end of C3.

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.

The CPU has no more outstanding requests, so **p_htrans** indicates IDLE. The address and attribute signals are thus undefined.

14.3.2.2 Read transfer with wait state

Figure 14-8 shows an example of wait state operation. Signal **p_hready** for the first request (addr$_x$) is not asserted during C2, so a wait state is inserted until **p_hready** is recognized (during C3).

Meanwhile, a subsequent request has been generated by the CPU for addr$_{y,}$ which is not *taken* in C2, since the previous transaction is still outstanding. The address and transfer attributes remain driven in cycle C3 and are taken at the end of C3 since the previous access is completing. Data for addr$_x$ and a ready/OKAY response is driven back by the slave device. In cycle C4, a request for addr$_z$ is made. The request for access to addr$_z$ is taken at the end of C4, and during C5, the data and a ready/OKAY response is provided by the slave device. In cycle C5, no further accesses are requested.

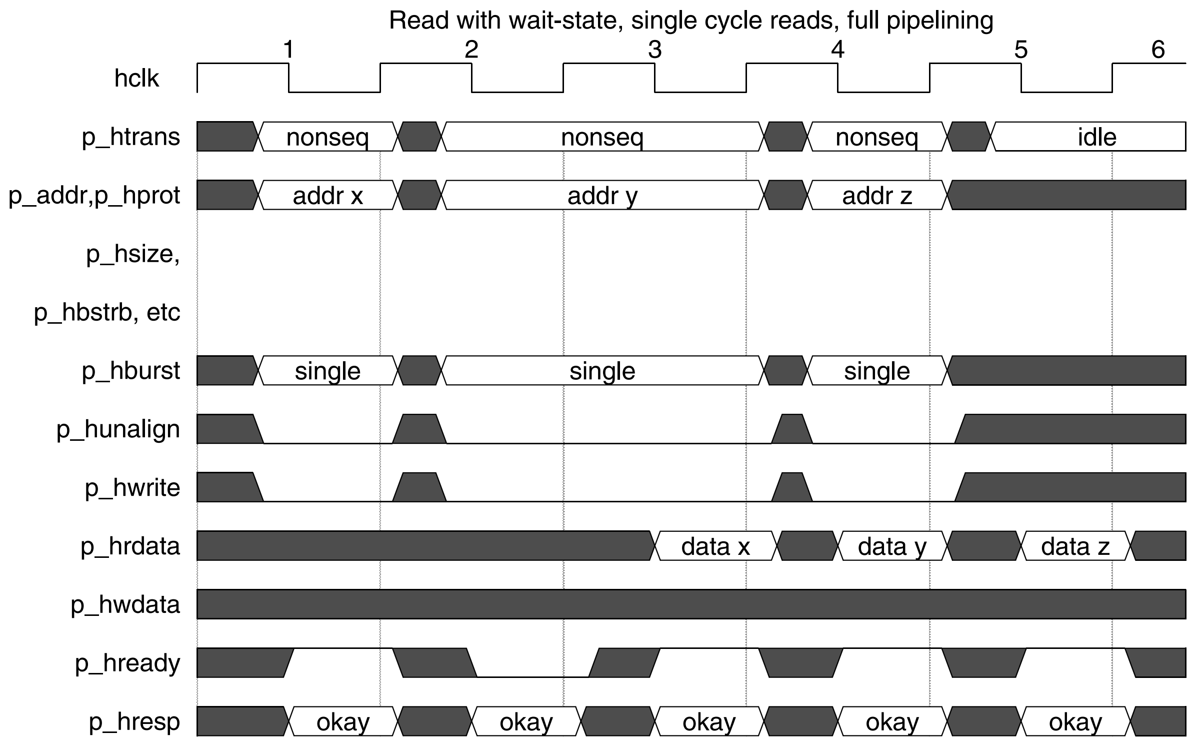


Figure 14-8. Read transfer with wait state

14.3.2.3 Basic write transfer cycles

During a write transfer, the core provides write data to a memory or peripheral device. Figure 14-9 illustrates functional timing for basic write transfers. Clock-by-clock descriptions of activity in Figure 14-9 follows:

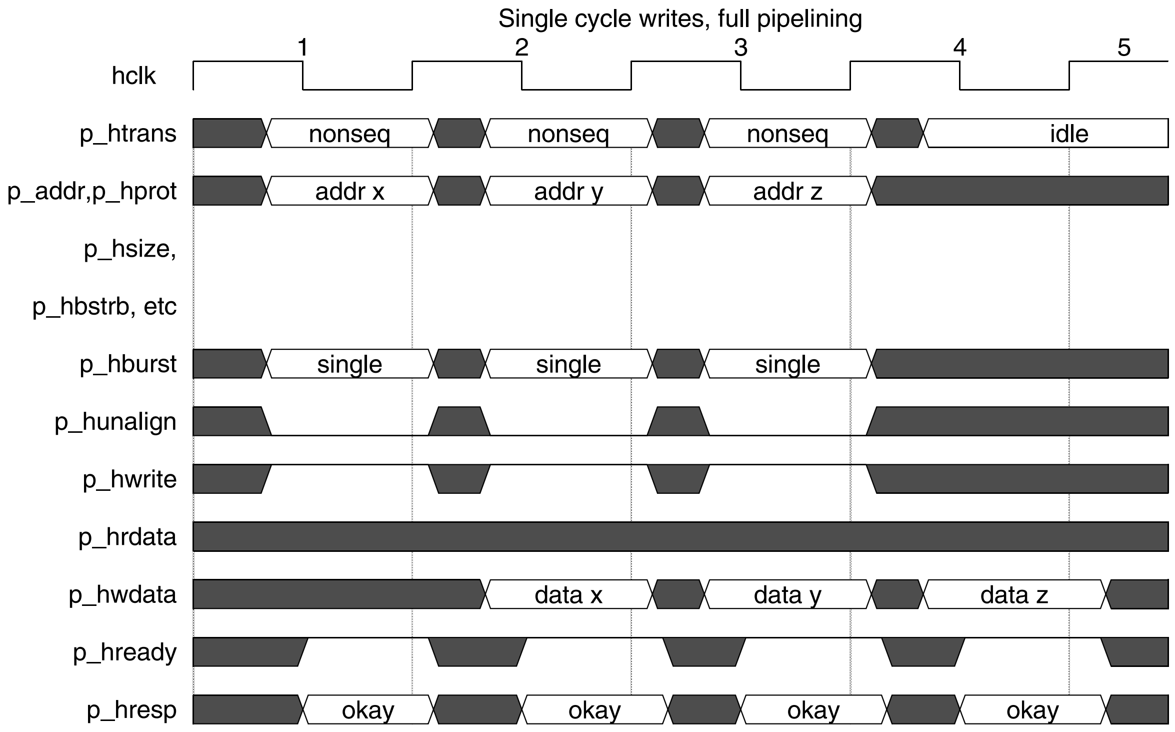


Figure 14-9. Basic write transfers

Clock 1 (C1)

The first write transfer starts in clock cycle 1. During C1, the core places valid values on the address bus and transfer attributes. The burst type (**p_hburst[2:0]**), protection control (**p_hprot[5:0]**), and transfer type (**p_htrans[1:0]**) attributes identify the specific access type. The transfer size attributes (**p_hsize[1:0]**) indicates the size of the transfer. The byte strobes (**p_hbstrb[7:0]**) are driven to indicate active byte lanes. The write (**p_hwrite**) signal is driven high for a write cycle.

The core asserts transfer request (**p_htrans**= NONSEQ) during C1 to indicate that a transfer is being requested. Since the bus is currently idle, (0 transfers outstanding), the first write request to addr$_x$ is considered *taken* at the end of C1. The default slave drives an ready/OKAY response for the current idle cycle.

Clock 2 (C2):

During C2, the write data for the access is driven, and the addr$_x$ memory access takes place using the address and attribute values that were driven during C1 to enable writing of one or more bytes of memory. The slave device responds by asserting **p_hready** to indicate the cycle is completing and drives an OKAY response.

Another write transfer request is made during C2 to addr$_y$ (**p_htrans** = NONSEQ), and since the access to addr$_x$ is completing, it is considered *taken* at the end of C2.

Clock 3 (C3):

During C3, write data for addr$_y$ is driven, and the addr$_y$ memory access takes place using the address and attribute values that were driven during C2 to enable writing of one or more bytes of memory. The slave device responds by asserting **p_hready** to indicate the cycle is completing and drives an OKAY response.

Another write transfer request is made during C3 to addr$_z$ (**p_htrans** = NONSEQ), and since the access to addr$_y$ is completing, it is considered *taken* at the end of C3.

Clock 4 (C4):

During C4, write data for addr$_z$ is driven, and the addr$_z$ memory access takes place using the address and attribute values that were driven during C3 to enable writing of one or more bytes of memory. The slave device responds by asserting **p_hready** to indicate the cycle is completing and drives an OKAY response.

The CPU has no more outstanding requests, so **p_htrans** indicates IDLE. The address and attribute signals are thus undefined.

14.3.2.4 Write transfer with wait states

Figure 14-10 shows an example of write wait state operation. Signal **p_hready** for the first request (addr$_x$) is not asserted during C2, so a wait state is inserted until **p_hready** is recognized (during C3).

Meanwhile, a subsequent request has been generated by the CPU for addr$_y$, which is not *taken* in C2, since the previous transaction is still outstanding. The address, transfer attributes, and write data remain driven in cycle C3 and are taken at the end of C3 since a ready/OKAY response is driven back by the slave device for the previous access. In cycle C4, a request for addr$_z$ is made. The request for access to addr$_z$ is taken at the end of C4, and during C5, a ready/OKAY response is provided by the slave device. In cycle C5, no further accesses are requested.

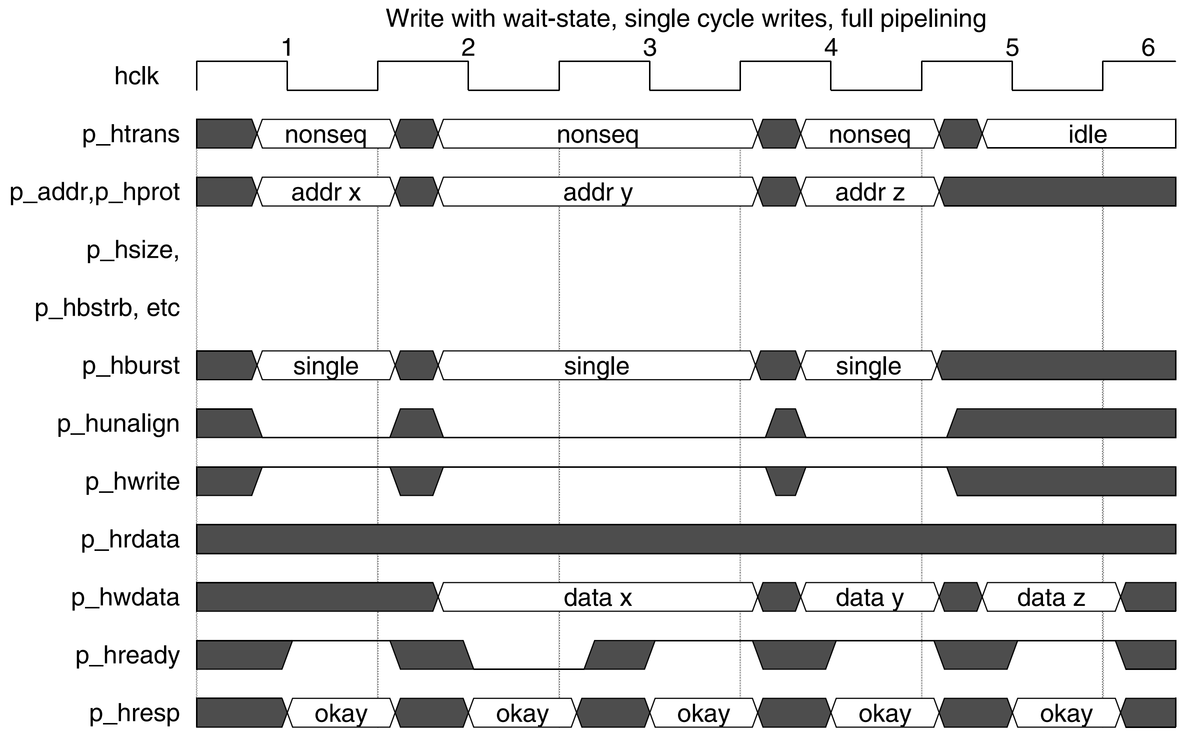


Figure 14-10. Write transfer with wait state

14.3.2.5 Read and write transfers

Figure 14-11 shows a sequence of read and write cycles.

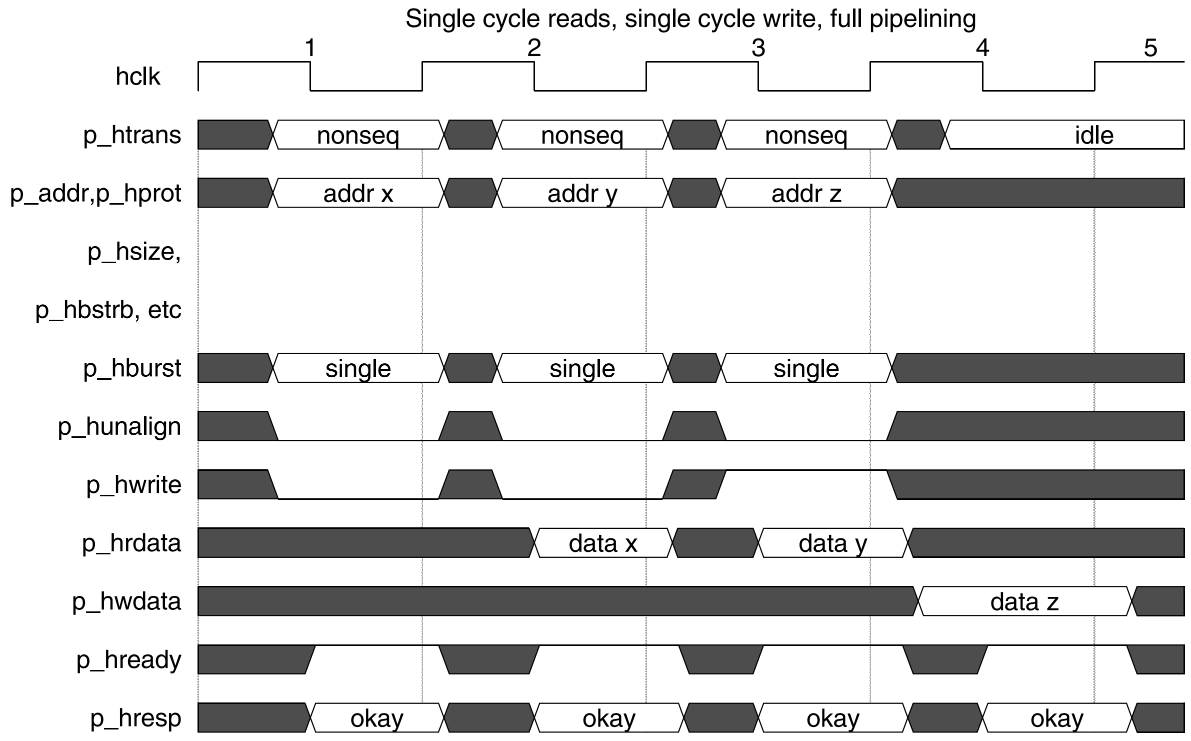


Figure 14-11. Single-cycle read and write transfers — 1

The first read request ($addr_x$) is *taken* at the end of cycle C1 since the bus is idle.

The second read request ($addr_y$) is *taken* at the end of C2 since a ready/OKAY response is asserted during C2 for the first read access ($addr_x$). During C3, a request is generated for a write to $addr_y$, which is taken at the end of C3 since the second access is terminating.

Data for the $addr_z$ write cycle is driven in C4, the cycle after the access is *taken*, and a ready/OKAY response is signaled to complete the write cycle to $addr_z$.

Figure 14-12 shows another sequence of read and write cycles. This example shows an interleaved write access between two reads.

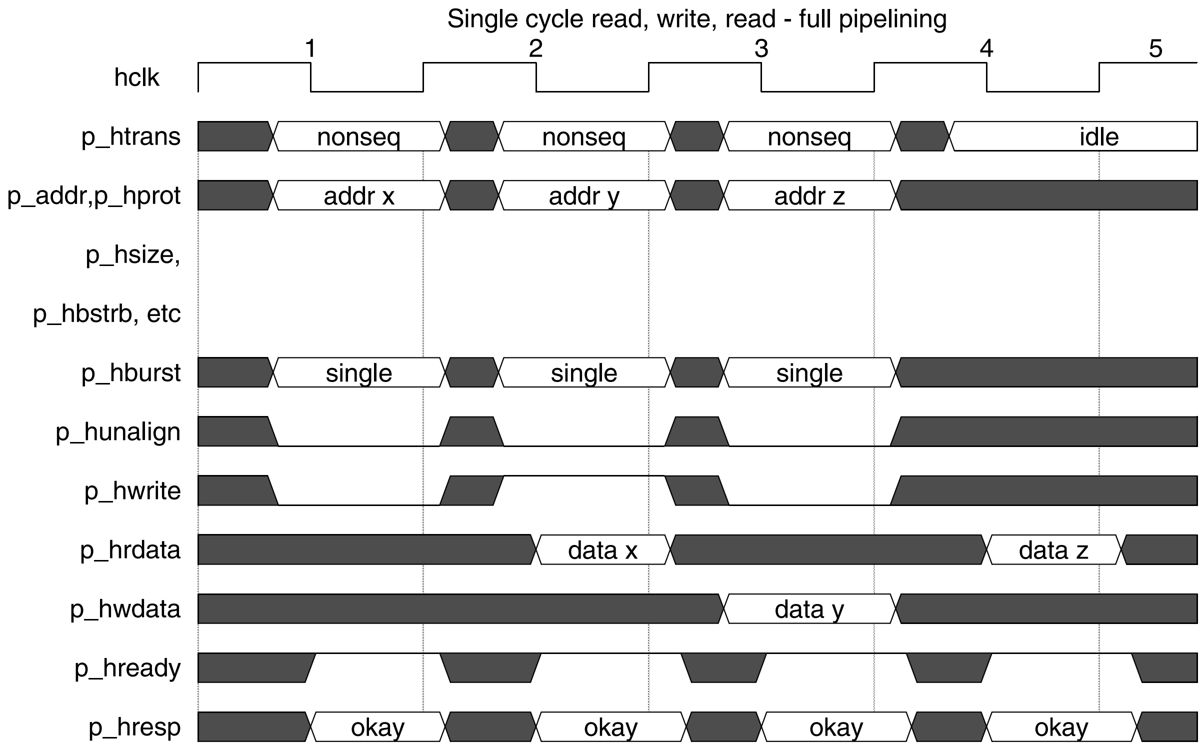


Figure 14-12. Single-cycle read and write transfers — 2

The first read request (addr_x) is *taken* at the end of cycle C1 since the bus is idle.

The first write request (addr_y) is *taken* at the end of C2 since 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 since the write access is terminating.

During C4, the addr_y write access is terminated, and no further access is requested.

Figure 14-13 shows another sequence of read and write cycles. In this example, reads incur a single wait state.

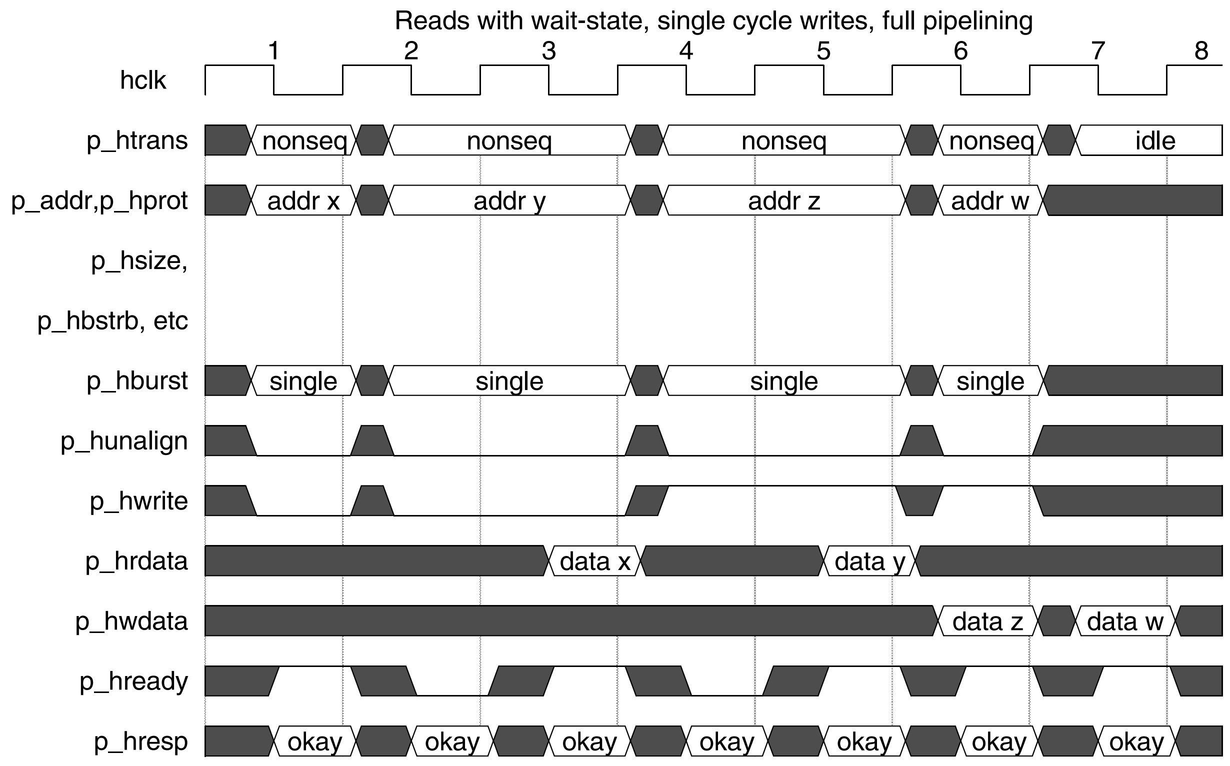


Figure 14-13. Multi-cycle read and write transfers — 1

The first read request (addr$_x$) is *taken* at the end of cycle C1 since the bus is idle.

The second read request (addr$_y$) is not *taken* at the end of cycle C2 since no ready response is signaled and only one access can be outstanding (addr$_x$). It is taken at the end of C3 once the first read request has signaled a ready/OKAY response.

The first write request (addr$_z$) is not taken during C4 since a ready response is not asserted during C4 for the second read access (addr$_y$). During C5, the request for a write to addr$_z$ is taken since the second access is terminating.

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 14-14 shows another sequence of read and write cycles. In this example, reads incur a single wait state.

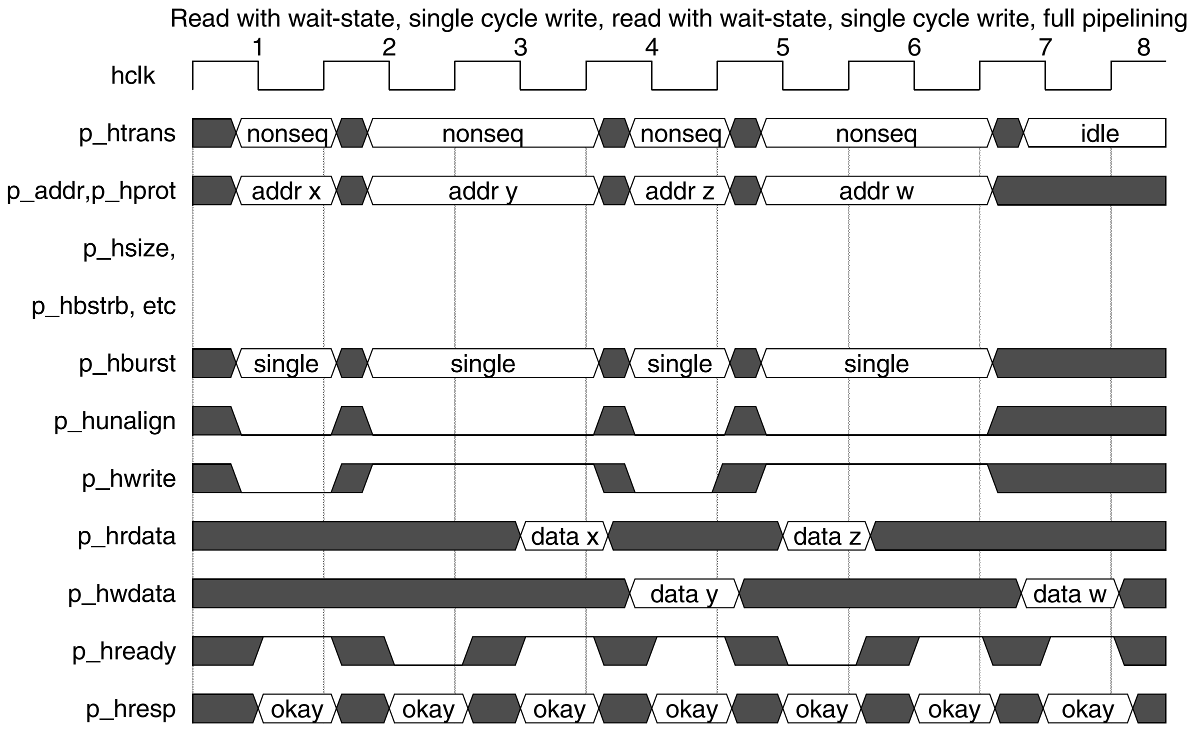


Figure 14-14. Multi-cycle read and write transfers — 2

The first read request (addr$_x$) is *taken* at the end of cycle C1 since the bus is idle.

The first write request (addr$_y$) is not *taken* at the end of cycle C2 since no ready response is signaled and only one access can be outstanding (addr$_x$). It is taken at the end of C3 once the first read request has signaled a ready/OKAY response.

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 since the addr$_y$ write is terminating.

A second write request (addr$_w$) is not taken at the end of C5 since the second read access is not terminating, thus it continues to drive the address and attributes into cycle C6.

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.

14.3.2.6 Misaligned accesses

Figure 14-15 illustrates functional timing for a misaligned read transfer. The read to addr$_x$ is misaligned across a 64-bit boundary.

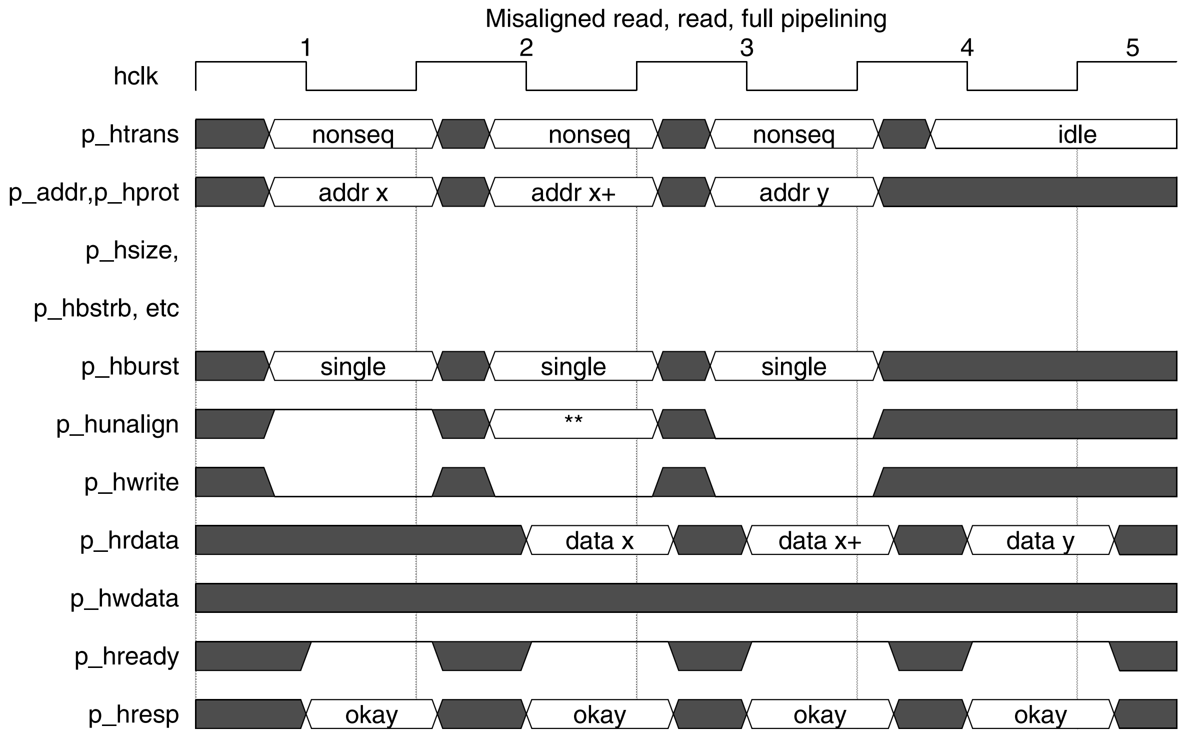


Figure 14-15. Misaligned read transfer

The first portion of the misaligned read transfer starts in C1. During C1, the core places valid values on the address bus and transfer attributes. The **p_hwrite** signal is driven low for a read cycle. The transfer size attributes (**p_hsize**) indicate the size of the transfer. Even though the transfer is misaligned, the size value driven corresponds to the size of the entire misaligned data item. **p_hunalign** is driven high to indicate that the access is misaligned. The **p_hbstrb** outputs are asserted to indicate the active byte lanes for the read, which may not correspond to size and low-order address outputs. **p_htrans** is driven to NONSEQ.

During C2, the $addr_x$ memory access takes place using the address and attribute values that were driven during C1 to enable reading of one or more bytes of memory.

The second portion of the misaligned read transfer request is made during C2 to $addr_{x+}$ (which will be aligned to the next higher 64-bit boundary), and since the first portion of the misaligned access is completing, it is *taken* at the end of C2. The p_htrans signals indicate NONSEQ. The size value driven is the size of the remaining bytes of data in the misaligned read, rounded up (for the 3-byte case) to the next higher power-of-2. The **p_hbstrb** signals indicate the active byte lanes. For the second portion of a misaligned transfer, the **p_hunalign** signal is driven high for the 3-byte case (low for all others). The next read access is requested in C3 and **p_htrans** indicates NONSEQ. **p_hunalign** is negated, since this access is aligned.

Figure 14-16 illustrates functional timing for a misaligned write transfer. The write to $addr_x$ is misaligned across a 64-bit boundary.

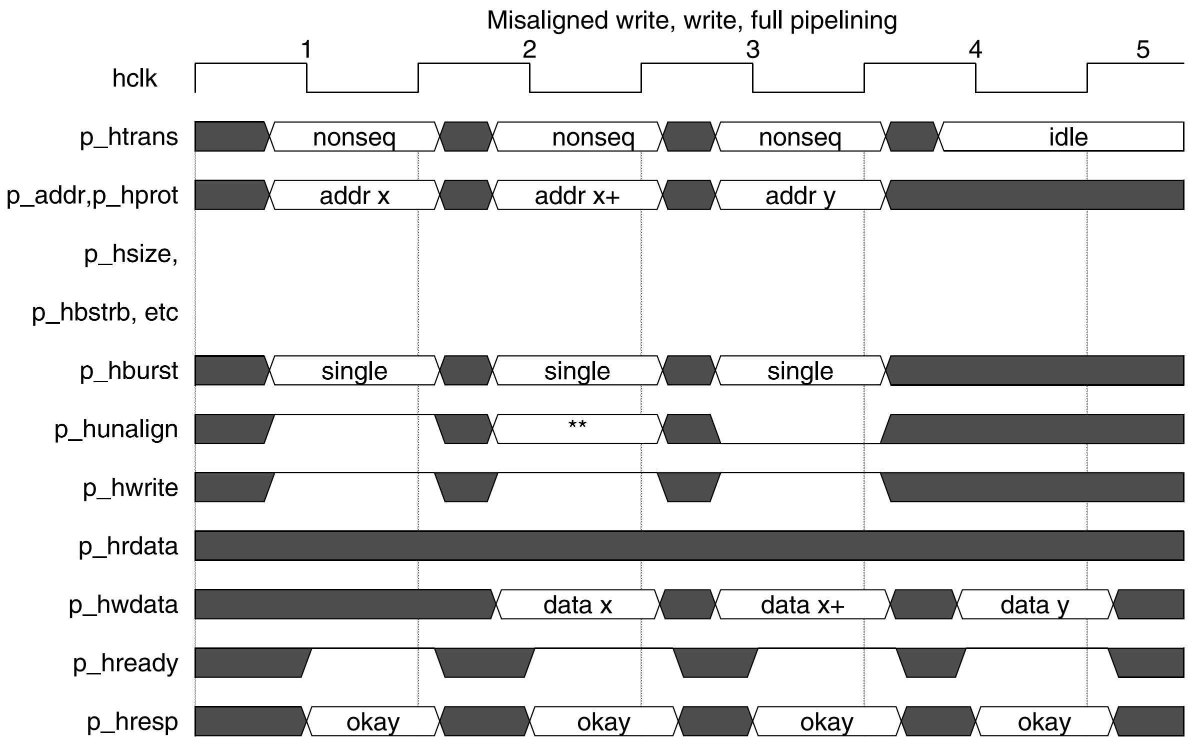


Figure 14-16. Misaligned write transfer

The first portion of the misaligned write transfer starts in C1. During C1, the core places valid values on the address bus and transfer attributes. The **p_hwrite** signal is driven high for a write cycle. The transfer size attribute (**p_hsize**) 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 will be aligned to the next higher 64-bit boundary), and since the first portion of the misaligned access is completing, it is *taken* at the end of C2. The p_htrans signals indicate NONSEQ. The size value driven is the size of the remaining bytes of data in the misaligned write, rounded up (for the 3-byte case) to the next higher power-of-2. The **p_hbstrb** signals indicate the active byte lanes. For the second portion of a misaligned transfer, the **p_hunalign** signal is driven high for the 3-byte case (low for all others).

The next write access is requested in C3 and **p_htrans** indicates NONSEQ. **p_hunalign** is negated, since this access is aligned.

An example of a misaligned write cycle followed by an aligned read cycle is shown in Figure 14-17. It is similar to the previous example in Figure 14-16.

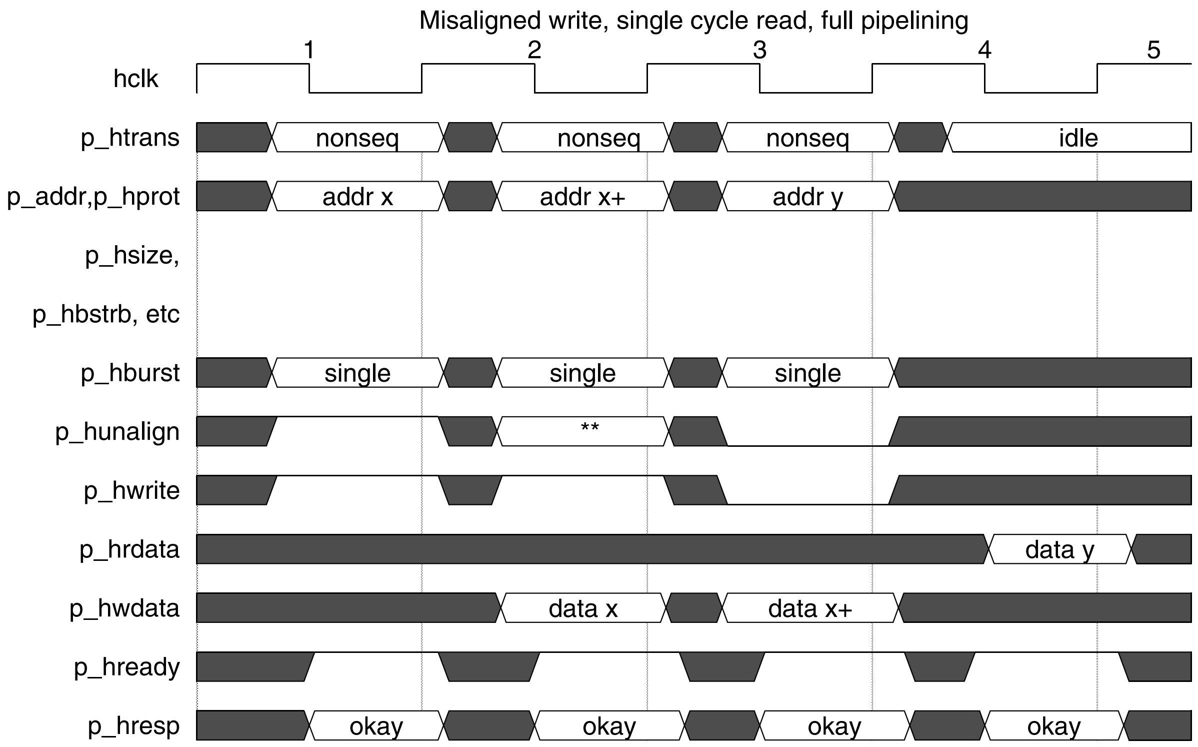


Figure 14-17. Misaligned write, single-cycle read transfer

14.3.2.7 Burst accesses

Figure 14-18 illustrates functional timing for a burst read transfer.

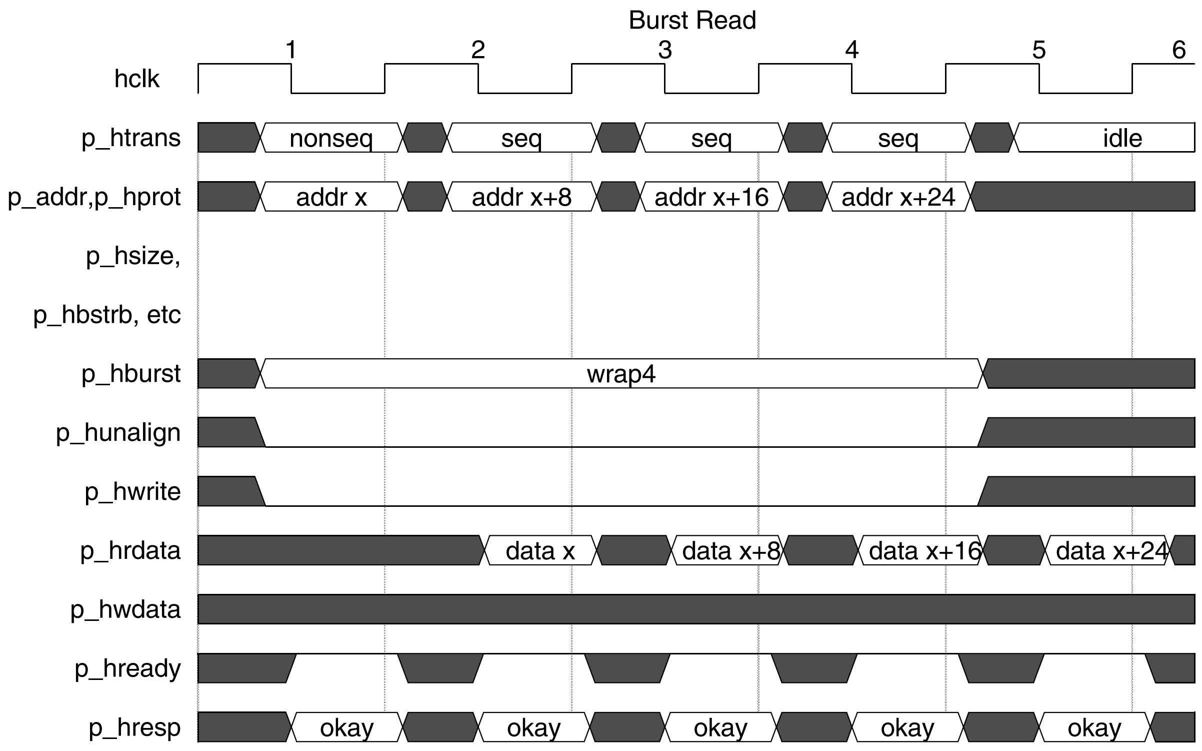


Figure 14-18. Burst read transfer

The p_hburst signals will indicate WRAP4 for all burst transfers. The **p_hunalign** signal will be negated. **p_hsize** will indicate 64-bits, and all eight **p_hbstrb** signals will be asserted. The burst address will be aligned to a 64-bit boundary and will wrap around modulo four doublewords. Note that in this example the **p_htrans** signal indicates IDLE after the last portion of the burst has been taken, but this is not always the case.

NOTE

Bursts may be followed immediately by any type of transfer. No idle cycle is required.

Figure 14-19 illustrates functional timing for a burst read with wait-state transfer.

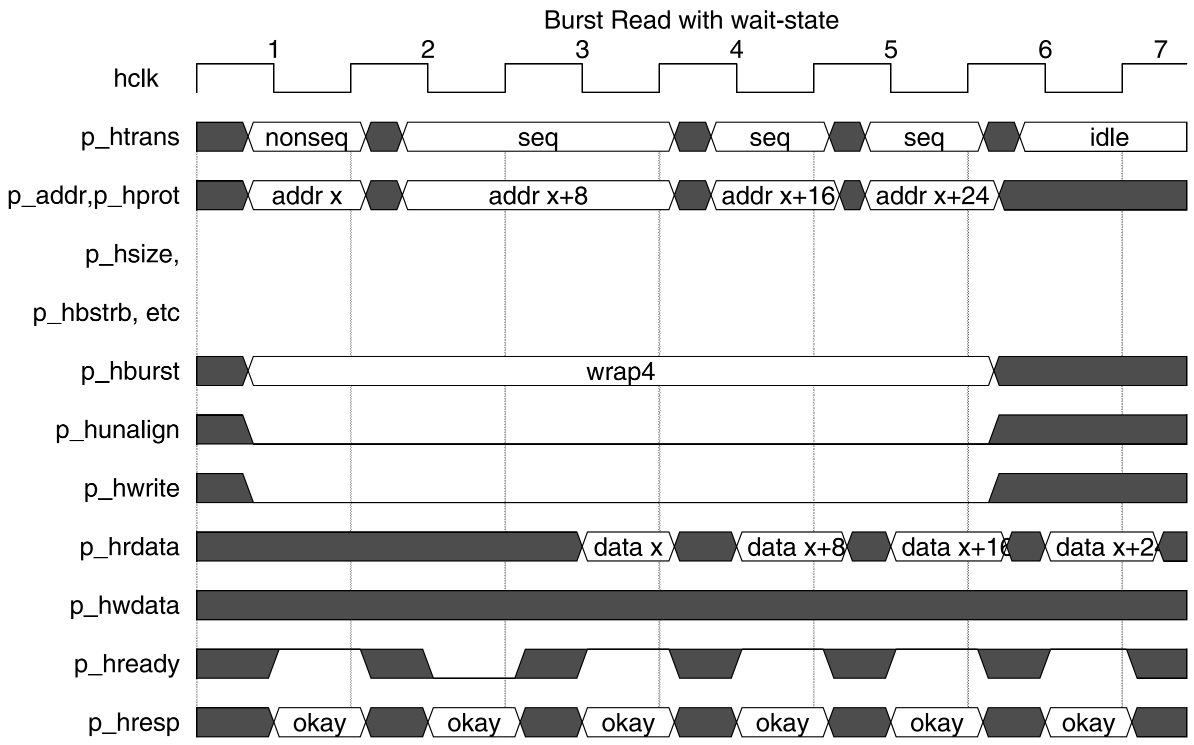


Figure 14-19. Burst read with wait-state transfer

The first cycle of the burst incurs a single wait-state.

Figure 14-20 illustrates functional timing for a burst write transfer.

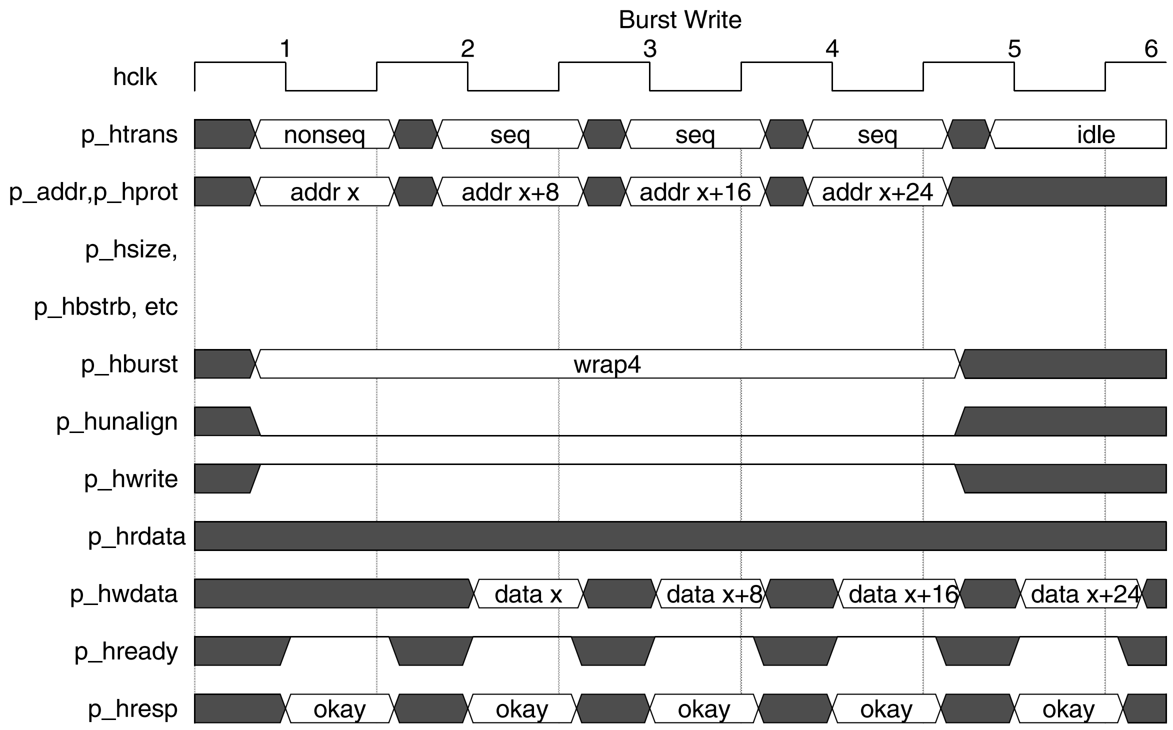


Figure 14-20. Burst write transfer

Figure 14-19 illustrates functional timing for a burst write with wait-state transfer.

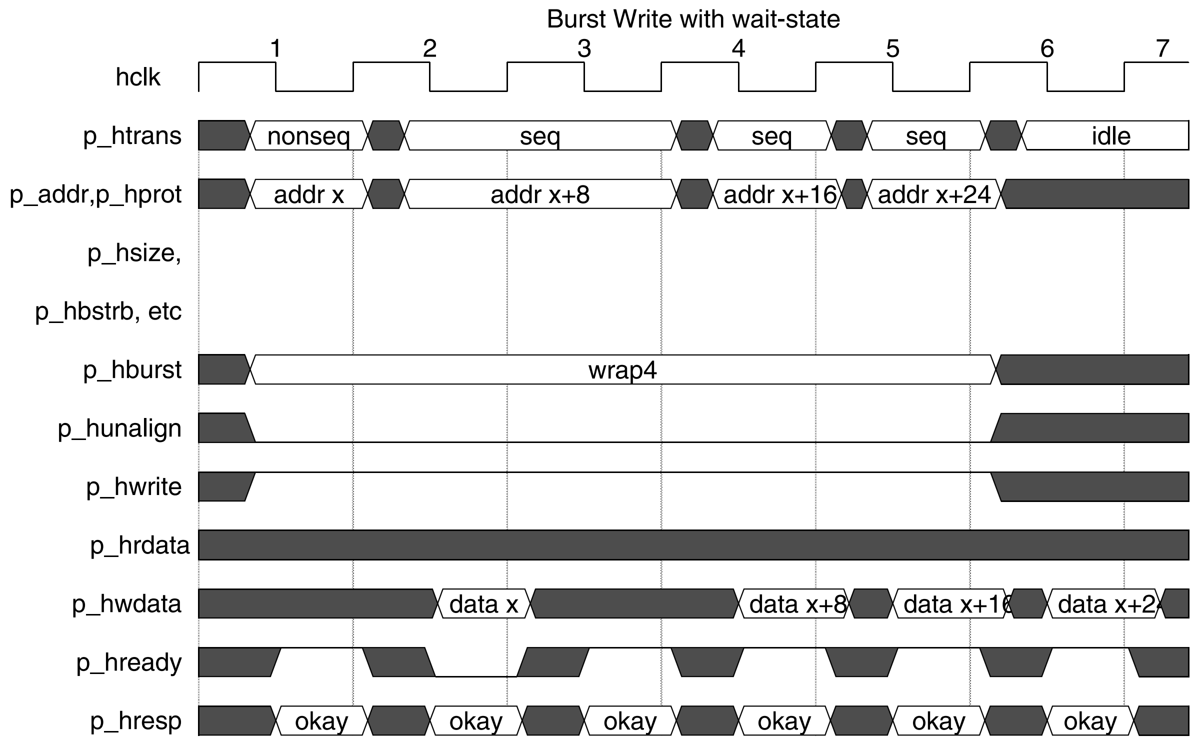


Figure 14-21. Burst write with wait-state transfer

The first cycle of the burst incurs a single wait-state. Data for the second beat of the burst is valid the cycle after the second beat is *taken*.

14.3.2.8 Error termination operation

The **p_hresp[2:0]** inputs are used to signal an error termination for an access in progress. The ERROR encoding is used in conjunction with the assertion of **p_hready** to terminate a cycle with error. Error termination is a two-cycle termination; the first cycle consists of signaling the ERROR response on **p_hresp[2:0]** while holding **p_hready** negated, and during the second cycle, asserting **p_hready** while continuing to drive the ERROR response on **p_hresp[2:0]**. This two cycle termination allows the BIU to retract a pending access if it desires to do so. **p_htrans** may be driven to IDLE during the second cycle of the two-cycle error response, or may change to any other value, and a new access unrelated to the pending access may be requested. The cycle 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 14-22 shows an example of error termination.

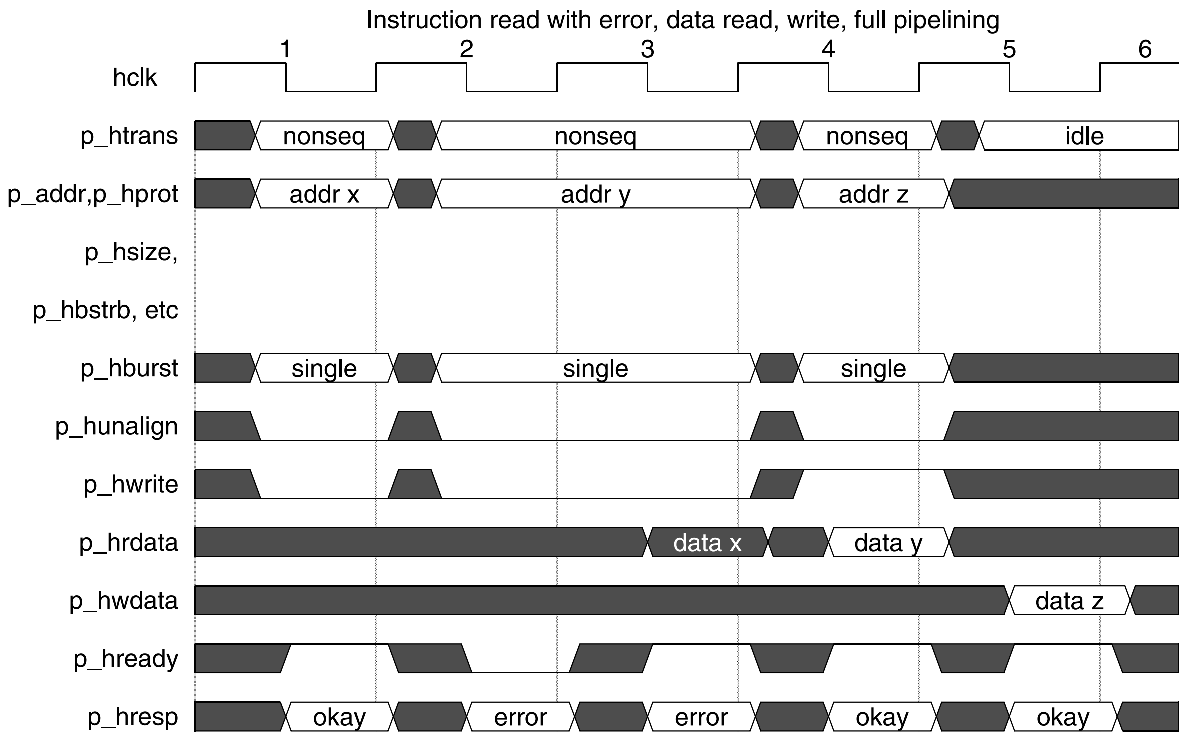


Figure 14-22. Read and write transfers, instruction read error termination

The first read request ($addr_x$) is *taken* at the end of cycle C1 since the bus is idle. It is an instruction prefetch.

The second read request ($addr_y$) is not *taken* at the end of C2 since the first access is still outstanding (no **p_hready** assertion). An error response is signaled by the addressed slave for $addr_x$ by driving ERROR onto the **p_hresp[2:0]** inputs. This is the first cycle of the two cycle error response protocol.

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 since 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 14-23 shows another example of error termination.

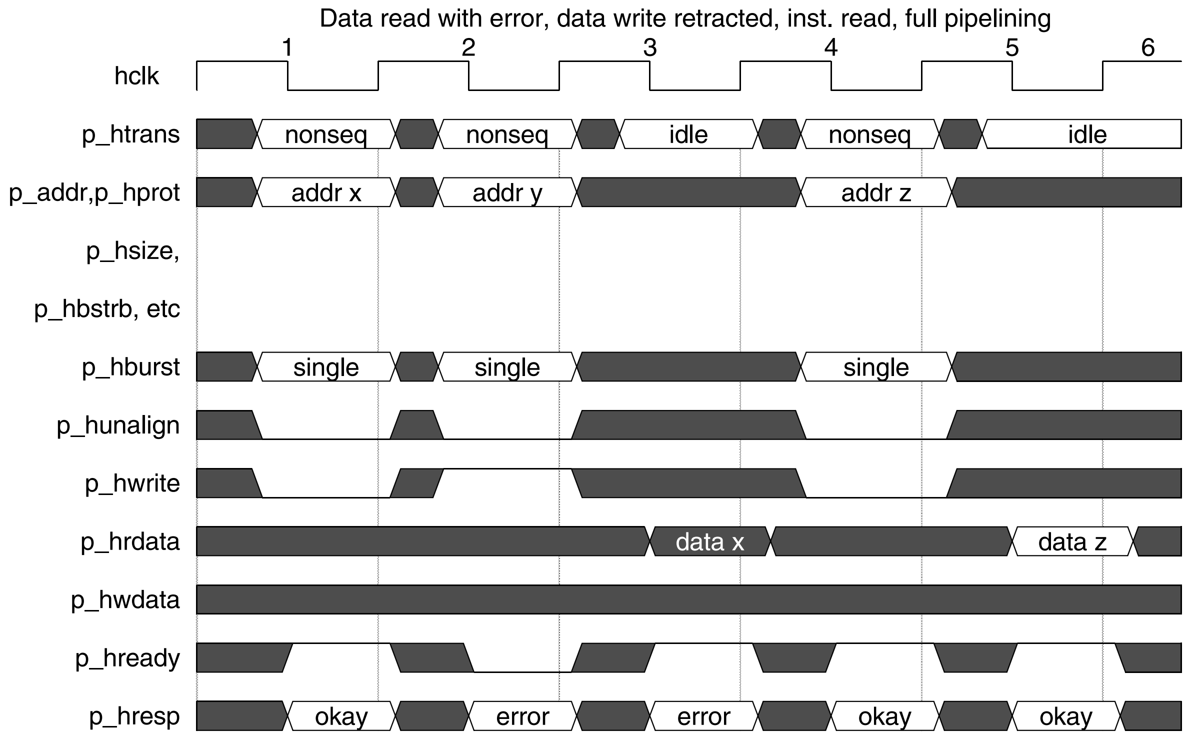


Figure 14-23. Data read error termination

The first read request (addr$_x$) is *taken* at the end of cycle C1 since the bus is idle. It is a data read.

The second request (write to addr$_y$) is not *taken* at the end of C2 since the first access is still outstanding (no **p_hready** assertion). An error response is signaled by the addressed slave for addr$_x$ by driving ERROR onto the **p_hresp[2:0]** inputs. This is the first cycle of the two cycle error response protocol.

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 the **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 14-24 shows another example of error termination, this time on the initial portion of a misaligned write.

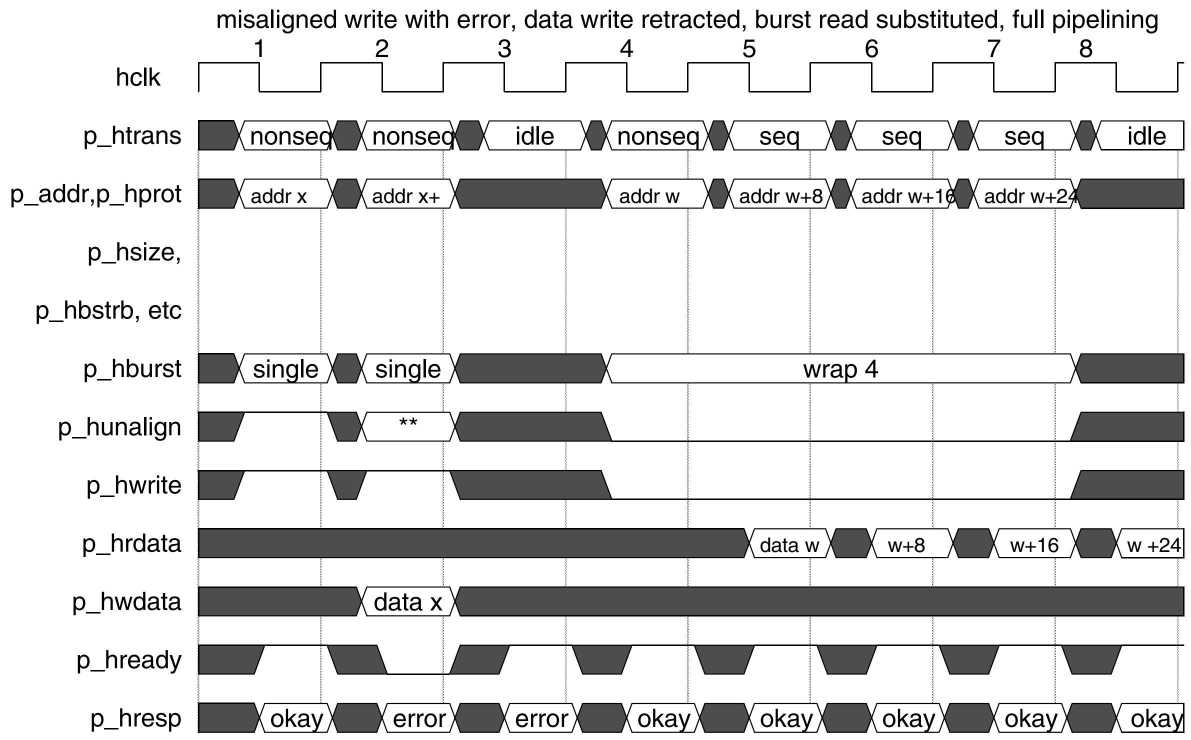


Figure 14-24. Misaligned write error termination, burst substituted

The first portion of the misaligned write request is terminated with error. The second portion is aborted by the CPU during the second cycle of the two cycle error response, and a subsequent burst read access to addr$_w$ becomes pending instead.

Figure 14-25 shows another example of error termination, this time on the initial portion of a burst read. The aborted burst is followed by a burst write.

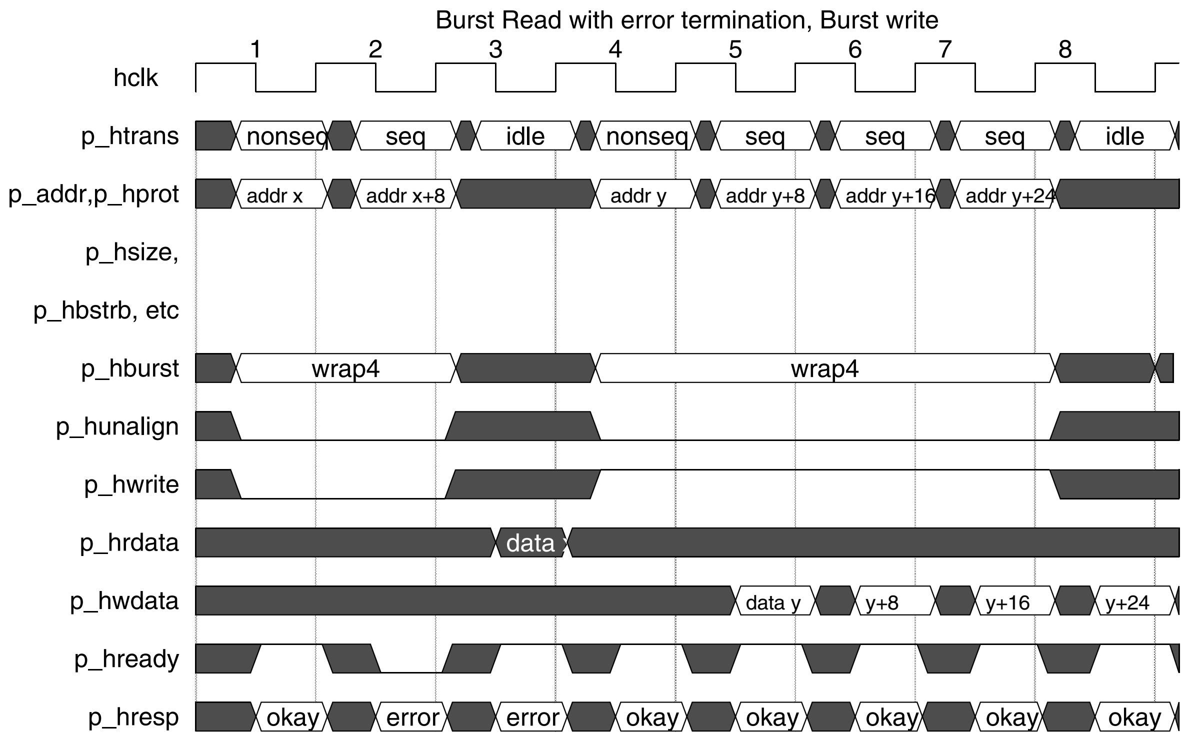


Figure 14-25. Burst read error termination, burst write substituted

The first portion of the burst read request is terminated with error. The second portion is aborted by the CPU during the second cycle of the two cycle error response, and a subsequent burst write access to addr$_y$ becomes pending instead.

14.3.3 Memory synchronization control operation

The memory synchronization signaling interface is provided to allow for synchronization operations initiated by execution of an **msync** or **mbar** (MO=0,1) to be signaled external to the CPU and to allow for handshaking of completion of the operations by other logic within the SoC. The interface provides a means for signaling that a synchronization operation should be performed, as well as controlling of the abort of an operation if a pending interrupt is detected by the CPU performing the synchronization instruction. This allows for minimization of interrupt latency while waiting for completion of the necessary operations required for performing the synchronization. Such an aborted operation will be reattempted once the interrupt handler has completed and the synchronization instruction is re-executed. In general, the synchronization operations involved flushing any pending stores from the CPU executing the msync or mbar to their final destinations (performing of pending store operations), which requires (at a minimum) flushing the store buffers of the initiating CPU, and flushing any pending snoop invalidation operations that were required by the operations performed by the initiating CPU prior to execution of the **msync** or **mbar** instruction. This may involve flushing of various store buffers and snoop queues interposed between elements of the coherency domain in the SoC, including coherency manager structures and other queues. The signals comprising the Memory Synchronization control interface are described in Section 14.2.10, Memory synchronization control signals. The following diagrams show examples of basic operation of the interface.

Figure 14-26 illustrates functional timing for an example memory synchronization operation. In the example shown, there are two CPUs in the system. In cycles 1 and 2, CPU0 decodes an **msync** instruction, suspends any further operand transfers, and flushes the internal push and store buffers to ensure the results of all previous store instructions have been made visible. After this activity completes, CPU0 asserts the **p_sync_req_out** output to signal to the SoC that a memory synchronization operation is to be performed. In the example, there are no intermediate buffers or queues in the SoC needing to be flushed, so the memory synchronization request input **p_sync_req_in** of CPU1 is asserted in cycle N, without additional delay that would be need if such queues and buffers existed and needed to be flushed in order for CPU1 to see any previously initiated memory operations performed. In cycle N+1, CPU1 begins flushing its internal snoop queue to process all pending snoop operations present at the time of the receipt of the **p_sync_req_in**. Following the processing of all of the snoop commands pending up to the point of the memory sync request, in cycle M, CPU1 responds by asserting its **p_sync_ack_out** output signal, which in this example is driven back to CPU0's **p_sync_ack_in** input in cycle M, which results in completion of the memory synchronization operation in cycle M+1. CPU0 negates the p_sync_req_out signal in cycle M+1, thus negating CPU1's p_sync_req_in signal. In response, in cycle M+2, CPU1 negates p_sync_ack_out. Note that in this simple example, the corresponding inputs and outputs of CPU0 and CPU1 are tied together, however in many systems, this handshaking sequence will be controlled by intermediary logic such as a cache coherency manager responsible for the correct directing of synchronization operations to the proper participants.

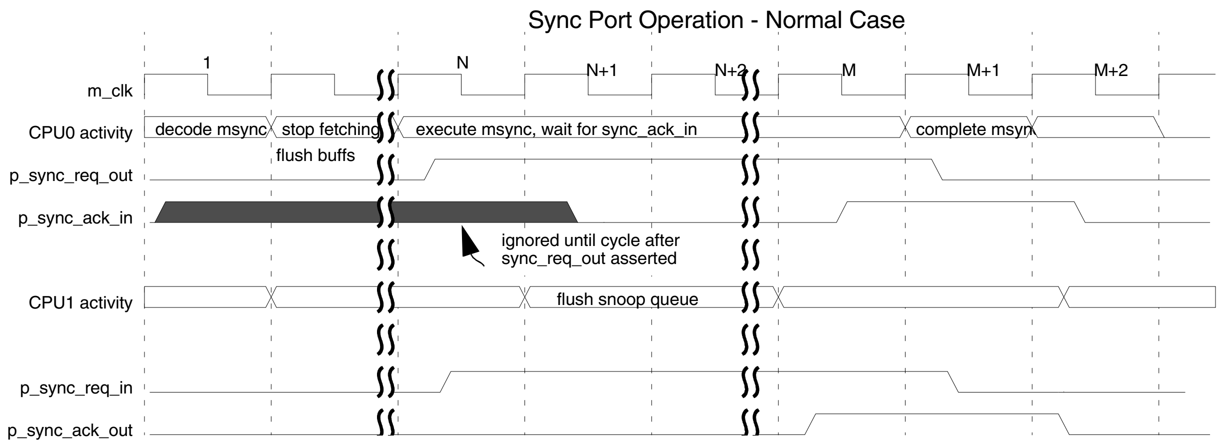


Figure 14-26. Memory sync operation — basic operation

Figure 14-27 illustrates functional timing for an example memory synchronization operation that is interrupted by a pending interrupt request. In cycles 1 and 2, CPU0 decodes an **msync** instruction, suspends any further operand transfers, and flushes the internal push and store buffers to ensure the results of all previous store instructions have been made visible. After this activity completes, CPU0 asserts the **p_sync_req_out** output in cycle N to signal to the SoC that a memory synchronization operation is to be performed. In cycle N however, an interrupt becomes pending in CPU0. In cycle N+1, CPU1 begins flushing its internal snoop queue to process all pending snoop operations present at the time of the receipt of the **p_sync_req_in**, however, in this cycle, CPU0 negates the **p_sync_request_out** output prior to receiving a **p_sync_ack_in** completion handshake, and aborts the msync instruction. In cycle N+1, CPU1's **p_sync_req_in** signal is negated in response. In subsequent cycles, CPU0 beings interrupt exception processing, and CPU1 is free to either complete the flushing of the snoop queue, or to abort it and resume normal operation. After CPU0 completes the interrupt handler, it will re-initiate the msync operation (not shown).

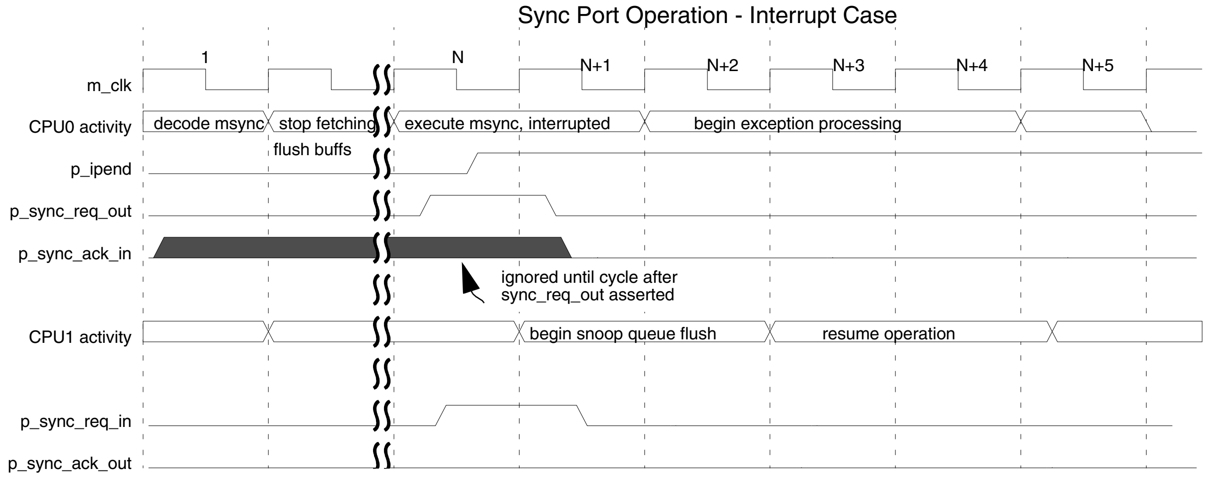


Figure 14-27. Memory sync operation — interruption operation

Figure 14-28 illustrates functional timing for another example memory synchronization operation. In this example, the snoop queue of CPU1 is empty, and the handshake completes earlier. This example is intended to show that there is no minimum time requirement between the assertion of p_sync_req_in, and the corresponding assertion of p_sync_ack_out, although not every implementation will respond this quickly.

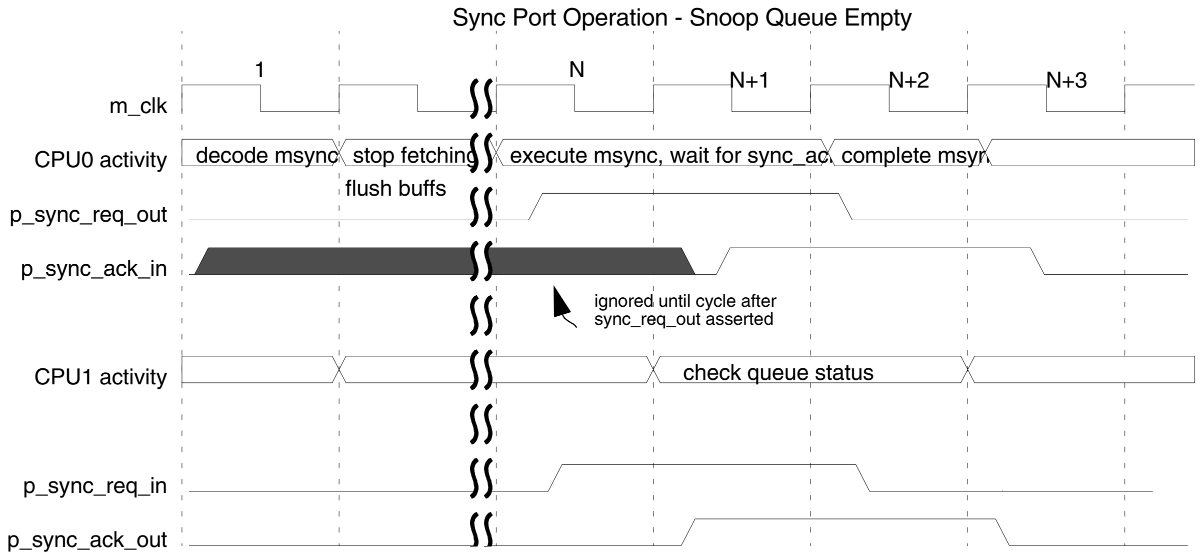


Figure 14-28. Memory sync operation — snoop queue empty

Figure 14-29 illustrates functional timing for another example memory synchronization operation. In this example, there are back-to-back sync instructions executed by CPU0. This example is intended to show that the **p_sync_req_out** output will transition for each individual synchronization request operation, with a minimum of one clock of negation interval between operation requests, and that because of the protocol on p_sync_ack_out assertion, p_sync_req_in must also negate and then reassert in order to request a second synchronization operation.

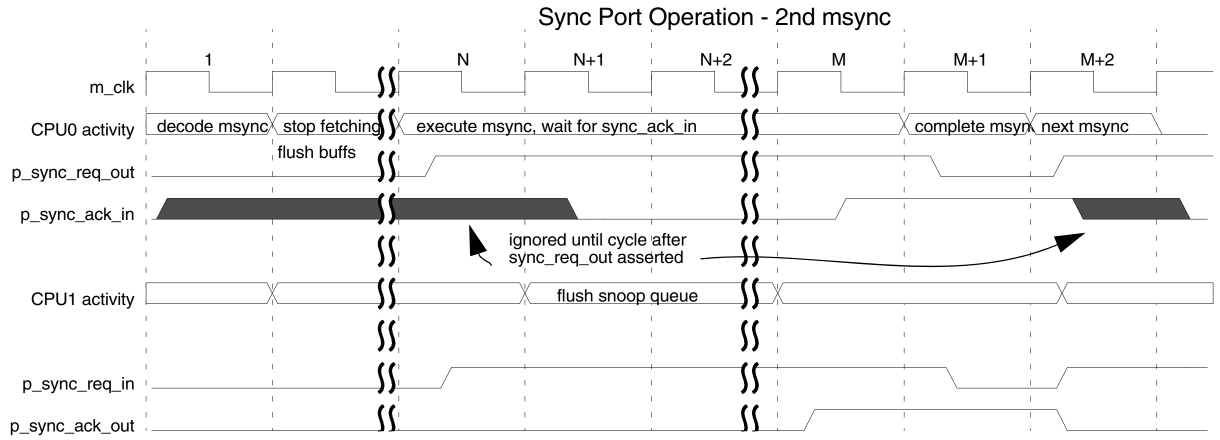


Figure 14-29. Memory sync operation — 2nd msync back-to-back

14.3.4 Cache coherency interface operation

The cache coherency signaling interface is provided to support hardware cache coherency operations by the e200z759n3.

Figure 14-30 illustrates functional timing for a set of basic snoop request operations. Snoop requests are presented in cycles 1, 2, and 3, and enter the snoop queue. As requests are processed, they are acknowledged. In this example, the snoops miss in the cache and require only a single cache access slot for lookup. The exact cycle the requests are acknowledged may vary, and are not directly related to the cycle the requests occur.

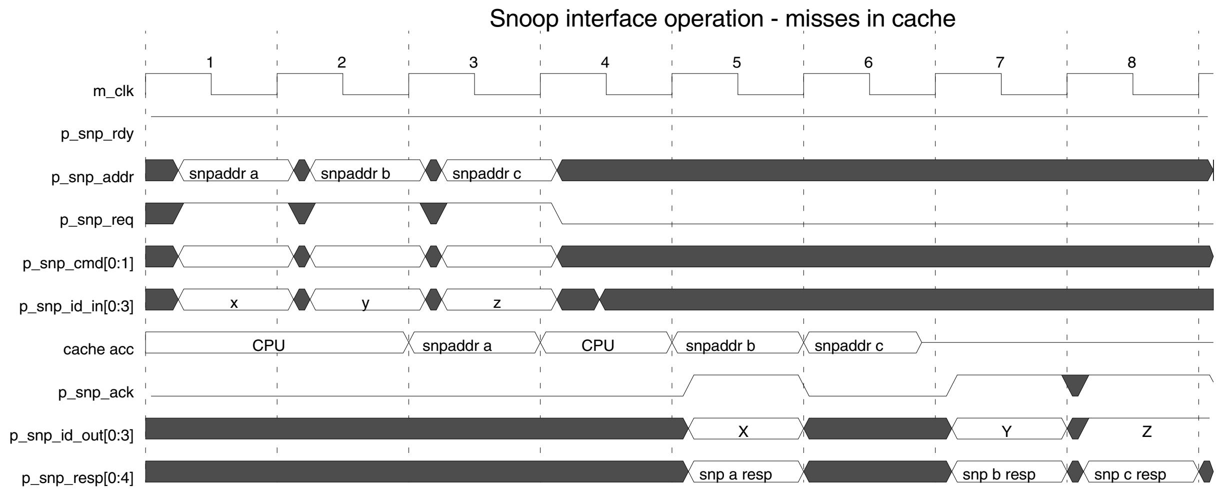


Figure 14-30. Basic cache coherency interface operation — misses

Figure 14-31 illustrates functional timing for a snoop hit with invalidate. The exact cycle the requests are acknowledged may vary, and are not directly related to the cycle the requests occur.

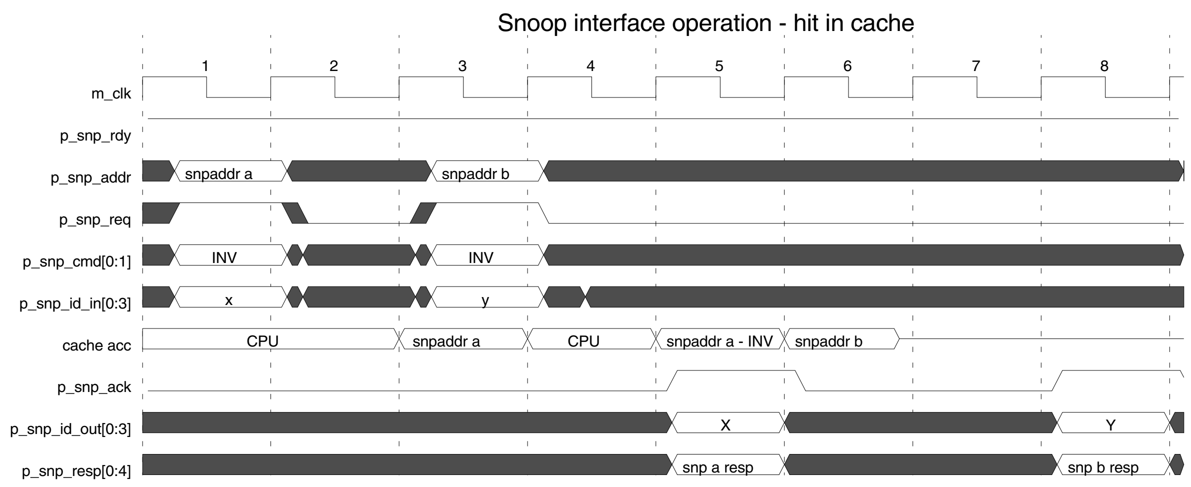


Figure 14-31. Basic cache coherency interface operation — hit

Figure 14-32 illustrates another example of timing for a snoop request. This example shows the starvation control for a snoop that sits in the snoop queue until the snoop starvation counter expires, due to blockage by a continuous stream of CPU requests.

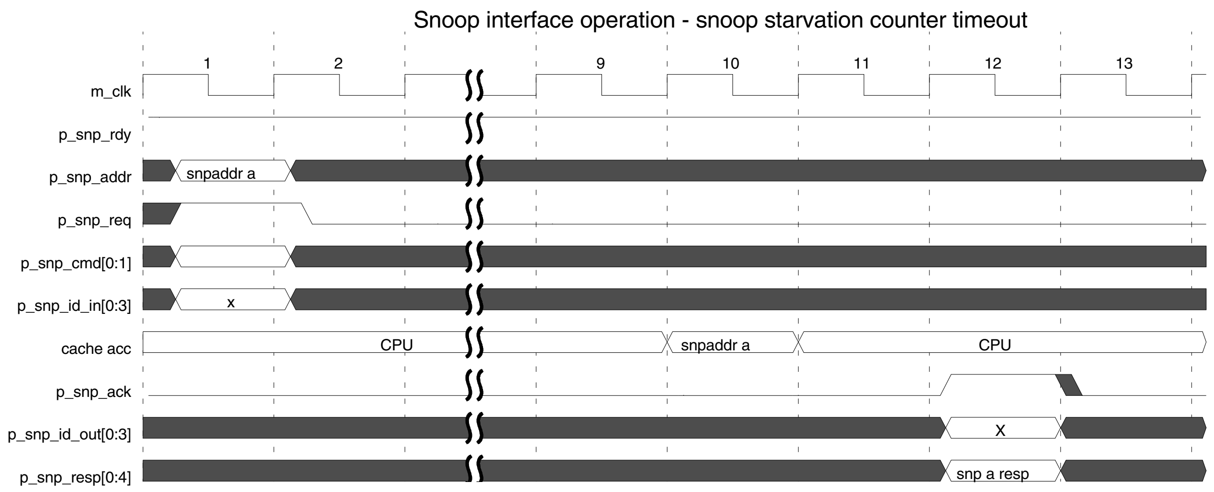


Figure 14-32. Cache coherency interface operation — snoop starvation timeout

Figure 14-33 illustrates operation of the **p_snp_rdy** output and snoop request acceptance. In this example, **p_snp_rdy** is initially asserted, but in cycle 1 is negated due to the snoop queue filling. A snoop request for snpaddr ‘a’ is asserted in cycle 1. This request is taken and entered into the snoop queue at the end of cycle 1. In cycle 2, **p_snp_rdy** is still negated, and a snoop request for snpaddr ‘b’ is presented. This request is also accepted and loaded into the snoop queue at the end of cycle 2, to allow for systems to use **p_snp_rdy** from one CPU as a control qualifier to drive the **p_stall_bus_gwrite** input control of another CPU. Following this, in cycle 3 another snoop request is presented for snpaddr ‘c’. This request is not accepted, and must remain pending until the cycle after **p_snp_rdy** re-asserts to be recognized. In cycle 5, **p_snp_rdy** is reasserted, indicating that the snoop queue can begin to store additional requests starting in the next cycle. Due to the protocol on **p_snp_rdy**, a minimum of two snoop queue entries must be available before **p_snp_rdy** can be re-asserted. Since a snoop request was pending at the end of cycle 4 (**p_snp_req** was asserted), the **p_snp_rdy** output will re-assert for one cycle once two free queue entries are available. The request for snpaddr 'c' will be queued at the end of cycle 6. In cycle 6, **p_snp_rdy** is again negated, due to limited available snoop queue entries. This negation occurs in cycle 6 since **p_snp_rdy** was asserted during cycle 5 with only two free entries in the queue. When no pending snoop request is presented (**p_snp_req** is negated), **p_snp_rdy** will not be re-asserted until three queue entries are available. This is so that the **p_snp_rdy** signal does not alternate between asserted and negated, which must happen when only two queue entries are available. The re-assertion of **p_snp_rdy** in cycle 5 allows the pending request for snpaddr 'c' to be accepted at the end of cycle 6. A new snoop request to snpaddr 'd' is made in cycle 7, and is accepted even though **p_snp_rdy** was negated in cycle 6, according to the **p_snp_rdy** protocol. A subsequent snoop request to snpaddr 'e' presented in cycle 8 must remain pending to be accepted until **p_snp_rdy** re-asserts after two free queue entries are once again available. Note that in cycles 5 and 6, earlier snoop requests to snpaddr 'm' and 'n' are processed, and the completion of these requests are signaled in cycles 7 and 8.

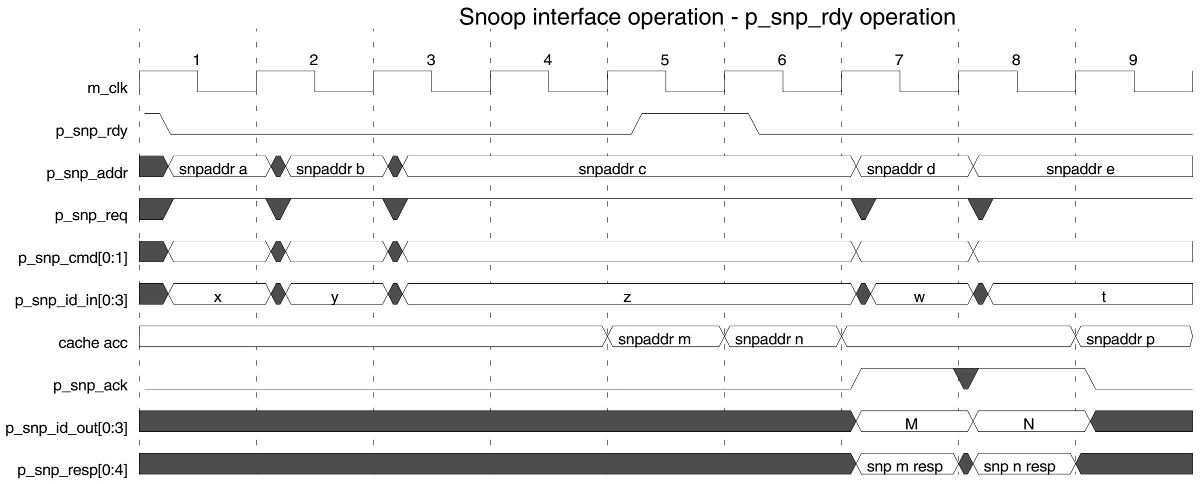


Figure 14-33. Cache coherency interface operation — p_snp_rdy operation

Figure 14-34 illustrates another example of operation of the **p_snp_rdy** output and snoop request acceptance. In this example, **p_snp_rdy** is initially asserted, but in cycle 1 is negated due to the snoop queue filling. A snoop request for snpaddr 'a' is asserted in cycle 1. This request is taken and entered into the snoop queue at the end of cycle 1. In cycle 2, **p_snp_rdy** is still negated, and a snoop request for snpaddr 'b' is presented. This request is also accepted and loaded into the snoop queue at the end of cycle 2, to allow for systems to use **p_snp_rdy** from one CPU as a control qualifier to drive the **p_stall_bus_gwrite** input control of another CPU. Following this, in cycle 3 another snoop request is presented for snpaddr 'c'. This request is not accepted, and must remain pending until the cycle after **p_snp_rdy** re-asserts to be recognized. In cycle 5, **p_snp_rdy** is reasserted, indicating that the snoop queue can begin to store additional requests starting in the next cycle. Due to the protocol on **p_snp_rdy**, a minimum of two snoop queue entries must be available before **p_snp_rdy** can be re-asserted. Since a snoop request was pending at the end of cycle 4 (**p_snp_req** was asserted), the **p_snp_rdy** output will re-assert for one cycle once two free queue entries are available. In this example however, the request for snpaddr 'c' will not be queued at the end of cycle 6, since the request is no longer present. In cycle 6, **p_snp_rdy** negates, since three queue entries have not yet become available. In cycle 7, **p_snp_rdy** can be re-asserted indicating at least three queue entries are available, and in cycle 8, a new snoop request is presented and accepted for snpaddr 'x'. Note that in cycles 6 and 7, earlier snoop requests to snpaddr 'm' and ‘n’ are processed, freeing up the needed queue entries for the re-assertion of **p_snp_rdy**, and the completion of these requests are signaled in cycles 8 and 9.

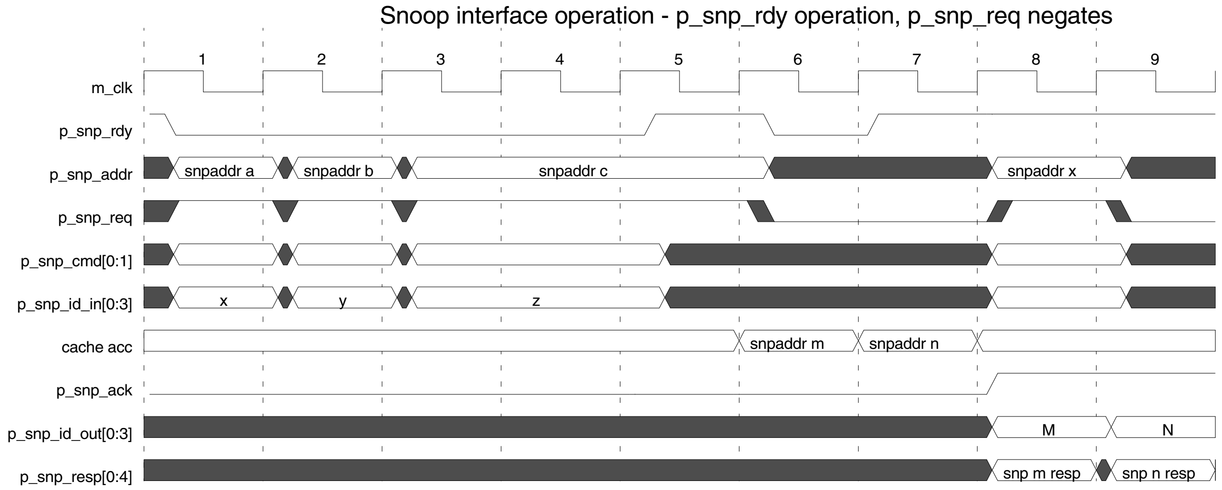


Figure 14-34. Cache coherency interface operation — p_snp_rdy operation, p_snp_req negation prior to acceptance

Figure 14-35 illustrates another example of operation of the **p_snp_rdy** output and snoop request acceptance. In this example, **p_snp_rdy** is initially asserted, but in cycle 1 is negated due to the snoop queue filling. A snoop request for snpaddr ‘a’ is asserted in cycle 1. This request is taken and entered into the snoop queue at the end of cycle 1. In cycle 2, **p_snp_rdy** is still negated, and a snoop request for snpaddr ‘b’ is presented. This request is also accepted and loaded into the snoop queue at the end of cycle 2, to allow for systems to use **p_snp_rdy** from one CPU as a control qualifier to drive the **p_stall_bus_gwrite** input control of another CPU. Following this, in cycle 3 another snoop request is presented for snpaddr ‘c’. This request is not accepted, and must remain pending until the cycle after **p_snp_rdy** re-asserts to be recognized. In cycle 5, **p_snp_rdy** is reasserted, indicating that the snoop queue can begin to store additional requests starting in the next cycle. Due to the protocol on **p_snp_rdy**, a minimum of two snoop queue entries must be available before **p_snp_rdy** can be re-asserted. Since a snoop request was pending at the end of cycle 4 (**p_snp_req** was asserted), the **p_snp_rdy** output will re-assert for one cycle once two free queue entries are available. In this example however, the request for snpaddr ‘c’ will not be queued at the end of cycle 6, since the request is no longer present. In cycle 6, **p_snp_rdy** negates, since three queue entries have not yet become available. In cycle 8, a new snoop request is presented and accepted for snpaddr ‘x’. Note that since the **p_snp_rdy** output was asserted in cycle 5, a snoop request present in ether or both of cycles 6 and 7 will be accepted as per the protocol.

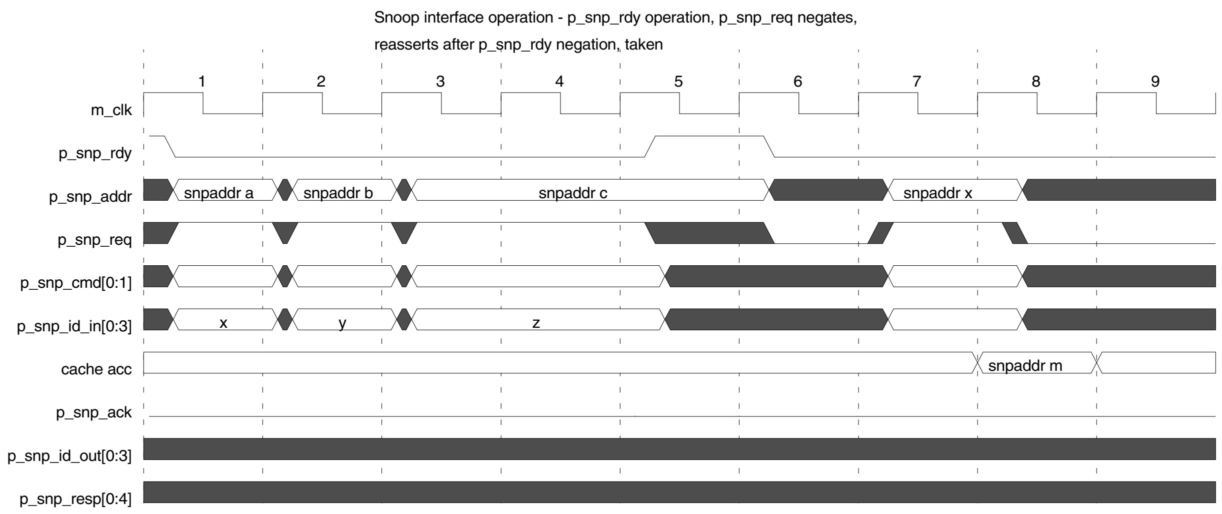


Figure 14-35. p_snp_rdy operation, p_snp_req negation prior to acceptance, reasserted later in ready window

14.3.4.1 Stop mode entry/exit and snoop ready signaling

When a request is made to enter stop mode via the assertion of **p_stop**, the **p_snp_rdy** output will be negated. While the core complex is in the Stopped (power-down) state, bus snooping is disabled, and the **p_snp_rdy** output is held negated. Snoop requests will be processed around the assertion of the stop mode entry request (assertion of **p_stop**) per the normal protocol associated with **p_snp_rdy** negation, including acceptance of a snoop request during a small interval around **p_snp_rdy** negation, thus additional snoop operations may need to occur prior to entering the stopped state. All snoop queue entries will be processed prior to the assertion of **p_stopped**.

Figure 14-36 illustrates an example of operation of the **p_snp_rdy** output when entering the Stopped state and snoop request acceptance. In cycle 1, p_stop is asserted, indicating a request to enter the stopped state. In cycle 2 the **p_snp_rdy** signal negates due to the stop request. Snoop requests for snpaddr 'a' and 'b' are taken in cycles 2 and 3 according to the **p_snp_rdy** protocol, although the system logic should typically stop generating new requests based on the **p_stop** input assertion. The request for snpaddr 'c' is not taken in cycle 4, again based on the snoop ready protocol. In cycle(s) 5, the snoop control logic continues to process any previously queued snoop requests, and in cycle N, and N+1, the final snoop responses for snoops A and B occur. Following the snoop responses for these final queued snoop requests, **p_stopped** asserts in cycle N+2. No further snoop requests will be accepted while the CPU is stopped.

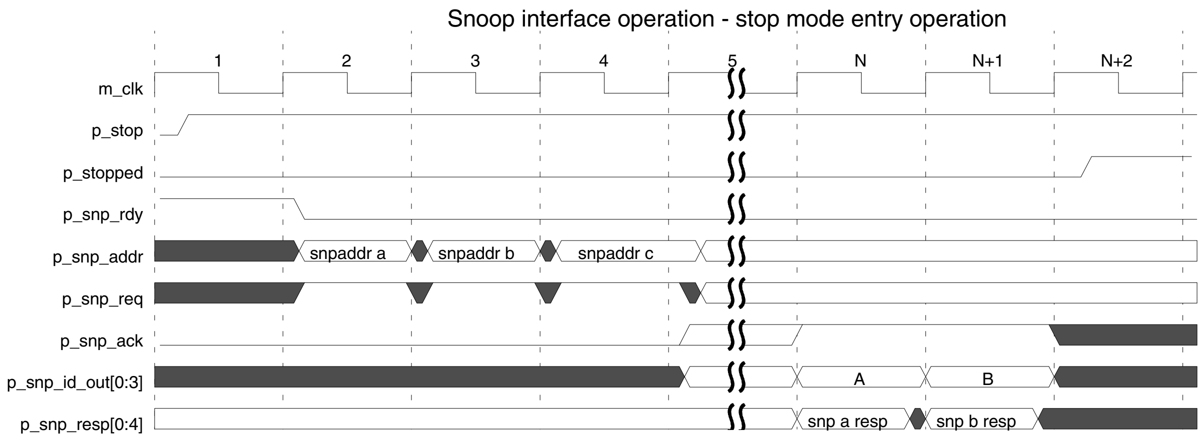


Figure 14-36. Stop mode entry, p_snp_rdy operation

Figure 14-37 illustrates an example of operation of the **p_snp_rdy** output when exiting the Stopped state and snoop request acceptance. In cycle 1, p_stop is negated, indicating a request to exit the stopped state. In cycle 2, the **p_stopped** output signal negates due to the negated stop request. Also in cycle 2, the p_snp_rdy output is asserted, indicating that snoop requests will begin to be accepted on the next clock cycle. Snoop requests for snpaddr 'a' and 'b' are taken in cycles 3 and 4 according to the **p_snp_rdy** protocol. In cycle N and N+1, the snoop responses for snoops A and B occur.

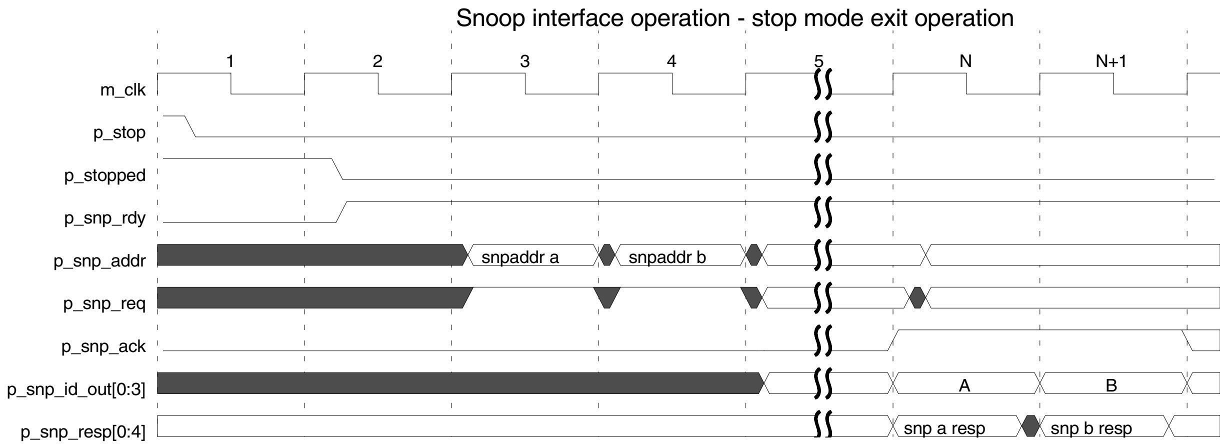


Figure 14-37. Stop mode exit, p_snp_rdy operation

14.3.5 Power management

The following diagram shows the relationship of the wakeup control signal **p_wakeup** to the relevant input signals.

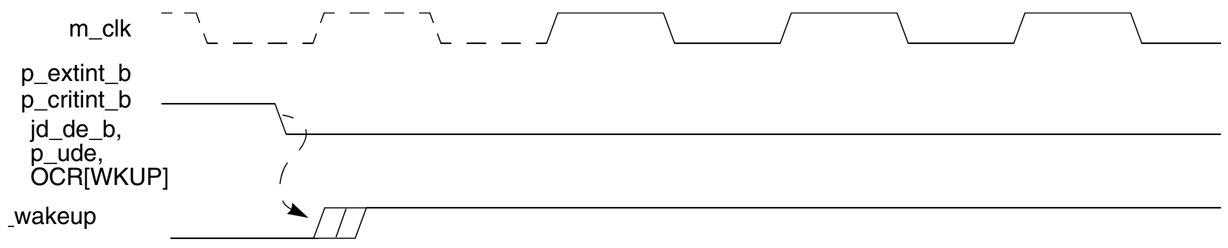


Figure 14-38. Wakeup control signal (p_wakeup)

14.3.6 Interrupt Interface

The following diagram shows the relationship of the interrupt input signals to the CPU clock. The **p_avec_b**, **p_extint_b**, **p_critint_b** and **p_voffset[0:15]** inputs as well as the **p_nmi_b** input must meet setup and hold timing relative to the rising edge of the **m_clk**. In addition, during each clock cycle in which either of the interrupt request inputs **p_extint_b** or **p_critint_b** are asserted, **p_avec_b** and **p_voffset[0:15]** are required to be in a valid state for the highest priority non-masked interrupt being requested.

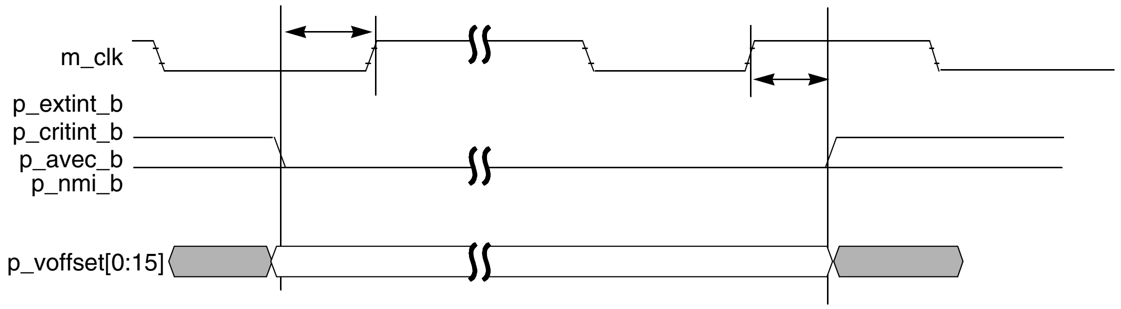


Figure 14-39. Interrupt interface input signals

Figure 14-40 shows the relationship of the interrupt pending signal to the interrupt request inputs. Note that **p_ipend** is asserted combinationally from the **p_extint_b**, **p_critint_b**, and **p_nmi_b** inputs, and the MCSR_{NMI} syndrome bit.

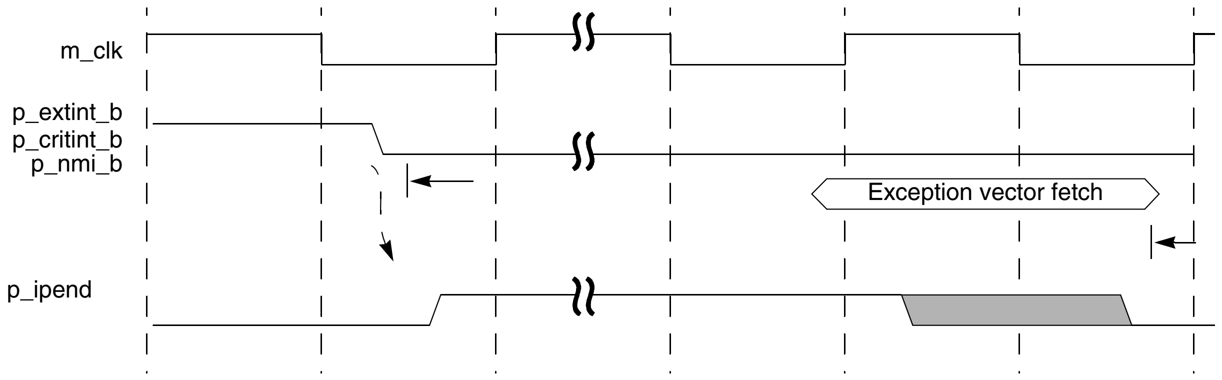


Figure 14-40. Interrupt pending operation

Figure 14-41 shows the relationship of the interrupt acknowledge signal to the interrupt request inputs and exception vector fetching.

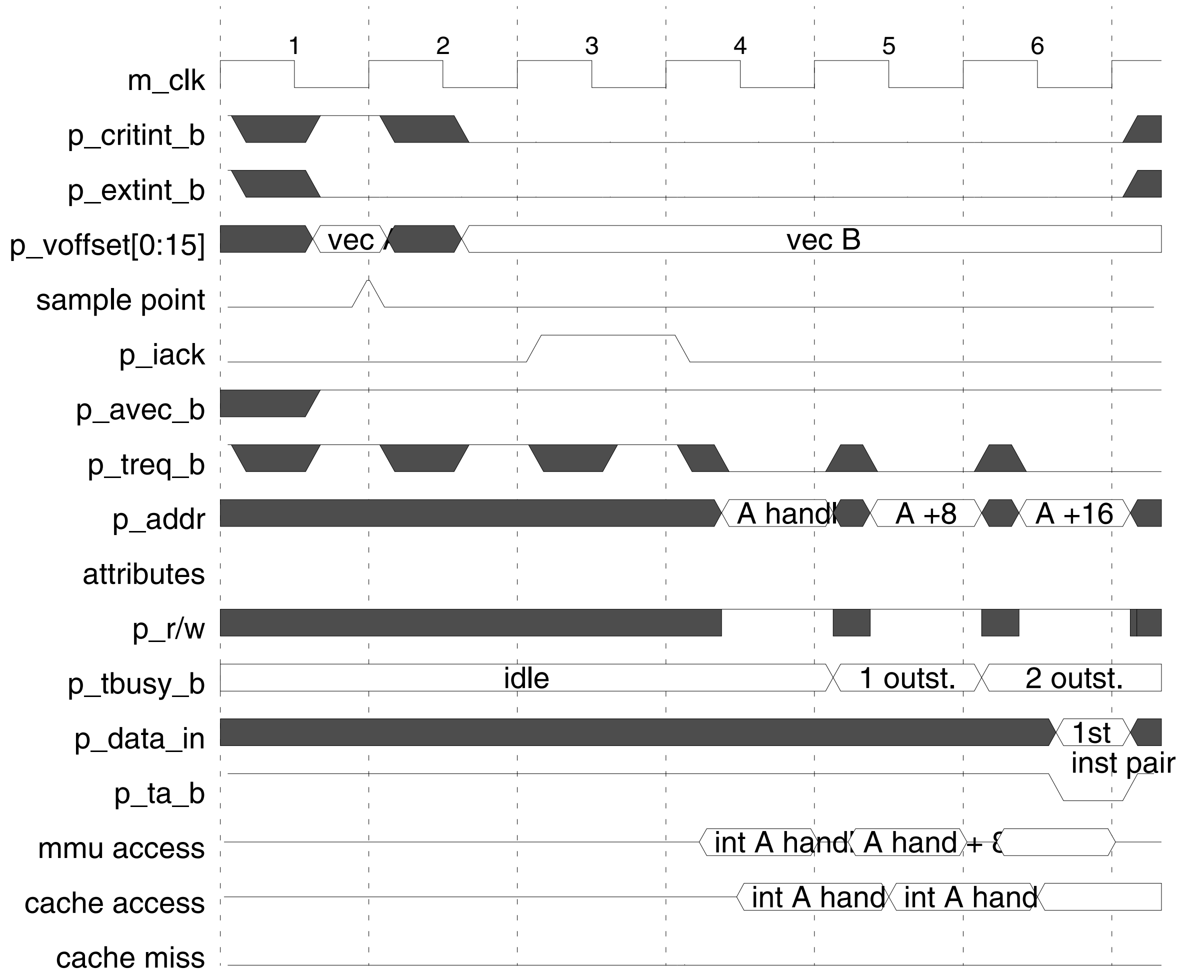


Figure 14-41. Interrupt acknowledge operation — 1

In this example, an external input interrupt is requested in cycle 1. The **p_voffset[0:15]** inputs are driven with the vector offset for ‘A’, and **p_avec_b** is negated, indicating vectoring is desired. For this example, the bus is idle at the time of assertion. The CPU may sample a requested interrupt as early as the cycle it is initially requested, and does so in this example. The interrupt request and the vector offset and autovector input are sampled at the end of cycle 1. In cycle 3, the interrupt is acknowledged by the assertion of the **p_iack** output, indicating that the values present on interrupt inputs at the beginning of cycle 2 have been internally latched and committed to for servicing. Note that the interrupt vector lines have changed to a value of ‘B’ during cycle 2, and the **p_critint_b** input has been asserted by the interrupt controller. The vector number / autovector signals must be consistent with the higher priority critical input request, thus must change at the same time the state of the interrupt request inputs change. Since 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 4, the CPU begins instruction fetching of the handler for vector ‘A’. The new request for a subsequent critical interrupt ‘B’ was not received in time to be acted upon first. It will be acknowledged after the fetch for the external input interrupt handler has been completed and has entered decode.

Note that the time between assertion of an interrupt request input and the acknowledgment of an interrupt may be multiple cycles, and the interrupt inputs may change during that interval. The CPU will assert the **p_iack** output to indicate the cycle at which an interrupt is committed to. In the following example, since the CPU was unable to acknowledge the external input interrupt during cycle 2 due to internal or external execution conditions, the critical input request was sampled. This case is shown in Figure 14-42.

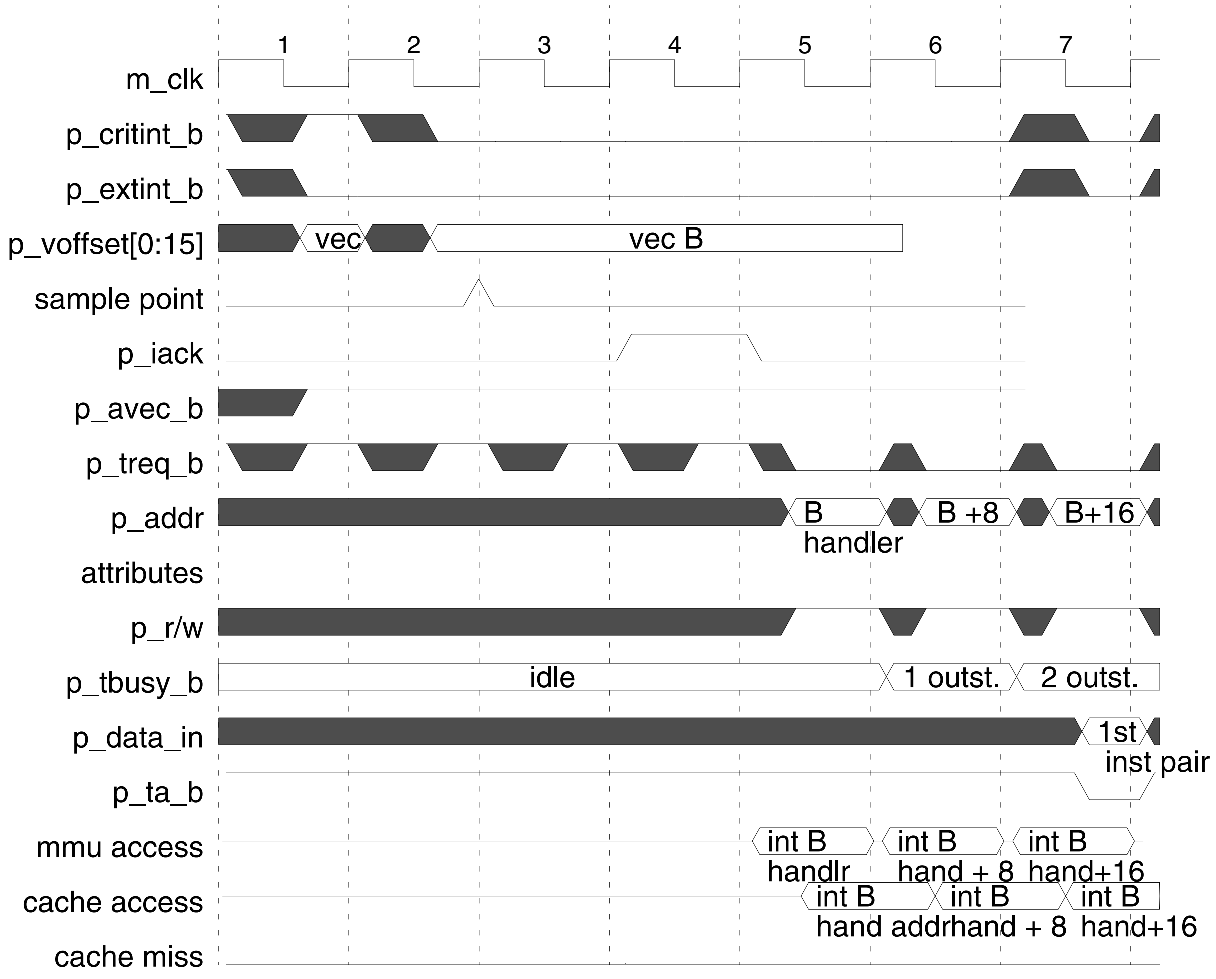


Figure 14-42. Interrupt acknowledge operation — 2

14.3.7 Time base interface

The following figure shows the required relationships of the Time Base inputs. The electrical values associated with these timings may be found in the *Zen Integration Guide*.

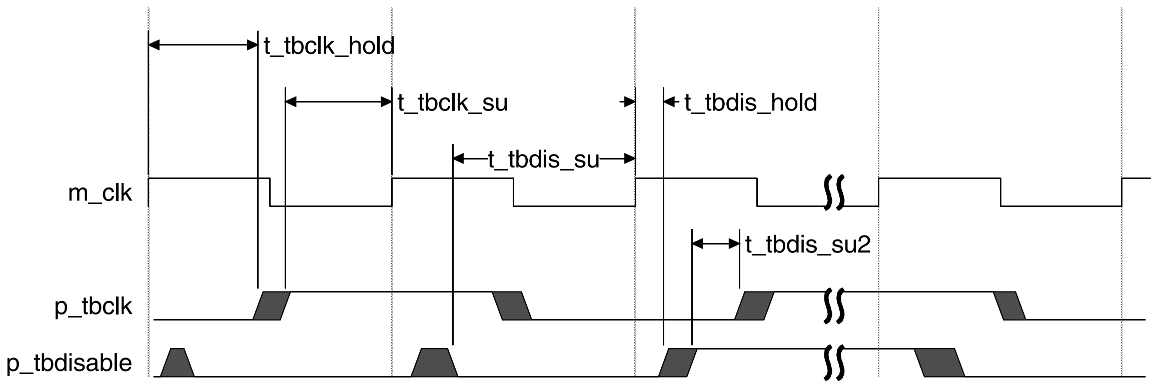


Figure 14-43. Time base input timing

14.3.8 JTAG test interface

The following figures show the relationships of the various JTAG related signals to the **j_tclk** input. The electrical values associated with these timings may be found in the *Zen Integration Guide*.

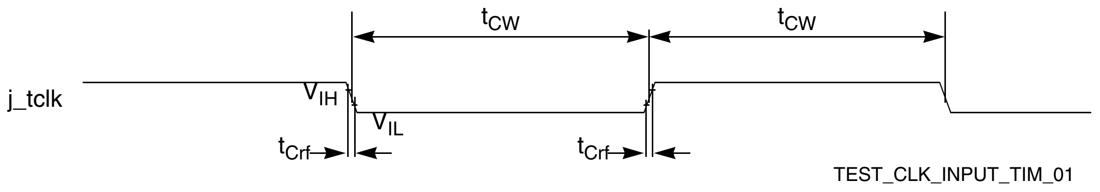


Figure 14-44. Test clock input timing

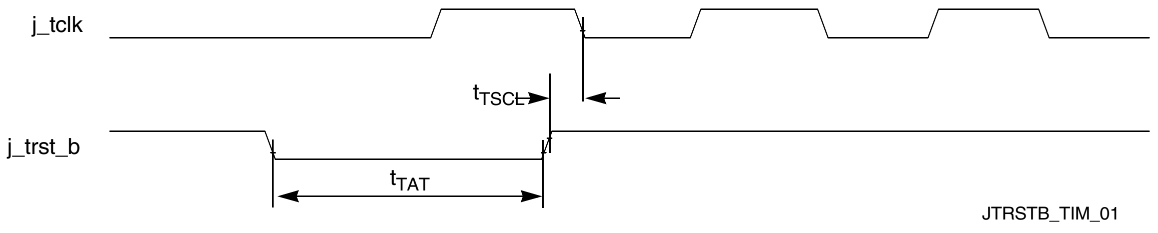


Figure 14-45. j_trst_b timing

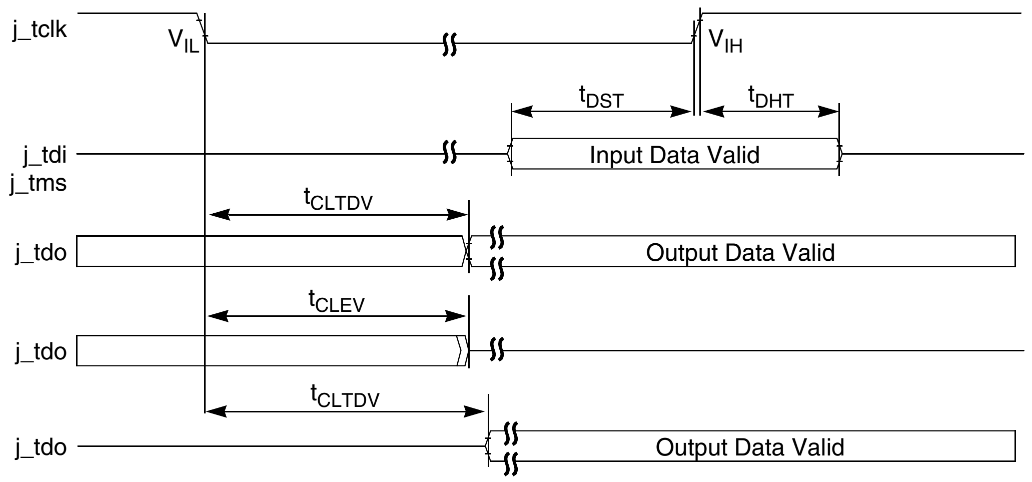


Figure 14-46. Test access port timing

Chapter 15 Internal Core Interfaces

This chapter describes several interfaces internal to the e200z759n3. Signal descriptions as well as the data transfer protocols are documented in the following sub-sections. The information is provided to assist in understanding operation of various internal interfaces. These interfaces are not visible outside of the core complex. Refer to Chapter 14, External Core Complex Interfaces, for external interfaces and protocols.

NOTE

These interface signals are not visible to the end user and do not require any user interaction. They are documented for informational purposes only.

The primary Zen internal interfaces comprises control and data signals supporting instruction and data transfers, support for accessing SPRs external to the CPU (but internal to the e200z759n3), and an interface to support the Embedded Floating-point (EFPU) APU.

The memory portions of the Zen core interface are utilized by the instruction and data caches and the Memory Management Unit (MMU). The data memory interface supports read and write transfers of 8, 16, 24, 32, and 64 bits, supports misaligned transfers, supports true big- and little-endian operating modes, and operates in a highly pipelined fashion. The instruction memory interface supports read and write transfers of 16, 32, and 64 bits, supports true big- and little-endian operating modes, and operates in a highly pipelined fashion.To achieve high frequency of operation, the processor pipeline has been designed to provide maximal access time for memory devices. In doing so, up to three accesses may be in some stage of progress at any one time on each interface, and a specific protocol is required for supporting maximal throughput. In particular, the Cache memory controller is responsible for ensuring that all accesses are issued and complete in-order, reporting completion and/or exceptions at the time the CPU expects, and handling the issues of aborting accesses that are in the memory pipeline to ensure a precise exception and in-order execution model. This control is assisted by the CPU, but some aspects are the sole responsibility of the Cache memory control logic.

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 doubleword, the core performs a single transfer, and the data cache interface is responsible for delivering (reads) or accepting (writes) the data corresponding to the size signals aligned according to the low order three address bits. If an access is misaligned and crosses a 64-bit boundary, the e200z759n3 load/store unit will perform a pair of transfers beginning at the effective address, requesting the original data size (either halfword or word) for the first transfer, and for the second transfer the address is incremented to the next 64-bit boundary, and the size signals are driven to indicate the number of remaining bytes to be transferred.

15.1 Signal index

This section contains an index of internal e200z759n3 signals.

The following prefixes are used for e200z759n3 signal mnemonics:

- **m** denotes master clock and reset signals
- **p** denotes processor or core-related signals
- **p_d_** denotes data interface related signals
- **p_i_** denotes instruction interface related signals

- **j** denotes JTAG mode signals
- **jd** denotes JTAG and Debug mode signals
- **ipt** denotes Scan and Test Mode signals

NOTE

The "_b" suffix denotes an active low signal. Signals without the active-low suffix are active high.

Figure 15-1 groups core bus and control signals by function.

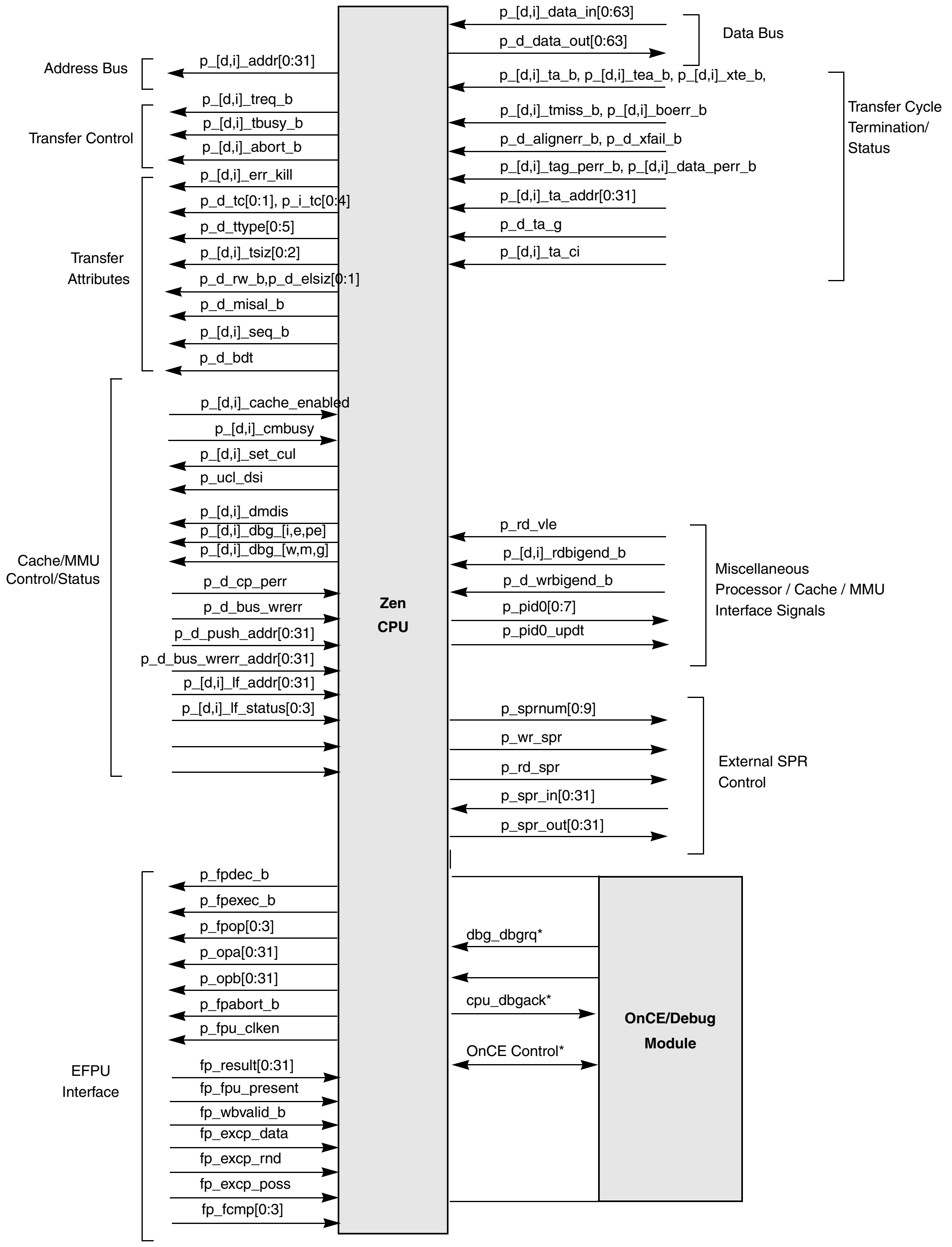


Figure 15-1. Zen internal signal groups

Table 15-1 shows e200z759n3 signal function and type, signal definition, and reset value. Signals are presented in functional groups.

Table 15-1. Internal interface signal definitions

Signal name	Type	Reset value	Definition
Memory interface signals			
p_[d,i]_addr[0:31]	O	—	Address bus
p_d_rw_b	O	1	Read/$\overline{\text{write}}$
p_i_tc[0:4], p_d_tc[0:1]	O	—	Transfer Code
p_d_ttype[0:5]	O	—	Transfer Type
p_[d,i]_tsiz[0:2]	O	—	Transfer Size
p_elsiz[0:1]	O	—	Element Size
p_d_seq_b	O	1	Indicates the current access is in sequential address order from the last access. For sequential data fetches.
p_i_seq_b	O	1	Indicates the current instruction access is in sequential address order from the last instruction access. For sequential instruction fetches.
p_d_misal_b	O	1	Indicates the current data access is the first portion of a misaligned access.
p_d_bdt	O	0	Indicates the current data access is part of a lmw or stmw transfer sequence.
p_[d,i]_treq_b	O	1	Transfer Request Indicates a request for a bus cycle.
p_[d,i]_tbusy[0:1]_b	O	1	Transfer Busy p_tbusy[0]_b Indicates a bus cycle is in progress. p_tbusy[1]_b Indicates that two accesses have been pipelined and a bus cycle is in progress.
p_[d,i]_abort_b	O	1	Aborts a requested access.
p_[d,i]_data_in[0:63]	I		Input data bus
p_d_data_out[0:63]	O	—	Output data bus
p_[d,i]_halt_zlb	I		Stall additional access requests
p_[d,i]_ta_b	I		Transfer Acknowledge
p_[d,i]_tea_b	I		Transfer Error
p_[d,i]_tmiss_b	I		Translation Miss
p_[d,i]_boerr_b	I		Byte Ordering Error
p_d_alignerr_b	I		Alignment Error
p_[d,i]_tag_perr_b	I		Cache Tag Parity Error
p_[d,i]_data_perr_b	I		Cache Data Parity Error

Table 15-1. Internal interface signal definitions (continued)

Signal name	Type	Reset value	Definition
p_[d,i]_xte_b	I		External Termination Error (machine check)
p_d_xfail_b	I		Store Exclusive Failure
p_[d,i]_ta_addr[0:31]	I		Physical address associated with ta_b/tea_b
p_d_ta_g	I		Guarded attribute of the access associated with ta_b/tea_b
p_d_ta_ci	I		Cache-inhibited attribute of the access associated with ta_b/tea_b
p_rd_vle	I		Indicates VLE or BookE mode for inst accesses.
p_[d,i]_rdbigend_b	I		Selects Little or Big Endian mode for read accesses.
p_d_wrbigend_b	I		Selects Little or Big Endian mode for write accesses.
SPR interface signals			
p_sprnum[0:9]	O	—	Global SPR address bus
p_spr_out[0:31]	O	—	Global SPR write bus
p_spr_in[0:31]	I		Global SPR read bus
p_rd_spr	O	0	SPR read control
p_wr_spr	O	0	SPR write control
Misc. CPU Signals			
p_pid0[0:7]	O	0	PID0[24:31] outputs
p_pid0_updt	O	0	PID0 update status
Cache/MMU status signals			
p_[i,d]_cache_enabled	I		Cache is enabled
p_[i,d]_cmbusy	I		Cache/MMU busy
p_[i,d]_set_cul	O		Set Cache CUL status
p_[i,d]_ucl_dsi	O		User mode Cache lock DSI control
p_[i,d]_dmdis	O		Debug Mode MMU disable
p_[i,d]_dbg_w	O		Debug Mode ‘W’ attribute
p_[i,d]_dbg_i	O		Debug Mode ‘I’ attribute
p_[i,d]_dbg_m	O		Debug Mode ‘M’ attribute
p_[i,d]_dbg_g	O		Debug Mode ‘G’ attribute
p_[i,d]_dbg_e	O		Debug Mode ‘E’ attribute
p_[i,d]_lf_status[0:3]	I		Cache Linefill Status
p_[i,d]_lf_addr[0:31]	I		Linefill Physical address

Table 15-1. Internal interface signal definitions (continued)

Signal name	Type	Reset value	Definition
p_d_cp_perr	I		Cache Push Parity Error
p_d_push_addr[0:31]	I		Address of the push line
p_d_bus_wrerr	I		Cache Buffered Write or Push Bus Error
p_d_bus_wrerr_addr[0:31]	I		Bus write error Physical address
EFPU interface signals			
p_fpdec_b	O	1	Indicates an FPU instruction is being decoded
p_fpexec_b	O	1	Indicates an FPU instruction is being executed
p_fpop[0:3]	O	0	FPU Opcode
p_fpabort_b	O	1	Indicates FPU instruction is to be aborted
p_opa[0:31]	O	—	Operand A to FPU
p_opb[0:31]	O	—	Operand B to FPU
p_fpu_clken	O	0	FPU clock enable
fp_result[0:31]	I		FPU result bus
fp_fpu_present	I		Indicate the FPU is present
fp_wbvalid_b	I		Indicates FPU resultant write-back valid
fp_excp_data	I		FPU result has a data exception
fp_excp_rnd	I		FPU result has a round exception
fp_excp_poss	I		FPU exception is possible, must stall subsequent data access
fp_fcmp[0:3]	I		FPU compare results
Test primary input/output signals			
Test Control Interface[1]			Test mode determination
Scan Test Interface[1]			Scan configuration and testing
Memory BIST Interface[1]			Memory BIST configuration and testing

[1] Please refer to the *e200z759n3 Test Guide* for information on the test signals.

15.2 Signal descriptions

15.2.1 Address and data buses

15.2.1.1 Data address bus (p_d_addr[0:31])

These outputs provide the address for a data transfer.

15.2.1.2 Instruction address bus (p_i_addr[0:31])

These outputs provide the address for an instruction transfer.

15.2.1.3 Data input data bus (p_d_data_in[0:63])

These inputs provide data to the e200z759n3 core on data read transfers. The data input data bus can transfer 8, 16, 24, 32, or 64 bits of data per bus transfer.

15.2.1.4 Instruction input data bus (p_i_data_in[0:63])

These inputs provide data to the e200z759n3 core for instruction transfers. The instruction input data bus can transfer 32 or 64 bits of data per bus transfer.

15.2.1.5 Data output data bus (p_d_data_out[0:63])

These outputs transfer data from the e200z759n3 core on data write transfers. The output data bus can transfer 8, 16, 24, 32, or 64 bits of data per bus transfer.

15.2.2 Transfer attribute signals

The following paragraphs describe the transfer attribute signals, which provide additional information about the bus transfer cycle. Transfer attributes are driven with address at the beginning of a bus transfer.

15.2.2.1 Read/write (p_d_rw_b)

This output signal defines the data transfer direction for the current data bus cycle. A high (logic one) level indicates a read cycle, and a low (logic zero) level indicates a write cycle.

15.2.2.2 Data transfer code (p_d_tc[0:1])

The e200z759n3 core drives the **p_d_tc[0:1]** signals to indicate the type of access for the current bus cycle. **pd_tc[0]** indicates user/supervisor, and **p_d_tc[1]** indicates address space 0/1. Table 15-2 shows the definitions of the **p_d_tc[0:1]** encodings.

Table 15-2. p_d_tc[0:1] transfer code encoding

p_d_tc[0:1]	Transfer type
00	User Data Space 0 Access
01	User Data Space 1 Access
10	Supervisor Data Space 0 Access
11	Supervisor Data Space 1 Access

15.2.2.3 Instruction transfer code (p_i_tc[0:4])

The e200z759n3 core drives the **p_i_tc[0:4]** signals to indicate the type of access for the current bus cycle. **p_i_tc[0]** indicates user/supervisor, **p_i_tc[1]** indicates address space 0/1, **p_i_tc[2]** indicates exception vectoring, **p_i_tc[3]** indicates change of flow, and **p_i_tc[4]** indicates a speculative branch target prefetch. Table 15-3 shows the definitions of the **p_i_tc[0:4]** encodings.

Table 15-3. p_i_tc[0:4] transfer code encoding

p_i_tc[0:4]	Transfer type
00000	Reserved
00001	Reserved
00010	Reserved
00011	Reserved
00100	User Instruction Space 0 Access[1]
00101	Reserved
00110	User Change of Flow Instruction Space 0 Access[2]
00111	User Change of Flow Instruction Space 0 Speculative Branch Target Access[3]
01000	Reserved
01001	Reserved
01010	Reserved
01011	Reserved
01100	User Instruction Space 1 Access[1]
01101	Reserved
01110	User Change of Flow Instruction Space 1 Access[2]
01111	User Change of Flow Instruction Space 1 Speculative Branch Target Access[3]
10000	Reserved
10001	Reserved
10010	Supervisor Exception Vector Instruction Access[4]
10011	Reserved
10100	Supervisor Instruction Space 0 Access
10101	Reserved
10110	Supervisor Change of Flow Instruction Space 0 Access[2]
10111	Supervisor Change of Flow Instruction Space 0 Speculative Branch Target Access[3]
11000	Reserved
11001	Reserved
11010	Reserved
11011	Reserved
11100	Supervisor Instruction Space 1 Access[1]
11101	Reserved
11110	Supervisor Change of Flow Instruction Space 1 Access[2]

Table 15-3. p_i_tc[0:4] transfer code encoding (continued)

p_i_tc[0:4]	Transfer type
11111	Supervisor Change of Flow Instruction Space 1 Speculative Branch Target Access[3]

[1] Except Change of Flow related instruction accesses
[2] Change of Flow related instruction access for taken branches
[3] Speculative Branch target instruction access
[4] Initial Instruction fetch for Interrupt Handler

15.2.2.4 Data transfer size (p_d_tsiz[0:2])

The **p_d_tsiz[0:2]** signals indicate the data size for a bus transfer. Table 15-4 shows the definitions of the **p_d_tsiz[0:2]** encodings.

Table 15-4. Data transfer size encoding

p_d_tsiz[0:2]	Transfer size
000	Doubleword [Pair[1]] (8 Bytes)
001	Byte
010	Halfword (2 Bytes)
011	Three bytes
100	Word (4 bytes)
101	Five bytes
110	Six bytes
111	Seven bytes

[1] Doubleword encoding is used for transfers of a pair of 32 bit words (lmw, stmw).

15.2.2.5 Element size (p_elsiz[0:1])

The **p_elsiz[0:1]** signals indicate the size of elements being transferred on certain writes. The element size may be smaller than the transfer size and further defines how byte ordering should be performed on these write cycles. Element size is required to distinguish the transfer of a pair of words, pair of halfwords, quad of bytes, octet of bytes, or a quad of halfwords from a normal word or doubleword transfer. **p_elsiz[0:1]** should only be used on writes to little-endian pages of memory, and should be ignored on all other cycles. For misaligned transfers that cross a 64-bit boundary, **p_elsiz[0:1]** is driven to the same value for both portions of the transfer. Table 15-5 shows the definitions of the **p_elsiz[0:1]** encodings.

Table 15-5. Element size encoding

p_elsiz[0:1]	Element size
00	Word (4bytes), Word Pairs for **stmw**
01	Byte

Table 15-5. Element size encoding (continued)

p_elsiz[0:1]	Element size
10	Halfword (2 Bytes)
11	Doubleword (8 bytes)

15.2.2.6 Instruction Transfer Size (p_i_tsiz[0:2])

The **p_i_tsiz[0:2]** signals indicate the data size for a bus transfer. Table 15-6 shows the definitions of the **p_i_tsiz[0:2]** encodings.

Table 15-6. Instruction transfer size encoding

p_i_tsiz[0:2]	Transfer size
000	Doubleword (8 bytes)
001	reserved
010	reserved
011	reserved
100	Word (4 bytes)
101	reserved
110	reserved
111	reserved

15.2.2.7 Data Transfer Type (p_d_ttype[0:5])

These signals indicate the type of transfer for the data bus cycle. Timing of these signals is early relative to address to allow sufficient control logic timing. Table 15-7 shows the definitions of the **p_d_ttype[0:5]** encodings.

Table 15-7. Transfer type encoding

p_d_ttype[0:5][1]	Transfer Type	Instruction
00000e	Normal	Normal loads / stores
000010	Atomic	**lbarx, lharx, lwarx, stbcx., sthcx.,** and **stwcx.**
00010e	Flush Data Block	**dcbst**
00011e	Flush and Invalidate Data Block	**dcbf**
00100e	Allocate and Zero Data Block	**dcbz**
001010	Invalidate Data Block	**dcbi**
00110e	Invalidate Instruction Block	**icbi**
001110	multiple word load/store	**lmw, stmw**
010000	TLB Invalidate	**tlbivax**

Table 15-7. Transfer type encoding (continued)

p_d_ttype[0:5][1]	Transfer Type	Instruction
010010	TLB Search	**tlbsx**
010100	TLB Read entry	**tlbre**
010110	TLB Write entry	**tlbwe**
011000	Touch for Instruction	**icbt**
011010	Lock Clear for Instruction	**icblc**
011100	Touch for Instruction and Lock Set	**icbtls**
011110	Lock Clear for Data	**dcblc**
10000e	Touch for Data	**dcbt**
10001e	Touch for Data Store	**dcbtst**
100100	Touch for Data and Lock Set	**dcbtls**
100110	Touch for Data Store and Lock Set	**dcbtstls**

[1] p_ttype[5] 'e' is set to set to 0.

15.2.2.8 Data sequential access (p_d_seq_b)

This active-low output signal indicates that the current data access is in sequential address order from the previous data access. The timing of this signal is approximately the same as address timing.

15.2.2.9 Instruction sequential access (p_i_seq_b)

This active-low output signal indicates that the current instruction access is in sequential address order from the previous instruction access. This signal is driven for sequential instruction fetches only. The timing of this signal is approximately the same as address timing.

15.2.2.10 Misaligned access (p_d_misal_b)

This active-low output signal indicates that the current data access is the first portion of a misaligned load or store access that crosses a 64-bit boundary. The timing of this signal is approximately the same as address timing.

15.2.2.11 Block data transfer (p_d_bdt)

This active-high output signal indicates that the current data access is part of a block data transfer for a lmw or stmw instruction. The timing of this signal is approximately the same as address timing.

15.2.2.12 Error kill control (p_d_err_kill, p_i_err_kill)

This active-high output signal indicates that the current access, if terminated with error (**p_tea_b** assertion), will cause a following pending access to be aborted by assertion of **p_abort_b**. If an access is pending or is requested in the cycle that **p_tea_b** is asserted for a current request for which **p_err_kill** was initially asserted, the new access will be aborted by assertion of **p_abort_b** in the cycle following error assertion for the current access. If no pending or requested access is present in the cycle **p_tea_b** terminates an access, this signal has no effect. The timing of this signal is approximately the same as address timing.

15.2.3 Transfer control signals

The following paragraphs describe the transfer control signals.

15.2.3.1 Halt ZLB (p_d_halt_zlb, p_i_halt_zlb)

These signals are drive to the CPU by the respective cache to indicate that further access requests should be temporarily halted due to allow the cache to process other operations. This signal may be held asserted for multiple cycles if a busy condition remains pending.

15.2.3.2 Transfer request (p_d_treq_b, p_i_treq_b)

The e200z759n3 core drives these active-low output signals to indicate that a new access has been requested. This signal is driven for a single cycle along with address and transfer attribute signals to request a new cycle, and may be held asserted for multiple cycles if a request remains pending, or if multiple requests occur.

15.2.3.3 Transfer busy (p_d_tbusy[0:1]_b, p_i_tbusy[0:1]_b)

The processor drives these active-low signals to indicate that one or more accesses are in progress. These signals are driven for the duration of a cycle, and may be held asserted for multiple transfers. Table 15-8 shows **p_[d,i]_tbusy[0:1]_b** encoding.

Table 15-8. p_tbusy[0:1]_b encoding

p_[d,i]_tbusy[0]_b	p_[d,i]_tbusy[1]_b	Access status
0	0	Two accesses in progress. Accesses have been pipelined, Awaiting termination for both accesses. A third access may be in the request phase.
0	1	One access in progress. Awaiting termination for one access. A second access may be in request phase
1	0	Illegal state
1	1	Idle — no access in progress

p_[d,i]_tbusy[0]_b is used to indicate that the bus is busy with one or more accesses. **p_[d,i]_tbusy[1]_b** indicates that two outstanding accesses exist. A third may be in the request phase, but will not become outstanding until the first outstanding access is terminated, or the second access is aborted.

15.2.3.4 Transfer abort (p_d_abort_b, p_i_abort_b)

This active-low signal to indicate that a requested access must be aborted. This signal may be driven on the clock following a valid requested cycle if an outstanding access on which **p_err_kill** was indicated is terminated with **p_tea_b**. During the clock cycle that **p_[d,i]_abort_b** is asserted, another access may be requested (**p_treq_b** asserted), but will not be taken. Aborted accesses are terminated with assertion of **p_tea_b**.

15.2.3.5 Transfer acknowledge (p_d_ta_b, p_i_ta_b)

This active-low input signal indicates completion of a requested transfer. Assertion of **p_[d,i]_ta_b** terminates the transfer. For the e200z759n3 core to accept the transfer as successful, **p_[d,i]_tea_b** must remain high while **p_[d,i]_ta_b** is asserted.

15.2.3.6 Transfer error acknowledge (p_d_tea_b, p_i_tea_b)

This active-low input signal indicates that a transfer error condition has occurred and causes the e200z759n3 core to immediately terminate the transfer. An external device asserts **p_[d,i]_tea_b** to terminate the transfer with error. The **p_[d,i]_tea_b** signal has higher priority than **p_[d,i]_ta_b**.

15.2.3.7 Translation miss (p_d_tmiss_b, p_i_tmiss_b)

This active-low input signal indicates a translation miss. The memory management unit asserts **p_[d,i]_tmiss_b** to indicate a TLB miss condition for the current transfer. The assertion of **p_[d,i]_tmiss_b** must be concurrent with the assertion of **p_[d,i]_tea_b** to be recognized, as it is sampled with assertion of **p_[d,i]_tea_b**.

15.2.3.8 Byte ordering error (p_d_boerr_b, p_i_boerr_b)

This active-low input signal indicates a byte ordering error due to mismatched endianness for a misaligned access that crosses a page boundary. The memory management unit asserts **p_[d,i]_boerr_b** to indicate this condition for the current transfer. The assertion of **p_[d,i]_boerr_b** must be concurrent with the assertion of **p_[d,i]_tea_b** to be recognized, as it is sampled with assertion of **p_[d,i]_tea_b**. This signal is ignored if the **p_[d,i]_tmiss_b** input is asserted.

15.2.3.9 Alignment error (p_d_alignerr_b)

This active-low input signal indicates an Alignment error due to execution of a **dcbz** instruction to a location marked as Cache-inhibited, Writethrough Required, or if the cache is operating in writethrough mode or if a line cannot be allocated on a miss due to locking or way-disabling constraints. Alignment errors due to a disabled cache are handled by the CPU directly. The data cache asserts **p_d_alignerr_b** to indicate this condition for the current transfer. The assertion of **p_d_alignerr_b** must be concurrent with the assertion of **p_d_tea_b** to be recognized, as it is sampled with assertion of **p_d_tea_b**.

15.2.3.10 Cache tag parity error (p_d_tag_perr_b, p_i_tag_perr_b)

The active-low **p_[d,i]_tag_perr_b** input signal is used to indicate that a cache parity error has occurred while accessing the cache for a load, store or instruction fetch. This signal is asserted in a precise fashion at termination of the cache access. The assertion of **p_[d,i]_tag_perr_b** must be concurrent with the assertion of **p_[d,i]_tea_b** to be recognized, as it is sampled with assertion of **p_[d,i]_tea_b**. This signal is used to generate a machine check condition and causes the associated syndrome bit to be set in the Machine Check Syndrome register (Section 2.4.7, Machine Check Syndrome Register (MCSR)).

15.2.3.11 Cache data parity error (p_d_data_perr_b, p_i_data_perr_b)

The active-low **p_[d,i]_data_perr_b** input signal is used to indicate that a cache parity error has occurred while accessing the cache for a load, store or instruction fetch. This signal is asserted in a precise fashion at termination of the cache access. The assertion of **p_[d,i]_data_perr_b** must be concurrent with the assertion of **p_[d,i]_tea_b** to be recognized, as it is sampled with assertion of **p_[d,i]_tea_b**. This signal is used to generate a machine check condition and causes the associated syndrome bit to be set in the Machine Check Syndrome register (Section 2.4.7, Machine Check Syndrome Register (MCSR)).

15.2.3.12 External termination error (p_d_xte_b, p_i_xte_b)

This active-low input signal indicates a Precise External Termination error occurred. Assertion of **p_[d,i]_xte_b** indicates that a precise external error condition for the current data transfer has occurred. The assertion of **p_[d,i]_xte_b** must be concurrent with the assertion of **p_d_tea_b** to be recognized, as it is sampled with assertion of **p_d_tea_b**. This signal is asserted to indicate a precise external error termination on the System bus has occurred, as opposed to a permission violation detected by the MMU.

15.2.3.13 Guarded termination status (p_d_ta_g)

This active-high input signal indicates that the access being terminated was a guarded access. The **p_d_ta_g** signal is sampled with assertion of **p_d_ta_b** or **p_d_tea_b**.

15.2.3.14 Cache-inhibited termination status (p_d_ta_ci)

This active-high input signal indicates that the access being terminated was a cache-inhibited access. The **p_d_ta_ci** signal is sampled with assertion of **p_d_ta_b** or **p_d_tea_b**.

15.2.3.15 Access physical address (p_[d,i]_ta_addr[0:31])

This active-high input bus provides the physical address of the access being terminated with **p_[d,i]_ta_b** or **p_[d,i]_tea_b**, and is used for status updates during error conditions. The **p_[d,i]_ta_addr[0:31]** signals are sampled with assertion of the corresponding **p_[d,i]_ta_b** or **p_[d,i]_tea_b**.

15.2.3.16 Termination error signaling and qualification

The active-low input signals **p_tmiss_b**, **p_alignerr_b**, **p_boerr_b**, **p_data_perr_b**, **p_tag_perr_b**, and **p_xte_b** are provided to indicate types of errors that have occurred on attempted accesses, and affect exception vectoring, priority, and status updates. These signals are ignored unless **p_tea_b** is also asserted. These signals are mutually exclusive and should not be asserted in conjunction with one another. Table 15-9 summarizes the meaning of these error termination qualifiers.

Table 15-9. Termination error qualifiers[1]

Priority (0= highest)	p_tea_b	p_alignerr_b	p_boerr_b	p_tmiss_b	p_tag_perr_b	p_data_perr_b	p_xte_b	Access status
0	0	1	1	0	1	1	1	TLB Error has occurred. Signaled by the MMU to indicated a TLB miss has occurred.
1	0	1	1	1	1	1	1	Storage Error has occurred. Signaled by the MMU to indicate improper access permissions for an instruction or data access.
2	0	0	1	1	1	1	1	Alignment Error has occurred. Signaled by cache to indicate improper page attributes for a **dcbz**.
	0	1	0	1	1	1	1	Byte-ordering Error has occurred. Signaled by the MMU to indicated mismatched endianness on access crossing a page boundary.
3	0	1	1	1	0	1	1	Cache Tag Parity Error has occurred. Signaled by the Cache to indicated a tag parity error occurred on a cache access for a load, store, or instruction fetch.
4	0	1	1	1	1	0	1	Cache Data Parity Error has occurred. Signaled by the Cache to indicated a data parity error occurred on a cache access for a load or instruction fetch.
5	0	1	1	1	1	1	0	Precise External Termination Error has occurred. Signaled by the BIU to indicated an external System bus ERROR response received on a load, unbuffered store, or instruction fetch.
6	1	x	x	x	x	x	x	No termination error

[1] All other combinations not listed in table are invalid.

15.2.3.17 Store exclusive failure (p_d_xfail_b)

This active-low input signal indicates a failure of the store portion of the **stwcx.** instruction. An external agent causes **p_d_xfail_b** to be asserted to indicate this condition for the current transfer. The assertion of **p_d_xfail_b** must be concurrent with the assertion of **p_d_ta_b** to be recognized, as it is sampled with assertion of **p_d_ta_b**. This signal is ignored if the **p_d_tea_b** input is asserted, since the store has terminated with error. Assertion of **p_d_xfail_b** with **p_d_ta_b** does not cause an exception to occur. It only indicates to the CPU that the store was not performed due to a loss of reservation (determined by an external agent). The CPU will update the condition code accordingly, and will clear an outstanding reservation. **p_d_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 will be signaled back to the CPU using this signal. See Section 3.5, Memory synchronization and reservation instructions, for additional information regarding reservations. The **p_d_xfail_b** input is ignored for all transfers other than a **stwcx.** store.

15.2.3.18 Read endian mode select (p_d_rdbigend_b, p_i_rdbigend_b)

This control input signal selects the Big Endian or Little Endian byte ordering mode for reads This input signal controls the byte-ordering operation of memory read transfers. When driven low, big-endian byte ordering is selected for reads, otherwise little-endian ordering is selected. This signal is required to be valid very early in the cycle that read data is returned.

15.2.3.19 Write endian mode select (p_d_wrbigend_b)

This control input signal selects the Big Endian or Little Endian byte ordering mode for writes. This signal is a static input that controls byte-ordering operation of memory write transfers. When driven low, big-endian byte ordering is selected for writes, otherwise little-endian ordering is selected. This signal should be tied low.

15.2.3.20 VLE mode select (p_rd_vle)

This control input signal selects BookE or VLE mode for instruction reads. When driven low, BookE mode is signaled, otherwise VLE is selected. This signal is required to be valid very early in the cycle that instruction read data is returned.

15.2.4 Byte lane specification

Read transactions transfer from 1 to 8 bytes of data on the **p_[d,i]_data_in** bus. The byte lanes involved in the transfer are determined by the starting byte number specified by the lower address bits in conjunction with the transfer size. Addressing of the byte lanes is always big-endian (left to right) regardless of the endian mode of the e200z759n3 core. The byte of memory corresponding to address 0 is connected to B0 and the byte of memory corresponding to address 7 is connected to B7. The CPU internally permutes read data as required for the endian mode of the current access. The endian mode is signaled with the **p_[d,i]_rdbigend_b** input signal that is sampled early in the cycle that data is returned.

Table 15-10 lists all of the read data transfer permutations. Note that misaligned data requests that cross a 64-bit boundary are broken up into two separate bus transactions, and the size encoding for the first transfer is not modified.

Table 15-10. Read data transfer permutations

Program size and byte offset	A(29:31)	TSIZ(0:2)	(0... p_data_in data bus byte lanes ...63)							
			B0	**B1**	**B2**	**B3**	**B4**	**B5**	**B6**	**B7**
Byte @000	0 0 0	0 0 1	A	—	—	—	—	—	—	—
Byte @001	0 0 1	0 0 1	—	A	—	—	—	—	—	—
Byte @010	0 1 0	0 0 1	—	—	A	—	—	—	—	—
Byte @011	0 1 1	0 0 1	—	—	—	A	—	—	—	—
Byte @100	1 0 0	0 0 1	—	—	—	—	A	—	—	—
Byte @101	1 0 1	0 0 1	—	—	—	—	—	A	—	—
Byte @110	1 1 0	0 0 1	—	—	—	—	—	—	A	—
Byte @111	1 1 1	0 0 1	—	—	—	—	—	—	—	A
Half @000	0 0 0	0 1 0	A	A	—	—	—	—	—	—
Half @001	0 0 1	0 1 0	—	A	A	—	—	—	—	—
Half @010	0 1 0	0 1 0	—	—	A	A	—	—	—	—
Half @011	0 1 1	0 1 0	—	—	—	A	A	—	—	—
Half @100	1 0 0	0 1 0	—	—	—	—	A	A	—	—
Half @101	1 0 1	0 1 0	—	—	—	—	—	A	A	—
Half @110	1 1 0	0 1 0	—	—	—	—	—	—	A	A
Half @111 (2 bus transfers)	1 1 1 0 0 0	0 1 0* 0 0 1	— A	— —	— —	— —	— —	— —	— —	A —
Word @000	0 0 0	1 0 0	A	A	A	A	—	—	—	—
Word @001	0 0 1	1 0 0		A	A	A	A	—	—	—
Word @010	0 1 0	1 0 0	—	—	A	A	A	A	—	—
Word @011	0 1 1	1 0 0	—	—	—	A	A	A	A	—
Word @100	1 0 0	1 0 0	—	—	—	—	A	A	A	A
Word @101 (2 bus transfers)	1 0 1 0 0 0	1 0 0* 0 0 1	— A	— —	— —	— —	— —	A —	A —	A —
Word @110 (2 bus transfers)	1 1 0 0 0 0	1 0 0* 0 1 0	— A	— A	— —	— —	— —	— —	A —	A —
Word @111 (2 bus transfers)	1 1 1 0 0 0	1 0 0* 0 1 1	— A	— A	— A	— —	— —	— —	— —	A —
Doubleword @000	0 0 0	0 0 0	A	A	A	A	A	A	A	A
Doubleword @001 (2 bus transfers)	0 0 1 0 0 0	0 0 0* 0 0 1	— A	A —	A —	A —	A —	A —	A —	A —
Doubleword @010 (2 bus transfers)	0 1 0 0 0 0	0 0 0* 0 1 0	— A	— A	A —	A —	A —	A —	A —	A —

Table 15-10. Read data transfer permutations (continued)

Program size and byte offset	A(29:31)	TSIZ(0:2)	(0... p_data_in data bus byte lanes ...63)							
			B0	B1	B2	B3	B4	B5	B6	B7
Doubleword @011 (2 bus transfers)	0 1 1 0 0 0	0 0 0* 0 1 1	— A	— A	— A	A —	A —	A —	A —	A —
Doubleword @100 (2 bus transfers)	1 0 0 0 0 0	1 0 0* 1 0 0	— A	— A	— A	— A	A —	A —	A —	A —
Doubleword @101 (2 bus transfers)	1 0 1 0 0 0	1 0 0* 1 0 1	— A	— A	— A	— A	— A	A —	A —	A —
Doubleword @110 (2 bus transfers)	11 0 0 0 0	1 0 0* 1 1 0	— A	— A	— A	— A	— A	— A	A —	A —
Doubleword @111 (2 bus transfers)	1 1 1 0 0 0	1 0 0* 1 1 1	— A	— A	— A	— A	— A	— A	— A	A —

Table Notes:
"A" indicates byte lanes involved in the transfer; Other lanes will contain driven but unused data.
* These misaligned cases drive request size according to the size specified by the load instruction.

For writes, the CPU drives data right justified onto the **p_d_data_out** bus, in the endian format defined by the **p_d_wrbigend_b** input signal, regardless of the byte offset. For misaligned accesses that are broken into two separate accesses, the **p_d_data_out** bus is driven with the same data value for both accesses. The memory controller must determine which bytes are written to which memory locations based on the endianness of the page(s).

Table 15-11 lists all of the write data transfer permutations. Note that misaligned data requests that cross a 64-bit boundary are broken up into two separate bus transactions, and the size encoding for the first transfer is not modified. For these accesses, **p_d_data_out** remains driven with the same value for both portions of the misaligned access.

Table 15-11. Write data transfer permutations

Program size and byte offset	A(29:31)	TSIZ(0:2)	(0... p_data_out data bus byte lanes ...63)							
			B0	B1	B2	B3	B4	B5	B6	B7
Byte @000	0 0 0	0 0 1	—	—	—	—	—	—	—	A
Byte @001	0 0 1	0 0 1	—	—	—	—	—	—	—	A
Byte @010	0 1 0	0 0 1	—	—	—	—	—	—	—	A
Byte @011	0 1 1	0 0 1	—	—	—	—	—	—	—	A
Byte @100	1 0 0	0 0 1	—	—	—	—	—	—	—	A
Byte @101	1 0 1	0 0 1	—	—	—	—	—	—	—	A
Byte @110	1 1 0	0 0 1	—	—	—	—	—	—	—	A
Byte @111	1 1 1	0 0 1	—	—	—	—	—	—	—	A
Half @000	0 0 0	0 1 0	—	—	—	—	—	—	A	A

Table 15-11. Write data transfer permutations (continued)

Program size and byte offset	A(29:31)	TSIZ(0:2)	(0... p_data_out data bus byte lanes ...63)							
			B0	B1	B2	B3	B4	B5	B6	B7
Half @001	0 0 1	0 1 0	—	—	—	—	—	—	A	A
Half @010	0 1 0	0 1 0	—	—	—	—	—	—	A	A
Half @011	0 1 1	0 1 0	—	—	—	—	—	—	A	A
Half @100	1 0 0	0 1 0	—	—	—	—	—	—	A	A
Half @101	1 0 1	0 1 0	—	—	—	—	—	—	A	A
Half @110	1 1 0	0 1 0	—	—	—	—	—	—	A	A
Half @111 (2 bus transfers)	1 1 1 0 0 0	0 1 0* 0 0 1	—	—	—	—	—	—	A	A
Word @000	0 0 0	1 0 0	—	—	—	—	A	A	A	A
Word @001	0 0 1	1 0 0	—	—	—	—	A	A	A	A
Word @010	0 1 0	1 0 0	—	—	—	—	A	A	A	A
Word @011	0 1 1	1 0 0	—	—	—	—	A	A	A	A
Word @100	1 0 0	1 0 0	—	—	—	—	A	A	A	A
Word @101 (2 bus transfers)	1 0 1 0 0 0	1 0 0* 0 0 1	—	—	—	—	A	A	A	A
Word @110 (2 bus transfers)	1 1 0 0 0 0	1 0 0* 0 1 0	—	—	—	—	A	A	A	A
Word @111 (2 bus transfers)	1 1 1 0 0 0	1 0 0* 0 1 1	—	—	—	—	A	A	A	A
Doubleword @000	0 0 0	0 0 0	A	A	A	A	A	A	A	A
Doubleword@001 (2 bus transfers)	0 0 1 0 0 0	0 0 0* 0 0 1	A	A	A	A	A	A	A	A
Doubleword@010 (2 bus transfers)	0 1 0 0 0 0	0 0 0* 0 1 0	A	A	A	A	A	A	A	A
Doubleword@011 (2 bus transfers)	0 1 1 0 0 0	0 0 0* 0 1 1	A	A	A	A	A	A	A	A
Doubleword@100 (2 bus transfers)	1 0 0 0 0 0	0 0 0* 1 0 0	A	A	A	A	A	A	A	A
Doubleword@101 (2 bus transfers)	1 0 1 0 0 0	0 0 0* 1 0 1	A	A	A	A	A	A	A	A
Doubleword@110 (2 bus transfers)	1 1 0 0 0 0	0 0 0* 1 1 0	A	A	A	A	A	A	A	A

Table 15-11. Write data transfer permutations (continued)

Program size and byte offset	A(29:31)	TSIZ(0:2)	(0... p_data_out data bus byte lanes ...63)							
			B0	B1	B2	B3	B4	B5	B6	B7
Doubleword@111 (2 bus transfers)	1 1 1 0 0 0	0 0 0* 1 1 1	A	A	A	A	A	A	A	A

Table Notes:
"A" indicates byte lanes involved in the transfer; Other lanes will contain driven but unused data.
* These misaligned cases drive request size according to the size specified by the store instruction. Write data is driven identically for both portions of the misaligned write. Byte ordering of driven write data is determined by the **p_d_wrbigend_b** control signal.

Table 15-12 shows the final layout in memory for data transferred from a register containing the bytes 'A B C D E F G H' to memory. Misaligned accesses that cross a doubleword boundary are broken into a pair of accesses by the CPU. Data is assumed to be 'A B C D E F G H' contained in a register, or 'A B C D E F G H', 'I J K L M N O P' contained in a pair of registers (**lmw**, **stmw** cases). Also shown are the cases where element size **p_elsiz[0:1]** is used for the transfers to memory in which element size differs from the requested transfer size.

Table 15-12. Big- and little-endian memory storage

Program size and byte offset	A(28:31)	TSIZ (0:2)	ELSIZ (0:1)	(0... data bus byte lanes ...63) even double word — 0								(0... data bus byte lanes ...63) 0dd double word — 1							
				B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7	B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7
Byte @0000	0 0 0 0	0 0 1	—	H	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Byte @0001	0 0 0 1	0 0 1	—	—	H	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Byte @0010	0 0 1 0	0 0 1	—	—	—	H	—	—	—	—	—	—	—	—	—	—	—	—	—
Byte @0011	0 0 1 1	0 0 1	—	—	—	—	H	—	—	—	—	—	—	—	—	—	—	—	—
Byte @0100	0 1 0 0	0 0 1	—	—	—	—	—	H	—	—	—	—	—	—	—	—	—	—	—
Byte @0101	0 1 0 1	0 0 1	—	—	—	—	—	—	H	—	—	—	—	—	—	—	—	—	—
Byte @0110	0 1 1 0	0 0 1	—	—	—	—	—	—	—	H	—	—	—	—	—	—	—	—	—
Byte @0111	0 1 1 1	0 0 1	—	—	—	—	—	—	—	—	H	—	—	—	—	—	—	—	—
Byte @1000	1 0 0 0	0 0 1	—	—	—	—	—	—	—	—	—	H	—	—	—	—	—	—	—
Byte @1001	1 0 0 1	0 0 1	—	—	—	—	—	—	—	—	—	—	H	—	—	—	—	—	—
Byte @1010	1 0 1 0	0 0 1	—	—	—	—	—	—	—	—	—	—	—	H	—	—	—	—	—
Byte @1011	1 0 1 1	0 0 1	—	—	—	—	—	—	—	—	—	—	—	—	H	—	—	—	—
Byte @1100	1 1 0 0	0 0 1	—	—	—	—	—	—	—	—	—	—	—	—	—	H	—	—	—
Byte @1101	1 1 0 1	0 0 1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	H	—	—
Byte @1110	1 1 1 0	0 0 1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	H	—
Byte @1111	1 1 1 1	0 0 1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	H

Table 15-12. Big- and little-endian memory storage (continued)

Program size and byte offset	A(28:31)	TSIZ (0:2)	ELSIZ (0:1)	(0... data bus byte lanes ...63) even double word — 0								(0... data bus byte lanes ...63) 0dd double word — 1							
				B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7	B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7
B. E. Half @0000	0 0 0 0	0 1 0	—	G	H	—	—	—	—	—	—	—	—	—	—	—	—	—	—
B. E. Half @0001	0 0 0 1	0 1 0	—	—	G	H	—	—	—	—	—	—	—	—	—	—	—	—	—
B. E. Half @0010	0 0 1 0	0 1 0	—	—	—	G	H	—	—	—	—	—	—	—	—	—	—	—	—
B. E. Half @0011	0 0 1 1	0 1 0	—	—	—	—	G	H	—	—	—	—	—	—	—	—	—	—	—
B. E. Half @0100	0 1 0 0	0 1 0	—	—	—	—	—	G	H	—	—	—	—	—	—	—	—	—	—
B. E. Half @0101	0 1 0 1	0 1 0	—	—	—	—	—	—	G	H	—	—	—	—	—	—	—	—	—
B. E. Half @0110	0 1 1 0	0 1 0	—	—	—	—	—	—	—	G	H	—	—	—	—	—	—	—	—
B. E. Half @0111	0 1 1 1	0 1 0	—	—	—	—	—	—	—	—	G	—	—	—	—	—	—	—	—
	1 0 0 0	0 0 1	—	—	—	—	—	—	—	—	—	H	—	—	—	—	—	—	—
B. E. Half @1000	1 0 0 0	0 1 0	—	—	—	—	—	—	—	—	—	G	H	—	—	—	—	—	—
B. E. Half @1001	1 0 0 1	0 1 0	—	—	—	—	—	—	—	—	—	—	G	H	—	—	—	—	—
B. E. Half @1010	1 0 1 0	0 1 0	—	—	—	—	—	—	—	—	—	—	—	G	H	—	—	—	—
B. E. Half @1011	1 0 1 1	0 1 0	—	—	—	—	—	—	—	—	—	—	—	—	G	H	—	—	—
B. E. Half @1100	1 1 0 0	0 1 0	—	—	—	—	—	—	—	—	—	—	—	—	—	G	H	—	—
B. E. Half @1101	1 1 0 1	0 1 0	—	—	—	—	—	—	—	—	—	—	—	—	—	—	G	H	—
B. E. Half @1110	1 1 1 0	0 1 0	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	G	H
B. E. Half @1111	1 1 1 1	0 1 0	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	G
	0 0 0 0 (next double-word)	0 0 1	—	H	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
L E. Half @0000	0 0 0 0	0 1 0	1x	H	G	—	—	—	—	—	—	—	—	—	—	—	—	—	—

Table 15-12. Big- and little-endian memory storage (continued)

Program size and byte offset	A(28:31)	TSIZ (0:2)	ELSIZ (0:1)	(0... data bus byte lanes ...63) even double word — 0								(0... data bus byte lanes ...63) 0dd double word — 1							
				B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7	B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7
L E. Half @0000	0 0 0 0	0 1 0	0x	G	H	—	—	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Half @0001	0 0 0 1	0 1 0	1x	—	H	G	—	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Half @0001	0 0 0 1	0 1 0	0x	—	G	H	—	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Half @0010	0 0 1 0	0 1 0	1x	—	—	H	G	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Half @0010	0 0 1 0	0 1 0	0x	—	—	G	H	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Half @0011	0 0 1 1	0 1 0	1x	—	—	—	H	G	—	—	—	—	—	—	—	—	—	—	—
L. E. Half @0011	0 0 1 1	0 1 0	0x	—	—	—	G	H	—	—	—	—	—	—	—	—	—	—	—
L. E. Half @0100	0 1 0 0	0 1 0	1x	—	—	—	—	H	G	—	—	—	—	—	—	—	—	—	—
L. E. Half @0100	0 1 0 0	0 1 0	0x	—	—	—	—	G	H	—	—	—	—	—	—	—	—	—	—
L. E. Half @0101	0 1 0 1	0 1 0	1x	—	—	—	—	—	H	G	—	—	—	—	—	—	—	—	—
L. E. Half @0101	0 1 0 1	0 1 0	0x	—	—	—	—	—	G	H	—	—	—	—	—	—	—	—	—
L. E. Half @0110	0 1 1 0	0 1 0	1x	—	—	—	—	—	—	H	G	—	—	—	—	—	—	—	—
L. E. Half @0110	0 1 1 0	0 1 0	0x	—	—	—	—	—	—	G	H	—	—	—	—	—	—	—	—
L. E. Half @0111	0 1 1 1	0 1 0	1x	—	—	—	—	—	—	—	H	—	—	—	—	—	—	—	—
	1 0 0 0	0 0 1	1x	—	—	—	—	—	—	—	—	G	—	—	—	—	—	—	—
L. E. Half @0111	0 1 1 1	0 1 0	0x	—	—	—	—	—	—	—	G	—	—	—	—	—	—	—	—
	1 0 0 0	0 0 1	0x	—	—	—	—	—	—	—	—	H	—	—	—	—	—	—	—
L. E. Half @1000	1 0 0 0	0 1 0	1x	—	—	—	—	—	—	—	—	H	G	—	—	—	—	—	—
L. E. Half @1000	1 0 0 0	0 1 0	0x	—	—	—	—	—	—	—	—	G	H	—	—	—	—	—	—
L. E. Half @1001	1 0 0 1	0 1 0	1x	—	—	—	—	—	—	—	—	—	H	G	—	—	—	—	—

Table 15-12. Big- and little-endian memory storage (continued)

Program size and byte offset	A(28:31)	TSIZ (0:2)	ELSIZ (0:1)	(0... data bus byte lanes ...63) even double word — 0								(0... data bus byte lanes ...63) 0dd double word — 1							
				B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7	B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7
L. E. Half @1001	1 0 0 1	0 1 0	0x	—	—	—	—	—	—	—	—	—	G	H	—	—	—	—	—
L. E. Half @1010	1 0 1 0	0 1 0	1x	—	—	—	—	—	—	—	—	—	—	H	G	—	—	—	—
L. E. Half @1010	1 0 1 0	0 1 0	0x	—	—	—	—	—	—	—	—	—	—	G	H	—	—	—	—
L. E. Half @1011	1 0 1 1	0 1 0	1x	—	—	—	—	—	—	—	—	—	—	—	H	G	—	—	—
L. E. Half @1011	1 0 1 1	0 1 0	0x	—	—	—	—	—	—	—	—	—	—	—	G	H	—	—	—
L. E. Half @1100	1 1 0 0	0 1 0	1x	—	—	—	—	—	—	—	—	—	—	—	—	H	G	—	—
L. E. Half @1100	1 1 0 0	0 1 0	0x	—	—	—	—	—	—	—	—	—	—	—	—	G	H	—	—
L. E. Half @1101	1 1 0 1	0 1 0	1x	—	—	—	—	—	—	—	—	—	—	—	—	—	H	G	—
L. E. Half @1101	1 1 0 1	0 1 0	0x	—	—	—	—	—	—	—	—	—	—	—	—	—	G	H	—
L. E. Half @1110	1 1 1 0	0 1 0	1x	—	—	—	—	—	—	—	—	—	—	—	—	—	—	H	G
L. E. Half @1110	1 1 1 0	0 1 0	0x	—	—	—	—	—	—	—	—	—	—	—	—	—	—	G	H
L. E. Half @1111	1 1 1 1	0 1 0	1x	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	H
	+ 0 0 0 0 (next double-word)	0 0 1	1x	G	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Half @1111	1 1 1 1	0 1 0	0x	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	G
	+ 0 0 0 0 (next double-word)	0 0 1	0x	H	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
B. E. Word @0000	0 0 0 0	1 0 0	—	E	F	G	H	—	—	—	—	—	—	—	—	—	—	—	—
B. E. Word @0001	0 0 0 1	1 0 0	—	—	E	F	G	H	—	—	—	—	—	—	—	—	—	—	—
B. E. Word @0010	0 0 1 0	1 0 0	—	—	—	E	F	G	H	—	—	—	—	—	—	—	—	—	—

Table 15-12. Big- and little-endian memory storage (continued)

Program size and byte offset	A(28:31)	TSIZ (0:2)	ELSIZ (0:1)	(0... data bus byte lanes ...63) even double word — 0								(0... data bus byte lanes ...63) 0dd double word — 1							
				B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7	B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7
B. E. Word @0011	0 0 1 1	1 0 0	—	—	—	—	E	F	G	H	—	—	—	—	—	—	—	—	—
B. E. Word @0100	0 1 0 0	1 0 0	—	—	—	—	—	E	F	G	H	—	—	—	—	—	—	—	—
B. E. Word @0101	0 1 0 1	1 0 0	—	—	—	—	—	—	E	F	G	—	—	—	—	—	—	—	—
	1 0 0 0	0 0 1	—	—	—	—	—	—	—	—	—	H	—	—	—	—	—	—	—
B. E. Word @0110	0 1 1 0	1 0 0	—	—	—	—	—	—	—	E	F	—	—	—	—	—	—	—	—
	1 0 0 0	0 1 0	—	—	—	—	—	—	—	—	—	G	H	—	—	—	—	—	—
B. E. Word @0111	0 1 1 1	1 0 0	—	—	—	—	—	—	—	—	E	—	—	—	—	—	—	—	—
	1 0 0 0	0 1 1	—	—	—	—	—	—	—	—	—	F	G	H	—	—	—	—	—
B. E. Word @1000	1 0 0 0	1 0 0	—	—	—	—	—	—	—	—	—	E	F	G	H	—	—	—	—
B. E. Word @1001	1 0 0 1	1 0 0	—	—	—	—	—	—	—	—	—	—	E	F	G	H	—	—	—
B. E. Word @1010	1 0 1 0	1 0 0	—	—	—	—	—	—	—	—	—	—	—	E	F	G	H	—	—
B. E. Word @1011	1 0 1 1	1 0 0	—	—	—	—	—	—	—	—	—	—	—	—	E	F	G	H	—
B. E. Word @1100	1 1 0 0	1 0 0	—	—	—	—	—	—	—	—	—	—	—	—	—	E	F	G	H
B. E. Word @1101	1 1 0 1	1 0 0	—	—	—	—	—	—	—	—	—	—	—	—	—	—	E	F	G
	+ 0 0 0 0 (next double-word)	0 0 1	—	H	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
B. E. Word @1110	1 1 1 0	1 0 0	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	E	F
	+ 0 0 0 0 (next double-word)	0 1 0	—	G	H	—	—	—	—	—	—	—	—	—	—	—	—	—	—
B. E. Word @1111	1 1 1 1	1 0 0	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	E
	+ 0 0 0 0 (next double-word)	0 1 1	—	F	G	H	—	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Word @0000	0 0 0 0	1 0 0	0 0	H	G	F	E	—	—	—	—	—	—	—	—	—	—	—	—

Table 15-12. Big- and little-endian memory storage (continued)

Program size and byte offset	A(28:31)	TSIZ (0:2)	ELSIZ (0:1)	(0... data bus byte lanes ...63) even double word — 0								(0... data bus byte lanes ...63) 0dd double word — 1							
				B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7	B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7
L. E. Word @0000	0 0 0 0	1 0 0	0 1	E	F	G	H	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Word @0000	0 0 0 0	1 0 0	1 x[1]	F	E	H	G	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Word @0001	0 0 0 1	1 0 0	0 0	—	H	G	F	E	—	—	—	—	—	—	—	—	—	—	—
L. E. Word @0001	0 0 0 1	1 0 0	0 1	—	E	F	G	H	—	—	—	—	—	—	—	—	—	—	—
L. E. Word @0001	0 0 0 1	1 0 0	1 x[1]	—	F	E	H	G	—	—	—	—	—	—	—	—	—	—	—
L. E. Word @0010	0 0 1 0	1 0 0	0 0	—	—	H	G	F	E	—	—	—	—	—	—	—	—	—	—
L. E. Word @0010	0 0 1 0	1 0 0	0 1	—	—	E	F	G	H	—	—	—	—	—	—	—	—	—	—
L. E. Word @0010	0 0 1 0	1 0 0	1 x[1]	—	—	F	E	H	G	—	—	—	—	—	—	—	—	—	—
L. E. Word @0011	0 0 1 1	1 0 0	0 0	—	—	—	H	G	F	E	—	—	—	—	—	—	—	—	—
L. E. Word @0011	0 0 1 1	1 0 0	0 1	—	—	—	E	F	G	H	—	—	—	—	—	—	—	—	—
L. E. Word @0011	0 0 1 1	1 0 0	1 x[1]	—	—	—	F	E	H	G	—	—	—	—	—	—	—	—	—
L. E. Word @0100	0 1 0 0	1 0 0	0 0	—	—	—	—	H	G	F	E	—	—	—	—	—	—	—	—
L. E. Word @0100	0 1 0 0	1 0 0	0 1	—	—	—	—	E	F	G	H	—	—	—	—	—	—	—	—
L. E. Word @0100	0 1 0 0	1 0 0	1 x[1]	—	—	—	—	F	E	H	G	—	—	—	—	—	—	—	—
L. E. Word @0101	0 1 0 1	1 0 0	0 0	—	—	—	—	—	H	G	F	—	—	—	—	—	—	—	—
	1 0 0 0	0 0 1	0 0	—	—	—	—	—	—	—	—	E	—	—	—	—	—	—	—
L. E. Word @0101	0 1 0 1	1 0 0	0 1	—	—	—	—	—	E	F	G	—	—	—	—	—	—	—	—
	1 0 0 0	0 0 1	0 1	—	—	—	—	—	—	—	—	H	—	—	—	—	—	—	—
L. E. Word @0101	0 1 0 1	1 0 0	1 x[1]	—	—	—	—	—	F	E	H	—	—	—	—	—	—	—	—
	1 0 0 0	0 0 1	1 x[1]	—	—	—	—	—	—	—	—	G	—	—	—	—	—	—	—
L. E. Word @0110	0 1 1 0	1 0 0	0 0	—	—	—	—	—	—	H	G	—	—	—	—	—	—	—	—
	1 0 0 0	0 1 0	0 0	—	—	—	—	—	—	—	—	F	E	—	—	—	—	—	—

Table 15-12. Big- and little-endian memory storage (continued)

Program size and byte offset	A(28:31)	TSIZ (0:2)	ELSIZ (0:1)	(0... data bus byte lanes ...63) even double word — 0								(0... data bus byte lanes ...63) 0dd double word — 1							
				B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7	B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7
L. E. Word @0110	0 1 1 0	1 0 0	0 1	—	—	—	—	—	—	E	F	—	—	—	—	—	—	—	—
	1 0 0 0	0 1 0	0 1	—	—	—	—	—	—	—	—	G	H	—	—	—	—	—	—
L. E. Word @0110	0 1 1 0	1 0 0	1 x[1]	—	—	—	—	—	—	F	E	—	—	—	—	—	—	—	—
	1 0 0 0	0 1 0	1 x[1]	—	—	—	—	—	—	—	—	H	G	—	—	—	—	—	—
L. E. Word @0111	0 1 1 1	1 0 0	0 0	—	—	—	—	—	—	—	H	—	—	—	—	—	—	—	—
	1 0 0 0	0 1 1	0 0	—	—	—	—	—	—	—	—	G	F	E	—	—	—	—	—
L. E. Word @0111	0 1 1 1	1 0 0	0 1	—	—	—	—	—	—	—	E	—	—	—	—	—	—	—	—
	1 0 0 0	0 1 1	0 1	—	—	—	—	—	—	—	—	F	G	H	—	—	—	—	—
L. E. Word @0111	0 1 1 1	1 0 0	1 x[1]	—	—	—	—	—	—	—	F	—	—	—	—	—	—	—	—
	1 0 0 0	0 1 1	1 x[1]	—	—	—	—	—	—	—	—	E	H	G	—	—	—	—	—
L. E. Word @1000	1 0 0 0	1 0 0	0 0	—	—	—	—	—	—	—	—	H	G	F	E	—	—	—	—
L. E. Word @1000	1 0 0 0	1 0 0	0 1	—	—	—	—	—	—	—	—	E	F	G	H	—	—	—	—
L. E. Word @1000	1 0 0 0	1 0 0	1 x[1]	—	—	—	—	—	—	—	—	F	E	H	G	—	—	—	—
L. E. Word @1001	1 0 0 1	1 0 0	0 0	—	—	—	—	—	—	—	—	—	H	G	F	E	—	—	—
L. E. Word @1001	1 0 0 1	1 0 0	0 1	—	—	—	—	—	—	—	—	—	E	F	G	H	—	—	—
L. E. Word @1001	1 0 0 1	1 0 0	1 x[1]	—	—	—	—	—	—	—	—	—	F	E	H	G	—	—	—
L. E. Word @1010	1 0 1 0	1 0 0	0 0	—	—	—	—	—	—	—	—	—	—	H	G	F	E	—	—
L. E. Word @1010	1 0 1 0	1 0 0	0 1	—	—	—	—	—	—	—	—	—	—	E	F	G	H	—	—
L. E. Word @1010	1 0 1 0	1 0 0	1 x[1]	—	—	—	—	—	—	—	—	—	—	F	E	H	G	—	—
L. E. Word @1011	1 0 1 1	1 0 0	0 0	—	—	—	—	—	—	—	—	—	—	—	H	G	F	E	—
L. E. Word @1011	1 0 1 1	1 0 0	0 1	—	—	—	—	—	—	—	—	—	—	—	E	F	G	H	—
L. E. Word @1011	1 0 1 1	1 0 0	1 x[1]	—	—	—	—	—	—	—	—	—	—	—	F	E	H	G	—

Table 15-12. Big- and little-endian memory storage (continued)

Program size and byte offset	A(28:31)	TSIZ (0:2)	ELSIZ (0:1)	(0... data bus byte lanes ...63) even double word — 0								(0... data bus byte lanes ...63) 0dd double word — 1							
				B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7	B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7
L. E. Word @1100	1 1 0 0	1 0 0	0 0	—	—	—	—	—	—	—	—	—	—	—	—	H	G	F	E
L. E. Word @1100	1 1 0 0	1 0 0	0 1	—	—	—	—	—	—	—	—	—	—	—	—	E	F	G	H
L. E. Word @1100	1 1 0 0	1 0 0	1 x[1]	—	—	—	—	—	—	—	—	—	—	—	—	F	E	H	G
L. E. Word @1101	1 1 0 1	1 0 0	0 0	—	—	—	—	—	—	—	—	—	—	—	—	—	H	G	F
	+ 0 0 0 0 (next double-word)	0 0 1	0 0	E	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Word @1101	1 1 0 1	1 0 0	0 1	—	—	—	—	—	—	—	—	—	—	—	—	—	E	F	G
	+ 0 0 0 0 (next double-word)	0 0 1	0 1	H	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Word @1101	1 1 0 1	1 0 0	1 x[1]	—	—	—	—	—	—	—	—	—	—	—	—	—	F	E	H
	+ 0 0 0 0 (next double-word)	0 0 1	1 x[1]	G	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Word @1110	1 1 1 0	1 0 0	0 0	—	—	—	—	—	—	—	—	—	—	—	—	—	—	H	G
	+ 0 0 0 0 (next double-word)	0 1 0	0 0	F	E	—	—	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Word @1110	1 1 1 0	1 0 0	0 1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	E	F
	+ 0 0 0 0 (next double-word)	0 1 0	0 1	G	H	—	—	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Word @1110	1 1 1 0	1 0 0	1 x[1]	—	—	—	—	—	—	—	—	—	—	—	—	—	—	F	E
	+ 0 0 0 0 (next double-word)	0 1 0	1 x[1]	H	G	—	—	—	—	—	—	—	—	—	—	—	—	—	—

Table 15-12. Big- and little-endian memory storage (continued)

Program size and byte offset	A(28:31)	TSIZ (0:2)	ELSIZ (0:1)	(0... data bus byte lanes ...63) even double word — 0								(0... data bus byte lanes ...63) 0dd double word — 1							
				B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7	B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7
L. E. Word @1111	1 1 1 1	1 0 0	0 0	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	H
	+ 0 0 0 0 (next double-word)	0 1 1	0 0	G	F	E	—	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Word @1111	1 1 1 1	1 0 0	0 1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	E
	+ 0 0 0 0 (next double-word)	0 1 1	0 1	F	G	H	—	—	—	—	—	—	—	—	—	—	—	—	—
L. E. Word @1111	1 1 1 1	1 0 0	1 x[1]	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	F
	+ 0 0 0 0 (next double-word)	0 1 1	1 x[1]	E	H	G	—	—	—	—	—	—	—	—	—	—	—	—	—
B.E. Doubleword @0000	0 0 0 0	0 0 0	—	A	B	C	D	E	F	G	H	—	—	—	—	—	—	—	—
B.E. Doubleword @0001	0 0 0 1	0 0 0	—	—	A	B	C	D	E	F	G	—	—	—	—	—	—	—	—
	1 0 0 0	0 0 1	—	—	—	—	—	—	—	—	—	H	—	—	—	—	—	—	—
B.E. Doubleword @0010	0 0 1 0	0 0 0	—	—	—	A	B	C	D	E	F	—	—	—	—	—	—	—	—
	1 0 0 0	0 1 0	—	—	—	—	—	—	—	—	—	G	H	—	—	—	—	—	—
B.E. Doubleword @0011	0 0 1 1	0 0 0	—	—	—	—	A	B	C	D	E	—	—	—	—	—	—	—	—
	1 0 0 0	0 1 1	—	—	—	—	—	—	—	—	—	F	G	H	—	—	—	—	—
B.E. Doubleword @0100	0 1 0 0	0 0 0	—	—	—	—	—	A	B	C	D	—	—	—	—	—	—	—	—
	1 0 0 0	1 0 0	—	—	—	—	—	—	—	—	—	E	F	G	H	—	—	—	—
B.E. Doubleword @0101	0 1 0 1	0 0 0	—	—	—	—	—	—	A	B	C	—	—	—	—	—	—	—	—
	1 0 0 0	1 0 1	—	—	—	—	—	—	—	—	—	D	E	F	G	H	—	—	—
B.E. Doubleword @0110	0 1 1 0	0 0 0	—	—	—	—	—	—	—	A	B	—	—	—	—	—	—	—	—
	1 0 0 0	1 1 0	—	—	—	—	—	—	—	—	—	C	D	E	F	G	H	—	—
B.E. Doubleword @0111	0 1 1 1	0 0 0	—	—	—	—	—	—	—	—	A	—	—	—	—	—	—	—	—
	1 0 0 0	1 1 1	—	—	—	—	—	—	—	—	—	B	C	D	E	F	G	H	—

Table 15-12. Big- and little-endian memory storage (continued)

Program size and byte offset	A(28:31)	TSIZ (0:2)	ELSIZ (0:1)	(0... data bus byte lanes ...63) even double word — 0								(0... data bus byte lanes ...63) 0dd double word — 1							
				B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7	B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7
B.E. Doubleword @1000	1 0 0 0	0 0 0	—	—	—	—	—	—	—	—	—	A	B	C	D	E	F	G	H
B.E. Doubleword @1001	1 0 0 1	0 0 0	—	—	—	—	—	—	—	—	—	—	A	B	C	D	E	F	G
	+0 0 0 0	0 0 1	—	H	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
B.E. Doubleword @1010	1 0 1 0	0 0 0	—	—	—	—	—	—	—	—	—	—	—	A	B	C	D	E	F
	+0 0 0 0	0 1 0	—	G	H	—	—	—	—	—	—	—	—	—	—	—	—	—	—
B.E. Doubleword @1011	1 0 1 1	0 0 0	—	—	—	—	—	—	—	—	—	—	—	—	A	B	C	D	E
	+0 0 0 0	0 1 1	—	F	G	H	—	—	—	—	—	—	—	—	—	—	—	—	—
B.E. Doubleword @1100	1 1 0 0	0 0 0	—	—	—	—	—	—	—	—	—	—	—	—	—	A	B	C	D
	+0 0 0 0	1 0 0	—	E	F	G	H	—	—	—	—	—	—	—	—	—	—	—	—
B.E. Doubleword @1101	1 1 0 1	0 0 0	—	—	—	—	—	—	—	—	—	—	—	—	—	—	A	B	C
	+0 0 0 0	1 0 1	—	D	E	F	G	H	—	—	—	—	—	—	—	—	—	—	—
B.E. Doubleword @1110	1 1 1 0	0 0 0	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	A	B
	+0 0 0 0	1 1 0	—	C	D	E	F	G	H	—	—	—	—	—	—	—	—	—	—
B.E. Doubleword @1111	1 1 1 1	0 0 0	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	A
	+0 0 0 0	1 1 1	—	B	C	D	E	F	G	H	—	—	—	—	—	—	—	—	—
B.E. Doubleword (word pairs) @-000	- 0 0 0	0 0 0	—	E	F	G	H	M	N	O	P	—	—	—	—	—	—	—	—
B.E. Doubleword (word pairs) @-100	- 1 0 0	0 0 0	—	—	—	—	—	E	F	G	H	—	—	—	—	—	—	—	—
	+- 0 0 0 (next double-word)	1 0 0	—	—	—	—	—	—	—	—	—	M	N	O	P	—	—	—	—
L.E. Doubleword @0000	0 0 0 0	0 0 0	1 1	H	G	F	E	D	C	B	A	—	—	—	—	—	—	—	—
L.E. Doubleword (byte elements) @0000	0 0 0 0	0 0 0	0 1	A	B	C	D	E	F	G	H	—	—	—	—	—	—	—	—

Table 15-12. Big- and little-endian memory storage (continued)

Program size and byte offset	A(28:31)	TSIZ (0:2)	ELSIZ (0:1)	(0... data bus byte lanes ...63) even double word — 0								(0... data bus byte lanes ...63) 0dd double word — 1							
				B0	B1	B2	B3	B4	B5	B6	B7	B0	B1	B2	B3	B4	B5	B6	B7
L.E. Doubleword (halfword elements) @0000	00 0 0	0 0 0	1 0[1]	B	A	D	C	F	E	H	G	—	—	—	—	—	—	—	—
L.E. Doubleword (word elements) @0000	00 0 0	0 0 0	0 0[1]	D	C	B	A	H	G	F	E	—	—	—	—	—	—	—	—
L.E. Doubleword @0001	0 0 0 1	0 0 0	1 1	—	H	G	F	E	D	C	B	—	—	—	—	—	—	—	—
	+- 0 0 0 (next double-word)	0 0 1	1 1	—	—	—	—	—	—	—	—	A	—	—	—	—	—	—	—
L.E. Doubleword (byte elements) @0001	0 0 0 1	0 0 0	0 1	—	A	B	C	D	E	F	G	—	—	—	—	—	—	—	—
	+- 0 0 0 (next double-word)	0 0 1	0 1	—	—	—	—	—	—	—	—	H	—	—	—	—	—	—	—
L.E. Doubleword (halfword elements) @0001	0 0 0 1	0 0 0	1 0[1]	—	B	A	D	C	F	E	H	—	—	—	—	—	—	—	—
	1 0 0 0 (next double-word)	0 0 1	1 0[1]	—	—	—	—	—	—	—	—	G	—	—	—	—	—	—	—
L.E. Doubleword (word elements) @0001	0 0 0 1	0 0 0	0 0[1]	—	D	C	B	A	H	G	F	—	—	—	—	—	—	—	—
	1 0 0 0 (next double-word)	0 0 1	0 0[1]	—	—	—	—	—	—	—	—	E	—	—	—	—	—	—	—
L.E. Doubleword @0010	0 0 1 0	0 0 0	1 1	—	H	G	F	E	D	C	B	—	—	—	—	—	—	—	—
	1 0 0 0 (next double-word)	0 1 0	1 1	—	—	—	—	—	—	—	—	A	—	—	—	—	—	—	—

Table 15-12. Big- and little-endian memory storage (continued)

Program size and byte offset	A(28:31)	TSIZ (0:2)	ELSIZ (0:1)	(0... data bus byte lanes ...63) even double word — 0								(0... data bus byte lanes ...63) 0dd double word — 1							
				B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7	B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7
L.E. Doubleword (byte elements) @0010	0 0 1 0	0 0 0	0 1	—	—	A	B	C	D	E	F	—	—	—	—	—	—	—	—
	1 0 0 0 (next double-word)	0 1 0	0 1	—	—	—	—	—	—	—	—	G	H	—	—	—	—	—	—
L.E. Doubleword (halfword elements) @0010	0 0 1 0	0 0 0	1 0[1]	—	—	B	A	D	C	F	E	—	—	—	—	—	—	—	—
	1 0 0 0 (next double-word)	0 1 0	1 0[1]	—	—	—	—	—	—	—	—	H	G	—	—	—	—	—	—
L.E. Doubleword (word elements) @0010	0 0 1 0	0 0 0	0 0[1]	—	—	D	C	B	A	H	G	—	—	—	—	—	—	—	—
	1 0 0 0 (next double-word)	0 1 0	0 0[1]	—	—	—	—	—	—	—	—	F	E	—	—	—	—	—	—
L.E. Doubleword @0011	0 0 1 1	0 0 0	1 1	—	—	—	H	G	F	E	D	—	—	—	—	—	—	—	—
	1 0 0 0 (next double-word)	0 1 1	1 1	—	—	—	—	—	—	—	—	C	B	A	—	—	—	—	—
L.E. Doubleword (byte elements) @0011	0 0 1 1	0 0 0	0 1	—	—	—	A	B	C	D	E	—	—	—	—	—	—	—	—
	1 0 0 0 (next double-word)	0 1 1	0 1	—	—	—	—	—	—	—	—	F	G	H	—	—	—	—	—
L.E. Doubleword (halfword elements) @0011	0 0 1 1	0 0 0	1 0[1]	—	—	—	B	A	D	C	F	—	—	—	—	—	—	—	—
	1 0 0 0 (next double-word)	0 1 1	1 0[1]	—	—	—	—	—	—	—	—	E	H	G	—	—	—	—	—
L.E. Doubleword (word elements) @0011	0 0 1 1	0 0 0	0 0[1]	—	—	—	D	C	B	A	H	—	—	—	—	—	—	—	—
	1 0 0 0 (next double-word)	0 1 1	0 0[1]	—	—	—	—	—	—	—	—	G	F	E	—	—	—	—	—

Table 15-12. Big- and little-endian memory storage (continued)

Program size and byte offset	A(28:31)	TSIZ (0:2)	ELSIZ (0:1)	(0... data bus byte lanes ...63) even double word — 0								(0... data bus byte lanes ...63) 0dd double word — 1							
				B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7	B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7
L.E. Doubleword @0100	0 1 0 0	0 0 0	1 1	—	—	—	—	H	G	F	E	—	—	—	—	—	—	—	—
	1 0 0 0 (next double-word)	1 0 0	1 1	—	—	—	—	—	—	—	—	D	C	B	A	—	—	—	—
L.E. Doubleword (byte elements) @0100	0 1 0 0	0 0 0	0 1	—	—	—	—	A	B	C	D	—	—	—	—	—	—	—	—
	1 0 0 0 (next double-word)	1 0 0	0 1	—	—	—	—	—	—	—	—	E	F	G	H	—	—	—	—
L.E. Doubleword (halfword elements) @0100	0 1 0 0	0 0 0	1 0[1]	—	—	—	—	B	A	D	C	—	—	—	—	—	—	—	—
	1 0 0 0 (next double-word)	1 0 0	1 0[1]	—	—	—	—	—	—	—	—	F	E	H	G	—	—	—	—
L.E. Doubleword (word elements) @0100	0 1 0 0	0 0 0	0 0[1]	—	—	—	—	D	C	B	A	—	—	—	—	—	—	—	—
	1 0 0 0 (next double-word)	1 0 0	0 0[1]	—	—	—	—	—	—	—	—	H	G	F	E	—	—	—	—
L.E. Doubleword @0101	0 1 0 1	0 0 0	1 1	—	—	—	—	—	H	G	F	—	—	—	—	—	—	—	—
	1 0 0 0 (next double-word)	1 0 1	1 1	—	—	—	—	—	—	—	—	E	D	C	B	A	—	—	—
L.E. Doubleword (byte elements) @0101	0 1 0 1	0 0 0	0 1	—	—	—	—	—	A	B	C	—	—	—	—	—	—	—	—
	1 0 0 0 (next double-word)	1 0 1	0 1	—	—	—	—	—	—	—	—	D	E	F	G	H	—	—	—
L.E. Doubleword (halfword elements) @0101	0 1 0 1	0 0 0	1 0[1]	—	—	—	—	—	B	A	D	—	—	—	—	—	—	—	—
	1 0 0 0 (next double-word)	1 0 1	1 0[1]	—	—	—	—	—	—	—	—	C	F	E	H	G	—	—	—

Table 15-12. Big- and little-endian memory storage (continued)

Program size and byte offset	A(28:31)	TSIZ (0:2)	ELSIZ (0:1)	(0... data bus byte lanes ...63) even double word — 0								(0... data bus byte lanes ...63) 0dd double word — 1							
				B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7	B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7
L.E. Doubleword (word elements) @0101	0 1 0 1	0 0 0	0 0[1]	—	—	—	—	—	D	C	B	—	—	—	—	—	—	—	—
	1 0 0 0 (next double-word)	1 0 1	0 0[1]	—	—	—	—	—	—	—	—	A	H	G	F	E	—	—	—
L.E. Doubleword @0110	0 1 1 0	0 0 0	1 1	—	—	—	—	—	—	H	G	—	—	—	—	—	—	—	—
	1 0 0 0 (next double-word)	1 1 0	1 1	—	—	—	—	—	—	—	—	F	E	D	C	B	A	—	—
L.E. Doubleword (byte elements) @0110	0 1 1 0	0 0 0	0 1	—	—	—	—	—	—	A	B	—	—	—	—	—	—	—	—
	1 0 0 0 (next double-word)	1 1 0	0 1	—	—	—	—	—	—	—	—	C	D	E	F	G	H	—	—
L.E. Doubleword (halfword elements) @0110	0 1 1 0	0 0 0	1 0[1]	—	—	—	—	—	—	B	A	—	—	—	—	—	—	—	—
	1 0 0 0 (next double-word)	1 1 0	1 0[1]	—	—	—	—	—	—	—	—	D	C	F	E	H	G	—	—
L.E. Doubleword (word elements) @0110	0 1 1 0	0 0 0	0 0[1]	—	—	—	—	—	—	D	C	—	—	—	—	—	—	—	—
	1 0 0 0 (next double-word)	1 1 0	0 0[1]	—	—	—	—	—	—	—	—	B	A	H	G	F	E	—	—
L.E. Doubleword @0111	0 1 1 1	0 0 0	1 1	—	—	—	—	—	—	—	H	—	—	—	—	—	—	—	—
	1 0 0 0 (next double-word)	1 1 1	1 1	—	—	—	—	—	—	—	—	G	F	E	D	C	B	A	—
L.E. Doubleword (byte elements) @0111	0 1 1 1	0 0 0	0 1	—	—	—	—	—	—	—	A	—	—	—	—	—	—	—	—
	1 0 0 0 (next double-word)	1 1 1	0 1	—	—	—	—	—	—	—	—	B	C	D	E	F	G	H	—

Table 15-12. Big- and little-endian memory storage (continued)

Program size and byte offset	A(28:31)	TSIZ (0:2)	ELSIZ (0:1)	(0... data bus byte lanes ...63) even double word — 0								(0... data bus byte lanes ...63) 0dd double word — 1							
				B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7	B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7
L.E. Doubleword (halfword elements) @0111	0 1 1 1	0 0 0	1 0[1]	—	—	—	—	—	—	—	B	—	—	—	—	—	—	—	—
	1 0 0 0 (next double-word)	1 1 1	1 0[1]	—	—	—	—	—	—	—	—	A	D	C	F	E	H	G	—
L.E. Doubleword (word elements) @0111	0 1 1 1	0 0 0	0 0[1]	—	—	—	—	—	—	—	D	—	—	—	—	—	—	—	—
	1 0 0 0 (next double-word)	1 1 1	0 0[1]	—	—	—	—	—	—	—	—	C	B	A	H	G	F	E	—
L.E. Doubleword @1000	1 0 0 0	0 0 0	1 1	—	—	—	—	—	—	—	—	H	G	F	E	D	C	B	A
L.E. Doubleword (byte elements) @1000	1 0 0 0	0 0 0	0 1	—	—	—	—	—	—	—	—	A	B	C	D	E	F	G	H
L.E. Doubleword (halfword elements) @1000	1 0 0 0	0 0 0	1 0[1]	—	—	—	—	—	—	—	—	B	A	D	C	F	E	H	G
L.E. Doubleword (word elements) @1000	1 0 0 0	0 0 0	0 0[1]	—	—	—	—	—	—	—	—	D	C	B	A	H	G	F	E
L.E. Doubleword @1001	1 0 0 1	0 0 0	1 1	—	—	—	—	—	—	—	—	—	H	G	F	E	D	C	B
	+0 0 0 0 (next double-word)	0 0 1	1 1	A	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
L.E. Doubleword (byte elements) @1001	1 0 0 1	0 0 0	0 1	—	—	—	—	—	—	—	—	—	A	B	C	D	E	F	G
	+0 0 0 0 (next double-word)	0 0 1	0 1	H	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—

Table 15-12. Big- and little-endian memory storage (continued)

Program size and byte offset	A(28:31)	TSIZ (0:2)	ELSIZ (0:1)	(0... data bus byte lanes ...63) even double word — 0								(0... data bus byte lanes ...63) 0dd double word — 1							
				B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7	B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7
L.E. Doubleword (halfword elements) @1001	1 0 0 1	0 0 0	1 0[1]	—	—	—	—	—	—	—	—	—	B	A	D	C	F	E	H
	+0 0 0 0 (next double-word)	0 0 1	1 0[1]	G	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
L.E. Doubleword (word elements) @1001	1 0 0 1	0 0 0	0 0[1]	—	—	—	—	—	—	—	—	—	D	C	B	A	H	G	F
	+0 0 0 0 (next double-word)	0 0 1	0 0[1]	E	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
L.E. Doubleword @1010	1 0 1 0	0 0 0	1 1	—	—	—	—	—	—	—	—	—	H	G	F	E	D	C	B
	+0 0 0 0 (next double-word)	0 1 0	1 1	A	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
L.E. Doubleword (byte elements) @1010	1 0 1 0	0 0 0	0 1	—	—	—	—	—	—	—	—	—	—	A	B	C	D	E	F
	+0 0 0 0 (next double-word)	0 1 0	0 1	G	H	—	—	—	—	—	—	—	—	—	—	—	—	—	—
L.E. Doubleword (halfword elements) @1010	1 0 1 0	0 0 0	1 0[1]	—	—	—	—	—	—	—	—	—	—	B	A	D	C	F	E
	+0 0 0 0 (next double-word)	0 1 0	1 0[1]	H	G	—	—	—	—	—	—	—	—	—	—	—	—	—	—
L.E. Doubleword (word elements) @1010	1 0 1 0	0 0 0	0 0[1]	—	—	—	—	—	—	—	—	—	—	D	C	B	A	H	G
	+0 0 0 0 (next double-word)	0 1 0	0 0[1]	F	E	—	—	—	—	—	—	—	—	—	—	—	—	—	—
L.E. Doubleword @1011	1 0 1 1	0 0 0	1 1	—	—	—	—	—	—	—	—	—	—	—	H	G	F	E	D
	+0 0 0 0 (next double-word)	0 1 1	1 1	C	B	A	—	—	—	—	—	—	—	—	—	—	—	—	—

Table 15-12. Big- and little-endian memory storage (continued)

Program size and byte offset	A(28:31)	TSIZ (0:2)	ELSIZ (0:1)	(0... data bus byte lanes ...63) even double word — 0								(0... data bus byte lanes ...63) 0dd double word — 1							
				B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7	B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7
L.E. Doubleword (byte elements) @1011	1 0 1 1	0 0 0	0 1	—	—	—	—	—	—	—	—	—	—	—	A	B	C	D	E
	+0 0 0 0 (next double-word)	0 1 1	0 1	F	G	H	—	—	—	—	—	—	—	—	—	—	—	—	—
L.E. Doubleword (halfword elements) @1011	1 0 1 1	0 0 0	1 0[1]	—	—	—	—	—	—	—	—	—	—	—	B	A	D	C	F
	+0 0 0 0 (next double-word)	0 1 1	1 0[1]	E	H	G	—	—	—	—	—	—	—	—	—	—	—	—	—
L.E. Doubleword (word elements) @1011	1 0 1 1	0 0 0	0 0[1]	—	—	—	—	—	—	—	—	—	—	—	D	C	B	A	H
	+0 0 0 0 (next double-word)	0 1 1	0 0[1]	G	F	E	—	—	—	—	—	—	—	—	—	—	—	—	—
L.E. Doubleword @1100	1 1 0 0	0 0 0	1 1	—	—	—	—	—	—	—	—	—	—	—	—	H	G	F	E
	+0 0 0 0 (next double-word)	1 0 0	1 1	D	C	B	A	—	—	—	—	—	—	—	—	—	—	—	—
L.E. Doubleword (byte elements) @1100	1 1 0 0	0 0 0	0 1	—	—	—	—	—	—	—	—	—	—	—	—	A	B	C	D
	+0 0 0 0 (next double-word)	1 0 0	0 1	E	F	G	H	—	—	—	—	—	—	—	—	—	—	—	—
L.E. Doubleword (halfword elements) @1100	1 1 0 0	0 0 0	1 0[1]	—	—	—	—	—	—	—	—	—	—	—	—	B	A	D	C
	+0 0 0 0 (next double-word)	1 0 0	1 0[1]	F	E	H	G	—	—	—	—	—	—	—	—	—	—	—	—
L.E. Doubleword (word elements) @1100	1 1 0 0	0 0 0	0 0[1]	—	—	—	—	—	—	—	—	—	—	—	—	D	C	B	A
	+0 0 0 0 (next double-word)	1 0 0	0 0[1]	H	G	F	E	—	—	—	—	—	—	—	—	—	—	—	—

Table 15-12. Big- and little-endian memory storage (continued)

Program size and byte offset	A(28:31)	TSIZ (0:2)	ELSIZ (0:1)	(0... data bus byte lanes ...63) even double word — 0								(0... data bus byte lanes ...63) 0dd double word — 1							
				B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7	B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7
L.E. Doubleword @1101	1 1 0 1	0 0 0	1 1	—	—	—	—	—	—	—	—	—	—	—	—	—	H	G	F
	+0 0 0 0 (next double-word)	1 0 1	1 1	E	D	C	B	A	—	—	—	—	—	—	—	—	—	—	—
L.E. Doubleword (byte elements) @1101	1 1 0 1	0 0 0	0 1	—	—	—	—	—	—	—	—	—	—	—	—	—	A	B	C
	+0 0 0 0 (next double-word)	1 0 1	0 1	D	E	F	G	H	—	—	—	—	—	—	—	—	—	—	—
L.E. Doubleword (halfword elements) @1101	1 1 0 1	0 0 0	1 0[1]	—	—	—	—	—	—	—	—	—	—	—	—	—	B	A	D
	+0 0 0 0 (next double-word)	1 0 1	1 0[1]	C	F	E	H	G	—	—	—	—	—	—	—	—	—	—	—
L.E. Doubleword (word elements) @1101	1 1 0 1	0 0 0	0 0[1]	—	—	—	—	—	—	—	—	—	—	—	—	—	D	C	B
	+0 0 0 0 (next double-word)	1 0 1	0 0[1]	A	H	G	F	E	—	—	—	—	—	—	—	—	—	—	—
L.E. Doubleword @1110	1 1 1 0	0 0 0	1 1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	H	G
	+0 0 0 0 (next double-word)	1 1 0	1 1	F	E	D	C	B	A	—	—	—	—	—	—	—	—	—	—
L.E. Doubleword (byte elements) @1110	1 1 1 0	0 0 0	0 1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	A	B
	+0 0 0 0 (next double-word)	1 1 0	0 1	C	D	E	F	G	H	—	—	—	—	—	—	—	—	—	—
L.E. Doubleword (halfword elements) @1110	1 1 1 0	0 0 0	1 0[1]	—	—	—	—	—	—	—	—	—	—	—	—	—	—	B	A
	+0 0 0 0 (next double-word)	1 1 0	1 0[1]	D	C	F	E	H	G	—	—	—	—	—	—	—	—	—	—

Table 15-12. Big- and little-endian memory storage (continued)

Program size and byte offset	A(28:31)	TSIZ (0:2)	ELSIZ (0:1)	(0... data bus byte lanes ...63) even double word — 0								(0... data bus byte lanes ...63) 0dd double word — 1							
				B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7	B0	B 1	B 2	B 3	B 4	B 5	B 6	B 7
L.E. Doubleword (word elements) @1110	1 1 1 0	0 0 0	0 0[1]	—	—	—	—	—	—	—	—	—	—	—	—	—	—	D	C
	+0 0 0 0 (next double-word)	1 1 0	0 0[1]	B	A	H	G	F	E	—	—	—	—	—	—	—	—	—	—
L.E. Doubleword @1111	1 1 1 1	0 0 0	1 1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	H
	+0 0 0 0 (next double-word)	1 1 1	1 1	G	F	E	D	C	B	A	—	—	—	—	—	—	—	—	—
L.E. Doubleword (byte elements) @1111	1 1 1 1	0 0 0	0 1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	A
	+0 0 0 0 (next double-word)	1 1 1	0 1	B	C	D	E	F	G	H	—	—	—	—	—	—	—	—	—
L.E. Doubleword (halfword elements) @1111	1 1 1 1	0 0 0	1 0[1]	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	B
	+0 0 0 0 (next double-word)	1 1 1	1 0[1]	A	D	C	F	E	H	G	—	—	—	—	—	—	—	—	—
L.E. Doubleword (word elements) @1111	1 1 1 1 +0 0 0 0 (next double-word)	0 0 0	0 0[1]	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	D
		1 1 1	0 0[1]	C	B	A	H	G	F	E	—	—	—	—	—	—	—	—	—
L.E. Doubleword (word pairs) @-000	- 0 0 0	0 0 0	0 0[1]	H	G	F	E	P	O	N	M	—	—	—	—	—	—	—	—
L.E. Doubleword (word pairs) @-100	- 1 0 0	0 0 0	0 0[1]	—	—	—	—	H	G	F	E	—	—	—	—	—	—	—	—
	+- 0 0 0 (next double-word)	1 0 0	0 0[1]	—	—	—	—	—	—	—	—	P	O	N	M	—	—	—	—

Table Notes:

Assumes a GP Register contains “A B C D E F G H”. Doubleword with word pair assumes a pair of registers containing ‘A B C D E F G H’, ‘I J K L M N O P’.

All other combinations of ELSIZ[0:1] are illegal and will not occur. ELSIZ ignored on reads, CPU will perform proper byte ordering.

1 Elements permuted

15.2.5 External SPR interface signals

The following paragraph describes interface signals for Special Purpose registers (SPRs) located externally to the core.

15.2.5.1 SPR number (p_sprnum[0:9])

The **p_sprnum[0:9]** signals are provided to indicate the particular external SPR that is being accessed. These signals are only valid during external SPR accesses.

15.2.5.2 SPR read data (p_spr_in[0:31])

The **p_spr_in[0:31]** input signals provide read data from an external SPR that is being accessed with a **mfspr** instruction. These signals are only sampled during external move from SPR accesses.

15.2.5.3 SPR write data (p_spr_out[0:31])

The **p_spr_out[0:31]** output signals provide data to write to an external SPR that is being accessed with a **mtspr** instruction. These signals are only valid during external move to SPR accesses.

15.2.5.4 SPR read control (p_rd_spr)

The **p_rd_spr** output signal indicates an external SPR read is occurring, the **p_sprnum[0:9]** outputs are valid, and that the **p_spr_in[0:31]** inputs will be sampled at the end of the next clock cycle.

15.2.5.5 SPR write control (p_wr_spr)

The **p_wr_spr** output signal indicates an external SPR write is occurring, the **p_sprnum[0:9]** outputs are valid, and the **p_spr_out[0:31]** outputs will be driven the next clock cycle.

15.2.6 Miscellaneous processor signals

The following paragraph describes several miscellaneous processor signals.

15.2.6.1 PID0 outputs (p_pid0[0:7])

The active-high **p_pid0[0:7]** output signals are used to provide the current process ID in the Process ID Register 0 (PID0). These outputs correspond to the low order eight bits of PID0.

15.2.6.2 PID0 update (p_pid0_updt)

The active-high **p_pid0_updt** signal is used to indicate that the Process ID Register 0 (PID0) is being updated by a **mtspr** instruction. This output will assert during the clock cycle the **p_pid0[0:7]** outputs are changing.

15.2.7 Cache/MMU status signals

The following paragraph describes several miscellaneous processor to Cache / MMU status signals.

15.2.7.1 Cache enabled (p_d_cache_enabled, p_i_cache_enabled)

The active-high **p_[d,i]_cache_enabled** input signal is used to indicate that the Cache is enabled.

15.2.7.2 Cache/MMU busy (p_d_cmbusy, p_i_cmbusy)

The active-high **p_[d,i]_cmbusy** input signal is used to indicate that the Cache or MMU is busy processing a translation request or an external bus access such as a cache line transfer or write buffer flush. This signal is used to handshake operation of **mfspr**/**mtspr** instructions that specify a Cache or MMU special purpose register, as well as Cache and MMU control instructions. Execution of these instructions will be stalled until all outstanding processor requests have been completed (**p_[d,i]_tbusy[0]_b** is negated) and Cache and MMU are idle. The Cache should assume responsibility for proper assertion of this signal.

15.2.7.3 Cache set CUL (p_d_set_cul, p_i_set_cul)

The active-high **p_[d,i]_set_cul** output signal is used to indicate that the Cache Unable to Lock (CUL) status bit should be set due to an attempt by the CPU to execute a cache line-locking instruction that was not allowed.

15.2.7.4 User cache lock DSI control (p_ucl_dsi)

The active-high **p_ucl_dsi** output signal is used to indicate that the CPU is attempting to execute a cache line-locking instruction that should not be allowed due to a UCLE exception, and should result in a DSI. The Cache should return a **p_tea_b** in response, and should not allow the cache lock/unlock to occur. The MMU however, attempts translation so that a TLB miss may be detected and be prioritized over the UCLE DSI.

15.2.7.5 Cache push parity error (p_d_cp_perr)

The active-high **p_d_cp_perr** input signal is used to indicate that a cache parity error has occurred while loading dirty data into the push buffer for replacement. This signal is asserted in an imprecise fashion. This signal is used to generate a machine check condition and causes the associated syndrome bit to be set in the Machine Check Syndrome register (Section 2.4.7, Machine Check Syndrome Register (MCSR)).

15.2.7.6 Cache push address (p_d_push_addr[0:31])

The **p_d_push_addr[0:31]** bus is used to provide the physical address of a push that incurs a cache parity error while loading dirty data into the push buffer for replacement. This bus is sampled with the assertion of **p_d_cp_perr** for error reporting purposes.

15.2.7.7 Bus write error (p_d_bus_wrerr)

The active-high **p_d_bus_wrerr** input signal is used to indicate that a bus error has occurred while emptying the cache store buffer or push buffer. This signal is asserted in an imprecise fashion This signal is used to generate a machine check condition and causes the associated syndrome bit to be set in the Machine Check Syndrome register (Section 2.4.7, Machine Check Syndrome Register (MCSR)).

15.2.7.8 Bus write error address (p_d_bus_wrerr_addr[0:31])

The **p_d_bus_wrerr_addr[0:31]** bus is used to provide the physical address of a bus write transfer that incurs a bus write error while emptying the cache store buffer or push buffer. This bus is sampled with the assertion of **p_d_bus_wrerr** for error reporting purposes.

15.2.7.9 Cache linefill status (p_d_lf_status[0:3], p_i_lf_status[0:3])

The active-high **p_[d,i]_lf_status[0:3]** input signals are used to provide linefill status information to the CPU. Table 15-13 details these signals.

Table 15-13. p_[d,i]_lf_status[0:3]

Signal	Description
lf_status[0]	Linefill was terminated by a bus error
lf_status[1]	Linefill was initiated by a store-type access
lf_status[2]	Linefill was initiated by a touch access
lf_status[3]	Linefill was completed without error

15.2.7.10 Linefill status address (p_d_lf_addr[0:31], p_i_lf_addr[0:31])

The **p_[d,i]_lf_addr[0:31]** bus is used to provide the physical address of a linefill transfer. This bus is sampled with the assertion of one of the corresponding **p_[d,i]_lf_status[0:3]** inputs for error reporting purposes.

15.2.7.11 Debug mode MMU disable (p_d_dmdis, p_i_dmdis)

The active-high **p_[d,i]_dmdis** output signal reflects the sampled state of the OnCE Control Register OCR[D_DMDIS] and OCR[I_DMDIS] bits. It will be negated when the debug session ends. See Section 12.4.6.3, e200z759n3 OnCE Control Register (OCR), for more information on this function.

15.2.7.12 Debug mode MMU 'VLE' attribute (p_dbg_vle)

The active-high **p_dbg_vle** output signal reflects the sampled state of the OnCE Control Register OCR[I_DVLE] bit. See Section 12.4.6.3, e200z759n3 OnCE Control Register (OCR), for more information on this function.

15.2.7.13 Debug mode MMU 'W' attribute (p_d_dbg_w)

The active-high **p_d_dbg_w** output signal reflects the sampled state of the OnCE Control Register OCR[D_DW] bit. See Section 12.4.6.3, e200z759n3 OnCE Control Register (OCR), for more information on this function.

15.2.7.14 Debug mode MMU 'I' attribute (p_d_dbg_i, p_i_dbg_i)

The active-high **p_[d,i]_dbg_i** output signal reflects the sampled state of the OnCE Control Register OCR[D_DI] and OCR[I_DI] bits. See Section 12.4.6.3, e200z759n3 OnCE Control Register (OCR), for more information on this function.

15.2.7.15 Debug mode MMU 'M' attribute (p_d_dbg_m, p_i_dbg_m)

The active-high **p_[d,i]_dbg_m** output signal reflects the sampled state of the OnCE Control Register OCR[D_DM] and OCR[I_DM] bits. See Section 12.4.6.3, e200z759n3 OnCE Control Register (OCR), for more information on this function.

15.2.7.16 Debug mode MMU 'G' attribute (p_d_dbg_g)

The active-high **p_d_dbg_g** output signal reflects the sampled state of the OnCE Control Register OCR[D_DG] bit. See Section 12.4.6.3, e200z759n3 OnCE Control Register (OCR), for more information on this function.

15.2.7.17 Debug mode MMU 'E' attribute (p_d_dbg_e, p_i_dbg_e)

The active-high **p_[d,i]_dbg_e** output signal reflects the state of the OnCE Control Register OCR[D_DE] and OCR[I_DE] bits. See Section 12.4.6.3, e200z759n3 OnCE Control Register (OCR), for more information on this function.

15.2.8 EFPU interface signals

Please refer to the *EFPU Interface Specification* for information on the EFPU interface signals.

15.2.9 Test signals

Please refer to the e200z759n3 *Test Guide* for information on Test signals.

15.3 Timing diagrams

15.3.1 Processor instruction/data transfers

Transfer of data between the core and memory involves the address bus, data busses, and control and attribute signals. The address and data buses are parallel, non-multiplexed buses, supporting byte, halfword, three byte, word, and doubleword transfers. All bus input and output signals are sampled and driven with respect to the rising edge of the **m_clk** signal. The core moves data on the bus by issuing control signals and using a handshake protocol to ensure correct data movement.

Separate memory interface ports are provided for instruction (Instruction Port) and data (Data Port) accesses. The instruction memory interface supports aligned read transfers of 32 and 64 bits, supports true big- and little-endian operating modes, and operates in a highly pipelined fashion. The data memory interface supports read and write transfers of 8, 16, 24, 32, and 64 bits, supports misaligned transfers, supports true big- and little-endian operating modes, and operates in a pipelined fashion.

The memory interfaces operate in a pipelined fashion to allow additional access time for memory and peripherals. Accesses that are initiated in a given clock cycle complete two cycles later when running with no wait states.

Read transfers consist of a request cycle, where address and attributes are driven along with a transfer request, a MMU access cycle during which protection checks and address translation occurs, one or more memory access cycles to perform accesses (first cycle in parallel with MMU lookup, subsequent cycles if wait states are involved), and a data return cycle during which the requested information is returned to the CPU for alignment, sign or zero extension, and forwarding, and the access is terminated.

Write transfers consist of a request cycle, where address and attributes are driven along with a transfer request, a MMU access cycle during which protection checks and address translation occurs, a data drive cycle (in parallel with the MMU lookup) where write data is driven and external devices accept write data for the access, and a termination cycle during which termination status is returned. Writes are buffered externally and may be written to memory during unused cycles.

Misaligned data accesses are supported with one or more transfers to the data memory interface. If a data access is misaligned, but is contained within an aligned 64-bit doubleword, the core performs a single transfer, and the memory interface is responsible for delivering (reads) or accepting (writes) the data corresponding to the size signals aligned according to the low order three address bits. If a data access is misaligned and crosses a 64-bit boundary, the e200z759n3 load/store unit will perform a pair of transfers beginning at the effective address, requesting the original data size (either halfword or word) for the first transfer, and for the second transfer the address is incremented to the next 64-bit boundary, and the size signals are driven to indicate the number of remaining bytes to be transferred.

Access requests are generated in an overlapped fashion in order to support sustained single cycle transfers. Up to three access requests may be in progress at any one cycle, two accesses outstanding and a third in the pending request phase. In addition, the core may choose to change the request address and attribute values if a previous request is still pending.

Access requests are assumed to be accepted as long as there are fewer than two accesses in progress (**p_treq_b** asserted with **p_tbusy[1]_b** negated), or if an access in progress is terminated during the same cycle a new request is active (**p_treq_b** asserted with **p_tbusy[1]_b** asserted and one of **p_ta_b** or **p_tea_b** asserted), or if an access is aborted during the same cycle a new request is active (**p_treq_b** asserted with **p_abort_b** asserted). Once an access has been accepted, the core is free to change the current request—the interface control logic needs to capture access information.

The logic equation for *taken* is (~**p_treq_b** & (**p_tbusy[1]_b** | ~**p_ta_b** | ~**p_tea_b**) & ~**p_halt_zlb**).

The core can also abort an accepted access during the cycle following a valid (taken) request, by asserting **p_abort_b** during the clock cycle following a valid **p_treq_b**. In this case, external Cache control logic must terminate the accepted access. In the case of an aborted access, the address bus and all attributes associated with the aborted request are undefined. It is possible and normal for an access that follows the aborted access to be requested and accepted, and the assertion of abort for the prior request must not affect the subsequent access. Note that in this case **p_abort_b** may assert during the same clock cycle that the following access is being requested, and should only abort the access taken in the previous cycle.

The Cache control logic is responsible for proper pipelining and latching of all interface signals to initiate memory accesses.

The **p_tea_b** signal is used to terminate the current bus cycle when a fault is detected. When the core recognizes a bus error condition for an access, the access is terminated, and subsequent accesses may be aborted.

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 cache 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. The DCache memory controller is responsible for properly aborting a following data access in the pipeline when a data load or store is terminated with **p_tea_b**. This is referred to as an “implicit abort”. In addition to implicit aborts, the CPU may abort a requested access in the clock cycle after it is taken using the **p_abort_b** signal. Due to the pipelined nature of the interface, the CPU cannot cause an explicitly signaled abort of an access that is past the MMU lookup cycle; these accesses are aborted by the Cache memory controller in certain circumstances described in Section 15.3.1.8, Error termination and abort operation, and Section 15.3.1.7, Abort operation.

15.3.1.1 Basic read transfer cycles

During a read transfer, the core receives data from memory or a peripheral device. Figure 15-2 illustrates functional timing for basic read transfers. Clock-by-clock descriptions of activity in Figure 15-2 follows:

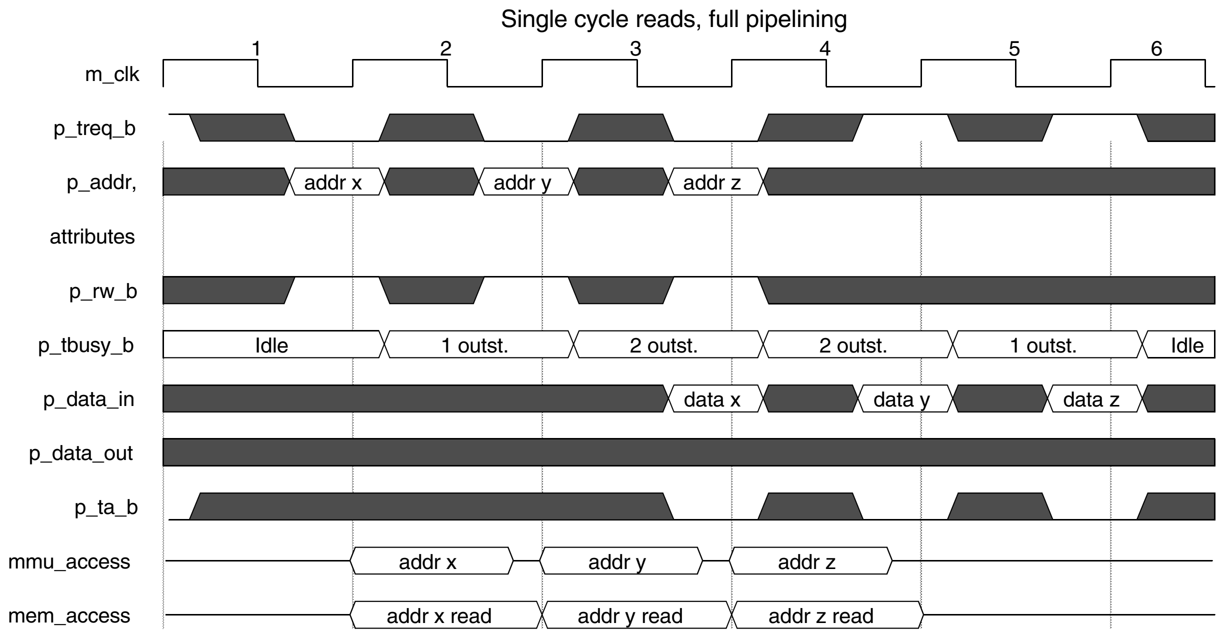


Figure 15-2. Basic read transfers

Clock 1 (C1)

The first read transfer starts in clock cycle 1. During C1, the core places valid values on the address bus and transfer attributes. The transfer code (**p_tc[0:2]**) and transfer type (**p_ttype[0:3]**) attributes identify the specific access type. The transfer size attribute (**p_tsiz[0:2]**) indicates the size of the transfer. The read/write (**p_rw_b**) signal is driven high for a read cycle.

The core asserts transfer request (**p_treq_b**) during C1 to indicate that a transfer is being requested. Since the bus is currently idle, as indicated by the **p_tbusy_b** encoding (0 transfers outstanding), the first read request to addr$_x$ is considered *taken* at the end of C1

Clock 2 (C2):

During C2, the **p_tbusy[0:1]_b** signals are driven to indicate that an access is in progress.

The MMU performs protection checks and address translation in the first part of C2.

The addr$_x$ memory access takes place using the address and attribute values that were driven during C1 to enable reading of one or more bytes of memory.

Another read transfer request is made during C2 to addr$_y$, and since the request pipeline is not full, it is considered *taken* at the end of C2.

Clock 3 (C3):

During C3, the **p_tbusy[0:1]_b** signals are driven to indicate that two accesses are now outstanding (addr$_x$ and addr$_y$).

The MMU performs protection checks and address translation in the first part of C3. Also 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.

During C3, the core samples the level of **p_ta_b**. Since it is asserted, the read cycle for addr$_x$ is completing, and the memory control logic uses the values of **p_tsiz[0:2]**, and **p_addr[29:31]]** that were driven during C1 to place information on the data bus. The memory drives valid data to the core in C3.

During C3, the core asserts **p_treq_b** indicating that another transfer is being requested. The address and attribute signals are driven for a request to addr$_z$.

Clock 4 (C4):

During C4, the **p_tbusy[0:1]_b** signals are driven to indicate that two accesses are now outstanding (addr$_y$ and addr$_z$).

The MMU performs protection checks and address translation in the first part of C4. Also 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.

During C4, the core samples the level of **p_ta_b**. Since it is asserted, the read cycle for addr$_y$ is completing, and the memory control logic uses the values of **p_tsiz[0:2]**, and **p_addr[29:31]** that were driven during C2 to place information on the data bus. The Cache controller drives valid data to the core in C4.

During C4, the core negates **p_treq_b** indicating that no further transfer is being requested. The address and attribute signals are thus undefined.

Clock 5 (C5):

During C5, the **p_tbusy[0:1]_b** signals are driven to indicate that only a single access is now outstanding (addr$_y$ access terminated at the end of C4).

Also during C5, the core samples the level of **p_ta_b**. Since it is asserted, the read cycle for addr$_z$ is completing, and the memory control logic uses the values of **p_tsiz[0:2]**, and **p_addr[29:31]** that were driven during C3 to place information on the data bus. The Cache controller drives valid data to the core in C5.

During C5, the core negates **p_treq_b** indicating that no further transfer is being requested. The address and attribute signals are thus undefined.

Clock 6 (C6):

During C6, the **p_tbusy[0:1]_b** signals are driven to indicate that there are no outstanding transfers (addr$_z$ access terminated at the end of C5).

15.3.1.2 Read transfer with wait states

Figure 15-3 shows an example of wait state operation. Signal **p_ta_b** for the first request (addr$_x$) is not asserted during C3, so wait states are inserted until **p_ta_b** is recognized (during C4).

Meanwhile, subsequent requests have been generated by the CPU for addr$_y$, which is *taken* in C2, since only a single transaction is then outstanding, and for addr$_z$, which is not taken until the end of C4. This request is not considered taken in C3 since there are already two outstanding transfers, and no termination signal is asserted during C3. During C4, **p_ta_b** is asserted to terminate the access to addr$_x$. The request for access to addr$_z$ is taken at the end of C4, and during C5, the MMU lookup and memory access occur.

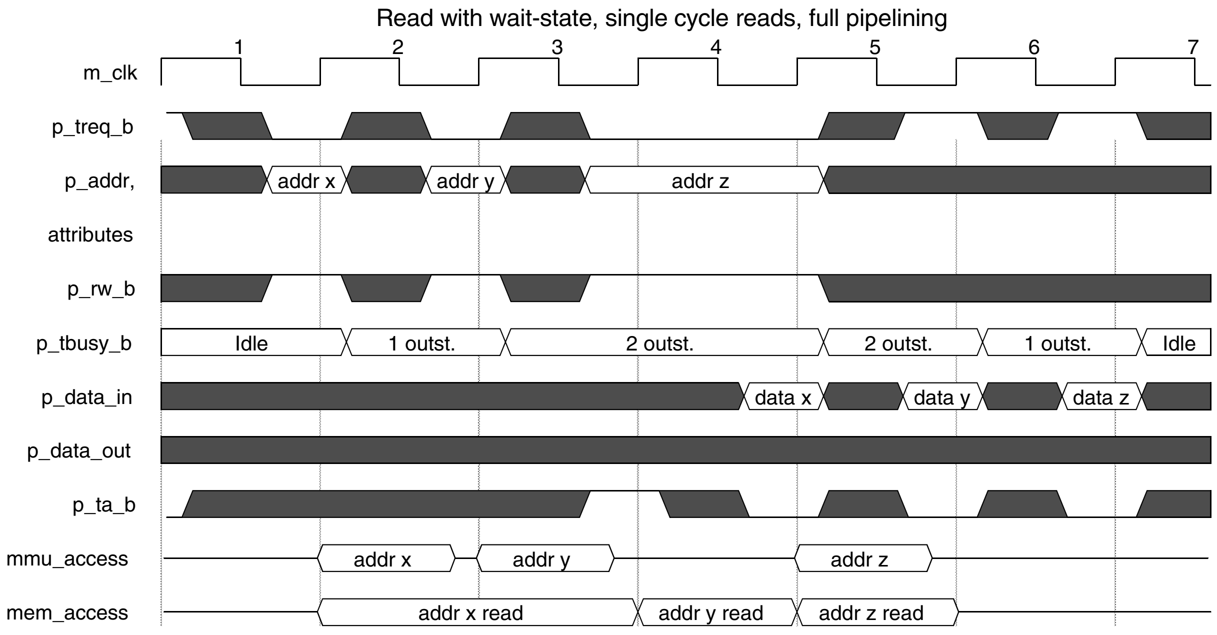


Figure 15-3. Read transfer with wait state

When a requested access is is not taken at the end of a given clock cycle, the CPU is free to negate or change the request on the next cycle. Cache and MMU control logic must be cognizant of this protocol. With two outstanding transfers in progress, a subsequent request is considered accepted only if one of **p_abort_b**, **p_ta_b,** or **p_tea_b** are asserted and **p_treq_b** is also asserted at the end of a clock cycle. Figure 15-4 shows an example of a request change due to a not-taken request. In C3, the request for addr$_z$ is not taken since two requests are outstanding and no termination was asserted. In C4, the request changes to addr$_w$, which is then taken in C4 since termination for the initial request to addr$_x$ is asserted. The addr$_w$ request terminates at the end of C6.

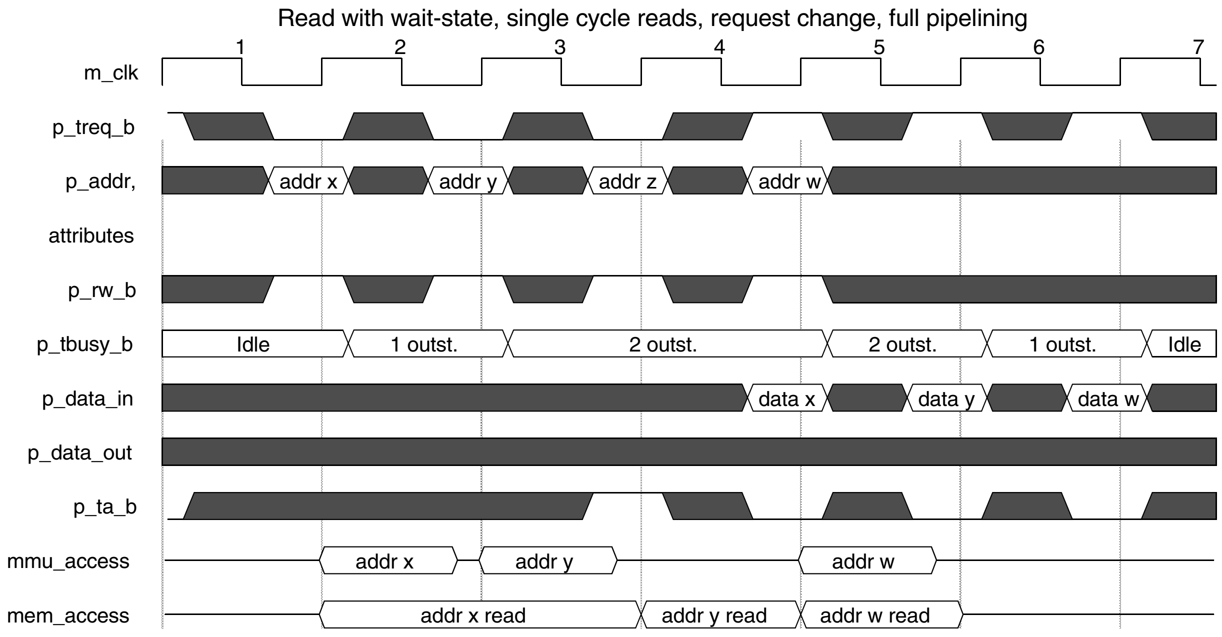


Figure 15-4. Read transfer with wait state, request change

15.3.1.3 Basic write transfer cycles

During a write transfer, the core provides write data to a memory or peripheral device. Figure 15-5 illustrates functional timing for basic write transfers. Clock-by-clock descriptions of activity in Figure 15-5 follows:

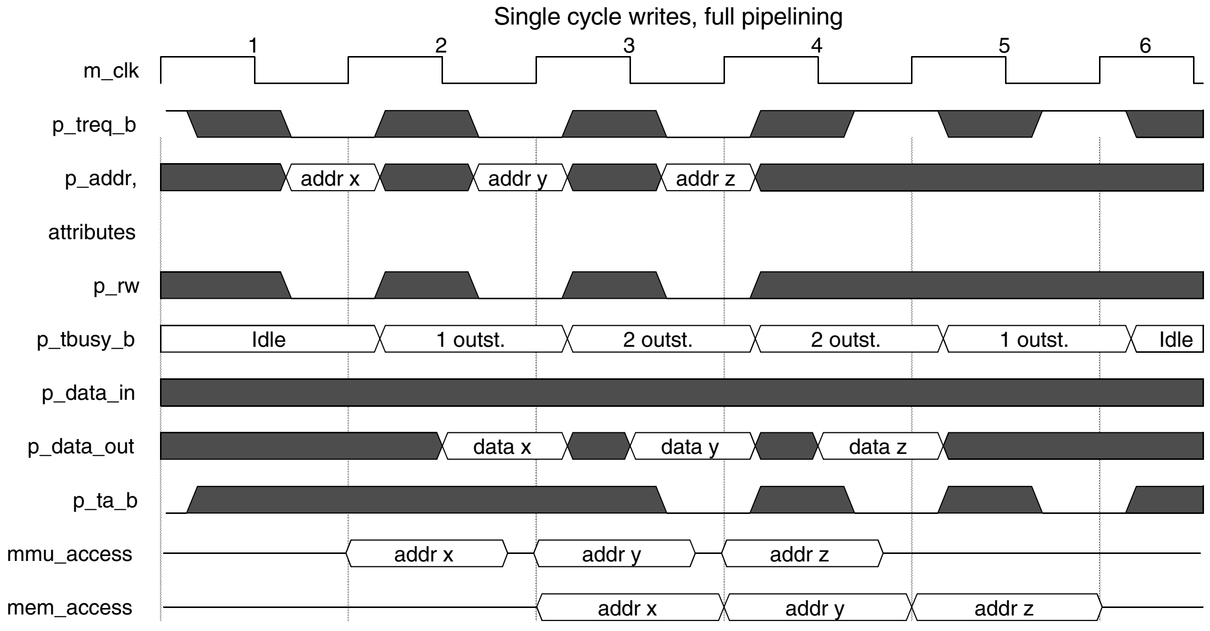


Figure 15-5. Basic write transfers

Clock 1 (C1)

The first write transfer starts in clock cycle 1. During C1, the core places valid values on the address bus and transfer attributes. The transfer code (**p_tc[0:2]**) and transfer type (**p_ttype[0:3]**) attributes identify the specific access type. The transfer size attribute (**p_tsiz[0:2]**) indicates the size of the transfer. The read/write (**p_rw_b**) signal is driven low for a write cycle.

The core asserts transfer request (**p_treq_b**) during C1 to indicate that a transfer is being requested. Since the bus is currently idle, as indicated by the p_tbusy_b encoding (0 transfers outstanding), the first write request to addr_x is considered *taken* at the end of C1

Clock 2 (C2):

During C2, the **p_tbusy[0:1]_b** signals are driven to indicate that an access is in progress.

The MMU performs protection checks and address translation in the first part of C2. Also during C2, data for the addr_x write (data_x) is provided on the **p_data_out** bus. Write data for an access is driven once the access is *taken*.

Unlike reads, the write is not performed during the MMU lookup cycle. It must be buffered for later use, since the protection check outcome may not allow the write. In addition, the write must be implicitly aborted if a preceding read or write data access results in an error.

Another write transfer request is made during C2 to addr_y, and since the request pipeline is not full, it is considered *taken* at the end of C2.

Clock 3 (C3):

During C3, the **p_tbusy[0:1]_b** signals are driven to indicate that two accesses are now outstanding (addr_x and addr_y). Data for the addr_y write (data_y) is provided on the **p_data_out** bus.

Also during C3, the addr_x memory access takes place using the address and attribute values that were driven during C1, and the data that was driven during C2 to enable writing of one or more bytes of memory.

During C3, the core samples the level of **p_ta_b**. Since it is asserted, the write cycle for addr_x is complete.

During C3, the core asserts **p_treq_b** indicating that another transfer is being requested. The address and attribute signals are driven for a request to addr_z.

Clock 4 (C4):

During C4, the **p_tbusy[0:1]_b** signals are driven to indicate that two accesses are now outstanding (addr_y and addr_z).

Also during C4, the addr_y memory access takes place using the address and attribute values that were driven during C2, and the data that was driven during C3 to enable writing of one or more bytes of memory.

During C4, the core samples the level of **p_ta_b**. Since it is asserted, the write cycle for addr_y is complete.

During C4, the core negates **p_treq_b** indicating that no further transfer is being requested. The address and attribute signals are thus undefined.

Clock 5 (C5):

During C5, the **p_tbusy[0:1]_b** signals are driven to indicate that only a single access (addr_z) is now outstanding (addr_y access terminated at the end of C4).

Also during C5, the $addr_z$ memory access takes place using the address and attribute values that were driven during C3, and the data that was driven during C4 to enable writing of one or more bytes of memory.

Also during C5, the core samples the level of **p_ta_b**. Since it is asserted, the write cycle for $addr_z$ is complete.

During C5, the core negates **p_treq_b** indicating that no further transfer is being requested. The address and attribute signals are thus undefined.

Clock 6 (C6):

During C6, the **p_tbusy[0:1]_b** signals are driven to indicate that there are no outstanding transfers ($addr_z$ access terminated at the end of C5).

15.3.1.4 Write transfer with wait states

Figure 15-6 shows an example of wait state operation during write cycles.

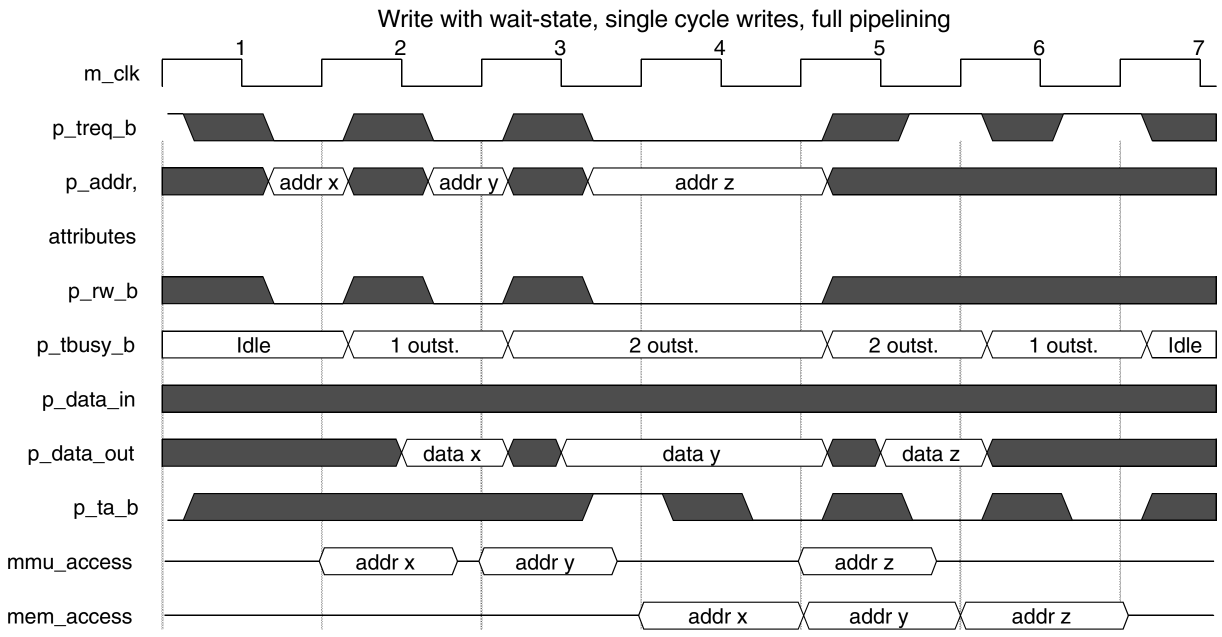


Figure 15-6. Write transfer with wait state

Signal **p_ta_b** for the first request ($addr_x$) is not asserted during C3, so wait states are inserted until **p_ta_b** is recognized (during C4).

Meanwhile, a subsequent request has been generated by the CPU for $addr_y$, which is *taken* in C2 (since only a single transaction is then outstanding), and $data_y$ is driven in C3 once $addr_y$ is *taken*. The MMU performs protection checks and address translation for $addr_y$ in the first part of C3. This information must be stored until the $addr_y$ write is allowed to begin, which is not until the successful outcome of the preceding write to $addr_x$.

Also during C3, a request is generated for a write to $addr_z$, which is not taken until the end of C4. This request is not considered taken in C3 since there are already two outstanding transfers, and no termination signal is asserted during C3. Data for addr$_y$ remains driven until the next access request is *taken*. Data for a write cycle is not driven until the cycle after access is *taken*, thus data$_z$ is not driven until C5.

During C4, the write to addr$_x$ successfully terminates. At the end of C4, the write to addr$_z$ is taken.

In C5, the write to addr$_y$ occurs using the information driven during C2 and C3. Data for addr$_z$ is also driven during C5.

During C5, the write to addr$_y$ successfully terminates. In C6, the write to addr$_z$ occurs using the information driven during C4 and C5.

15.3.1.5 Read and write transfers

Figure 15-7 shows a sequence of read and write cycles.

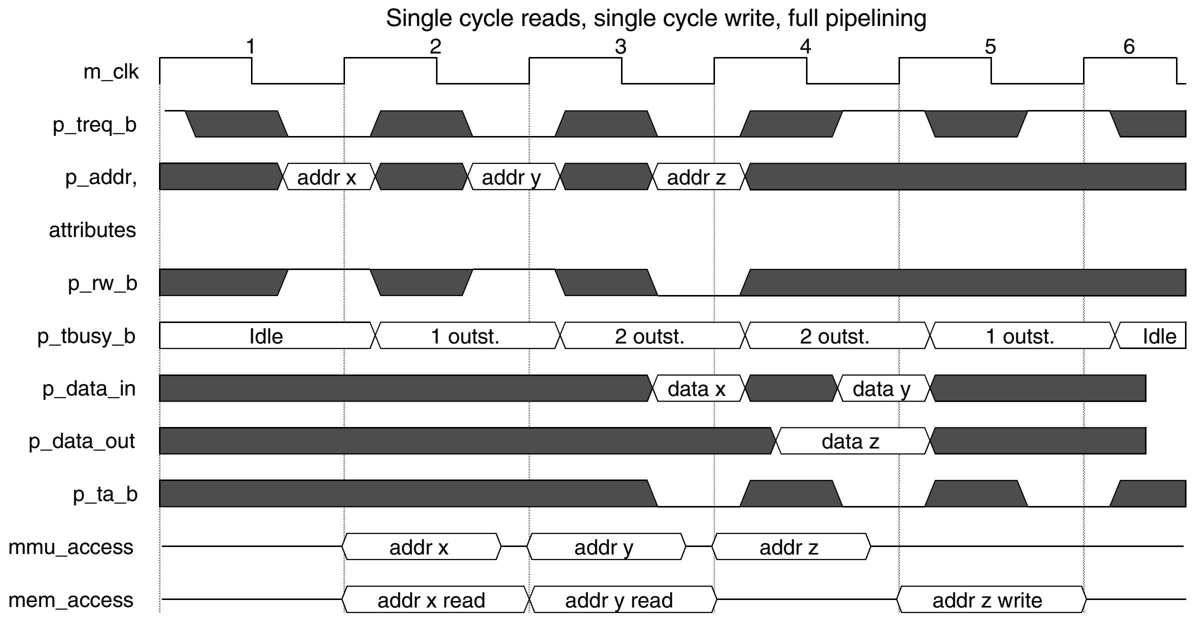


Figure 15-7. Single-cycle read and write transfers — 1

The first read request (addr$_x$) is *taken* at the end of cycle C1 since the bus is idle.

The second read request (addr$_y$) is *taken* at the end of C2 since only one access is outstanding (addr$_x$).

p_ta_b is asserted during C3 for the first read access (addr$_x$). Also during C3, a request is generated for a write to addr$_z$, which is taken at the end of C3 since the first access is terminating.

Data for the addr$_z$ write cycle is driven in C4, the cycle after the access is *taken*. During C4, read data is supplied for the addr$_y$ read, and the access is terminated.

During C5, **p_ta_b** is asserted to complete the write cycle to addr$_z$.

Figure 15-8 shows another sequence of read and write cycles. This example shows the memory controller buffering an interleaved write access between two reads.

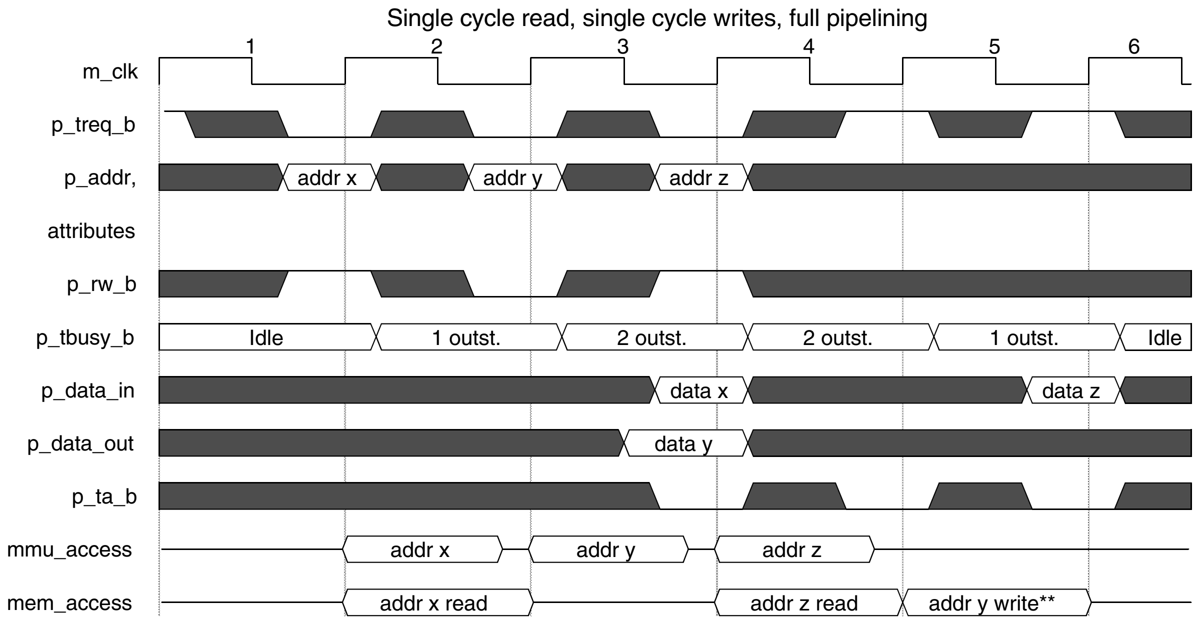


Figure 15-8. Single-cycle read and write transfers — 2

The first read request ($addr_x$) is *taken* at the end of cycle C1 since the bus is idle.

The first write request ($addr_y$) is *taken* at the end of C2 since only one access is outstanding ($addr_x$).

p_ta_b is asserted during C3 for the first read access ($addr_x$). Also during C3, a request is generated for a read to $addr_z$, which is taken at the end of C3 since the first access is terminating.

Data for the $addr_y$ write cycle is driven in C3, the cycle after the access is *taken*. Also during C3, protection checks are made for the $addr_y$ write, and any address translation is performed.

During C4, the $addr_y$ write access is terminated with **p_ta_b** (assumes no MMU fault occurred) after being buffered. Since there is another read access to $addr_z$ occurring during C4, the memory controller delays the $addr_y$ write while checking that no conflict or hazard is occurring with $addr_z$ and performs the read access to $addr_z$.

During C5, **p_ta_b** is asserted to complete the read cycle to $addr_z$. Since no read cycle needs to occur to memory during C5, the buffered write access for $addr_y$ is performed to memory. Optionally, the memory controller may continue to hold the write in the buffer until a later time.

Once a write has been buffered, **p_tea_b** may not be subsequently asserted for that write cycle. Also, once it has been buffered, a transfer error on an access that logically follows the write must not affect the write from being performed.

Figure 15-9 shows another sequence of read and write cycles. In this example, reads incur a single wait state.

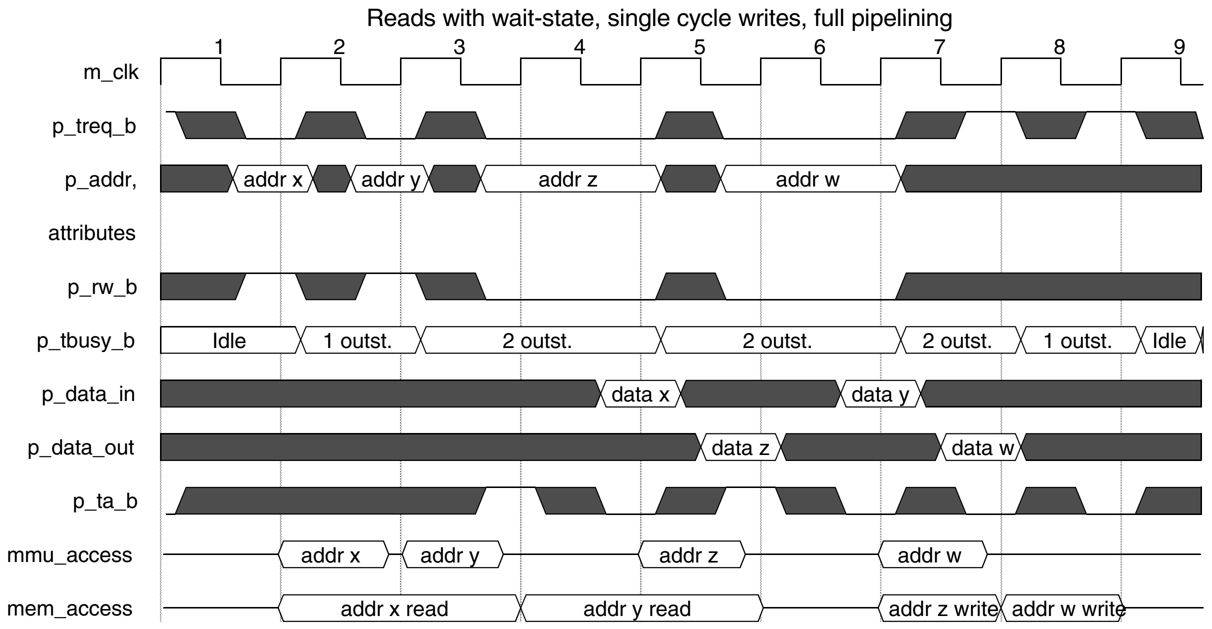


Figure 15-9. Multi-cycle read and write transfers — 1

The first read request (addr$_x$) is *taken* at the end of cycle C1 since the bus is idle.

The second read request (addr$_y$) is *taken* at the end of cycle C2 since only one access is outstanding (addr$_x$).

The first write request (addr$_z$) is not taken during C3 since two accesses are outstanding in C3 (addr$_x$,addr$_y$).

p_ta_b is asserted during C4 for the first read access (addr$_x$). Also during C4, the request for a write to addr$_z$ is taken since the first access is terminating.

Data for the addr$_z$ write cycle is driven in C5, the cycle after the access is *taken*. The MMU lookup occurs in C5 for the write to addr$_z$. A second write request (addr$_w$) is not taken during C5 since two accesses are still outstanding (addr$_y$,addr$_z$).

During C6, the addr$_y$ read access is terminated and the addr$_w$ write request is *taken*.

During C7, data for the addr$_w$ write access is driven. The MMU lookup occurs for the write to addr$_w$. In C7, the write to addr$_z$ may be performed, since it is determined that the preceding access terminated successfully. **p_ta_b** is asserted to complete the write cycle to addr$_z$.

During C8, **p_ta_b** is asserted to complete the write cycle to addr$_w$.

Figure 15-10 shows another sequence of read and write cycles. In this example, reads incur a single wait state.

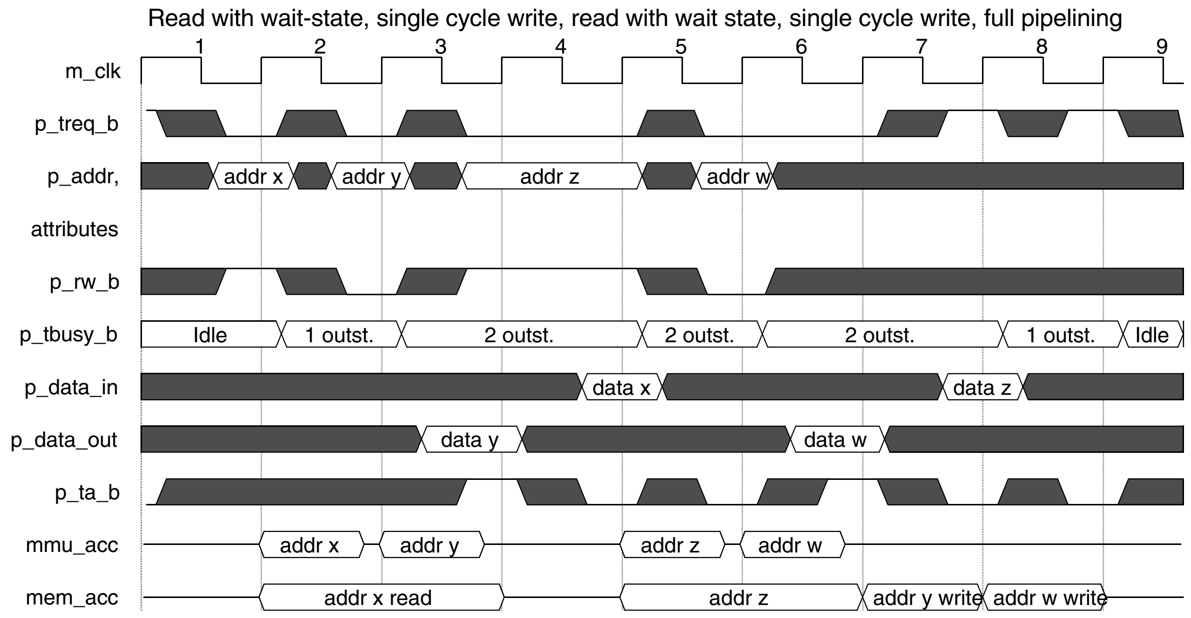


Figure 15-10. Multi-cycle read and write transfers — 2

The first read request (addr$_x$) is *taken* at the end of cycle C1 since the bus is idle.

The first write request (addr$_y$) is *taken* at the end of cycle C2 since only one access is outstanding (addr$_x$).

The second read request (addr$_z$) is not taken during C3 since two accesses are outstanding in C3 (addr$_x$,addr$_y$). Data for the addr$_y$ write cycle is driven in C3, the cycle after the access is *taken*. The MMU lookup (if present) occurs in C3 for the write to addr$_y$.

p_ta_b is asserted during C4 for the first read access (addr$_x$). Also during C4, the request for a read to addr$_z$ is taken since the first access is terminating.

The MMU lookup occurs in C5 for the read to addr$_z$. Also during C5, **p_ta_b** is asserted to terminate the write cycle to addr$_y$, which has been buffered. This write does not occur during C5 since a read request is being serviced (assumes no hazard has been detected). A second write request (addr$_w$) is taken at the end of C5 since the second access is terminating.

During C6, data for the addr$_w$ write access is driven. The MMU lookup occurs for the write to addr$_w$.

During C7, **p_ta_b** is asserted to complete the read cycle to addr$_z$. The buffered write for addr$_y$ is now performed since no read is occurring.

During C8,the write to addr$_w$ is performed, and **p_ta_b** is asserted to terminate the write cycle to addr$_w$.

15.3.1.6 Misaligned accesses

Figure 15-11 illustrates functional timing for a misaligned read transfer. The read to addr$_x$ is misaligned across a 64-bit boundary.

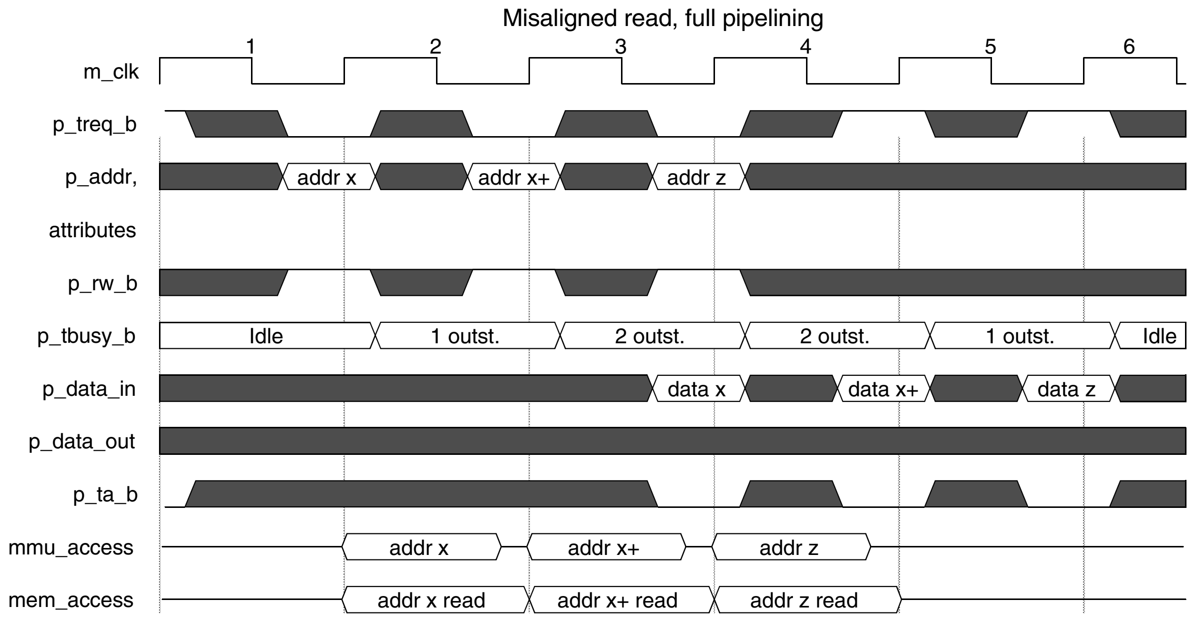


Figure 15-11. Misaligned read transfer

The first portion of the misaligned read transfer starts in C1. During C1, the core places valid values on the address bus and transfer attributes. The read/write (**p_rw_b**) signal is driven high for a read cycle. The transfer code (**p_tc[0:2]**) and transfer type (**p_ttype[0:3]**) attributes identify the specific access type. The transfer size attribute (**p_tsiz[0:2]**) indicates the size of the transfer. Even though the transfer is misaligned, the size value driven corresponds to the size of the entire misaligned data item.

The core asserts transfer request (**p_treq_b**) during C1 to indicate that a transfer is being requested. Since the bus is currently idle, as indicated by the **p_tbusy_b** encoding (0 transfers outstanding), the first read request to $addr_x$ is considered *taken* at the end of C1

During C2, the **p_tbusy[0:1]_b** signals are driven to indicate that an access is in progress.

The MMU performs protection checks and address translation in the first part of C2.

The $addr_x$ memory access takes place using the address and attribute values that were driven during C1 to enable reading of one or more bytes of memory.

The second portion of the misaligned read transfer request is made during C2 to $addr_{x+}$, and since the request pipeline is not full, it is considered *taken* at the end of C2. The size value driven is the size of the remaining bytes of data in the misaligned read.

During C3, the **p_tbusy[0:1]_b** signals are driven to indicate that two accesses are now outstanding ($addr_x$ and $addr_{x+}$).

The MMU performs protection checks and address translation in the first part of C3. Also during C3, the $addr_{x+}$ 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.

During C3, the core samples the level of **p_ta_b**. Since it is asserted, the read cycle for addr$_x$ is completing, and the memory control logic uses the values of **p_tsiz[0:2]**, and **p_addr[29:31]** that were driven during C1 to place information on the data bus. The memory drives valid data to the core in C3.

During C3, the core asserts **p_treq_b** indicating that another transfer is being requested. The address and attribute signals are driven for a request to addr$_z$.

During C4, the **p_tbusy[0:1]_b** signals are driven to indicate that two accesses are now outstanding (addr$_{x+}$ and addr$_z$).

The MMU performs protection checks and address translation in the first part of C4. Also 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.

During C4, the core samples the level of **p_ta_b**. Since it is asserted, the read cycle for addr$_{x+}$ is completing, and the memory control logic uses the values of **p_tsiz[0:2]**, and **p_addr[29:31]** that were driven during C2 to place information on the data bus. The memory drives valid data to the core in C4.

During C4, the core negates **p_treq_b** indicating that no further transfer is being requested. The address and attribute signals are thus undefined.

During C5, the **p_tbusy[0:1]_b** signals are driven to indicate that only a single access is now outstanding (addr$_{x+}$ access terminated at the end of C4).

Also during C5, the core samples the level of **p_ta_b**. Since it is asserted, the read cycle for addr$_z$ is completing, and the memory control logic uses the values of **p_tsiz[0:2]**, and **p_addr[29:31]** that were driven during C3 to place information on the data bus. The memory drives valid data to the core in C5.

During C5, the core negates **p_treq_b** indicating that no further transfer is being requested. The address and attribute signals are thus undefined.

During C6, the **p_tbusy[0:1]_b** signals are driven to indicate that there are no outstanding transfers (addr$_z$ access terminated at the end of C5).

Figure 15-12 illustrates functional timing for a misaligned write transfer. The write to addr$_x$ is misaligned across a 64-bit boundary.

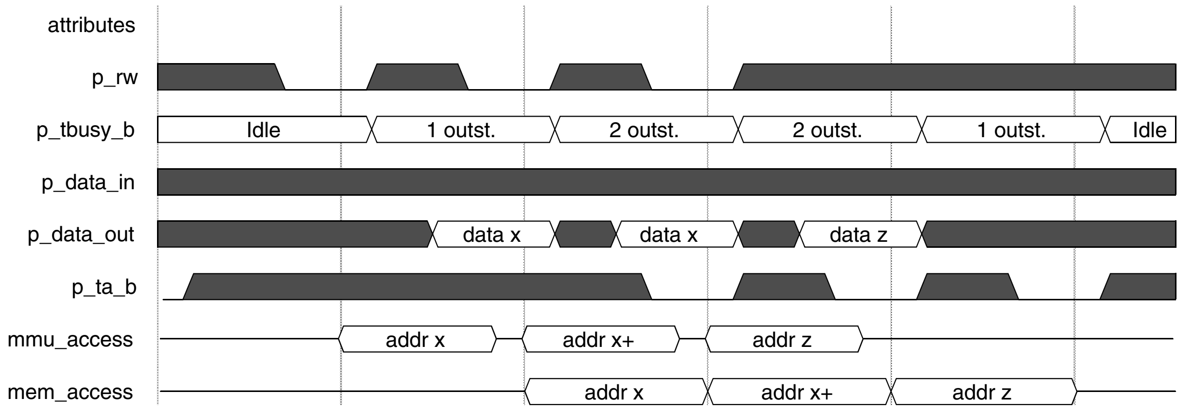


Figure 15-12. Misaligned write transfer

The first portion of the misaligned write transfer starts in C1. During C1, the core places valid values on the address bus and transfer attributes. The transfer code (**p_tc[0:2]**) and transfer type (**p_ttype[0:3]**) attributes identify the specific access type. The transfer size attribute (**p_tsiz[0:2]**) indicates the size of the transfer. Note that for the misaligned transfer, the size is driven to indicate the size of the entire misaligned data item, not just the portion driven in the first cycle. The read/write (**p_rw_b**) signal is driven low for a write cycle.

The core asserts transfer request (**p_treq_b**) during C1 to indicate that a transfer is being requested. Since the bus is currently idle, as indicated by the p_tbusy_b encoding (0 transfers outstanding), the first write request to addr_x is considered *taken* at the end of C1

During C2, the **p_tbusy[0:1]_b** signals are driven to indicate that an access is in progress.

If present, the MMU performs protection checks and address translation in the first part of C2. Also during C2, data for the addr_x write (data_x) is provided on the **p_data_out** bus. Write data for an access is driven once the access is *taken*.

Unlike reads, the write is not performed during the MMU lookup cycle. It must be buffered for later use, since the protection check outcome may not allow the write. In addition, the write must be implicitly aborted if a preceding read or write data access results in an error.

The second half of the misaligned write transfer request is made during C2 to addr_{x+}, and since the request pipeline is not full, it is considered *taken* at the end of C2.

During C3, the **p_tbusy[0:1]_b** signals are driven to indicate that two accesses are now outstanding (addr_x and addr_{x+}). Data for the addr_{x+} write (data_x) is provided on the **p_data_out** bus. The value driven remains unchanged from the previous cycle.

Also during C3, the addr_x memory access takes place using the address and attribute values that were driven during C1, and the data that was driven during C2 to enable writing of one or more bytes of memory.

During C3, the core samples the level of **p_ta_b**. Since it is asserted, the write cycle for addr_x is complete.

During C3, the core asserts **p_treq_b** indicating that another transfer is being requested. The address and attribute signals are driven for a request to addr_z.

During C4, the **p_tbusy[0:1]_b** signals are driven to indicate that two accesses are now outstanding (addr_{x+} and addr_z).

Also during C4, the addr_{x+} memory access takes place using the address and attribute values that were driven during C2, and the data that was driven during C3 to enable writing of one or more bytes of memory.

During C4, the core samples the level of **p_ta_b**. Since it is asserted, the write cycle for addr_{x+} is complete.

During C4, the core negates **p_treq_b** indicating that no further transfer is being requested. The address and attribute signals are thus undefined.

During C5, the **p_tbusy[0:1]_b** signals are driven to indicate that only a single access (addr_z) is now outstanding (addr_{x+} access terminated at the end of C4).

Also during C5, the addr_z memory access takes place using the address and attribute values that were driven during C3, and the data that was driven during C4 to enable writing of one or more bytes of memory.

Also during C5, the core samples the level of **p_ta_b**. Since it is asserted, the write cycle for addr$_z$ is complete.

During C5, the core negates **p_treq_b** indicating that no further transfer is being requested. The address and attribute signals are thus undefined.

During C6, the **p_tbusy[0:1]_b** signals are driven to indicate that there are no outstanding transfers (addr$_z$ access terminated at the end of C5).

An example of a misaligned write cycle followed by an aligned read cycle is shown in Figure 15-13.

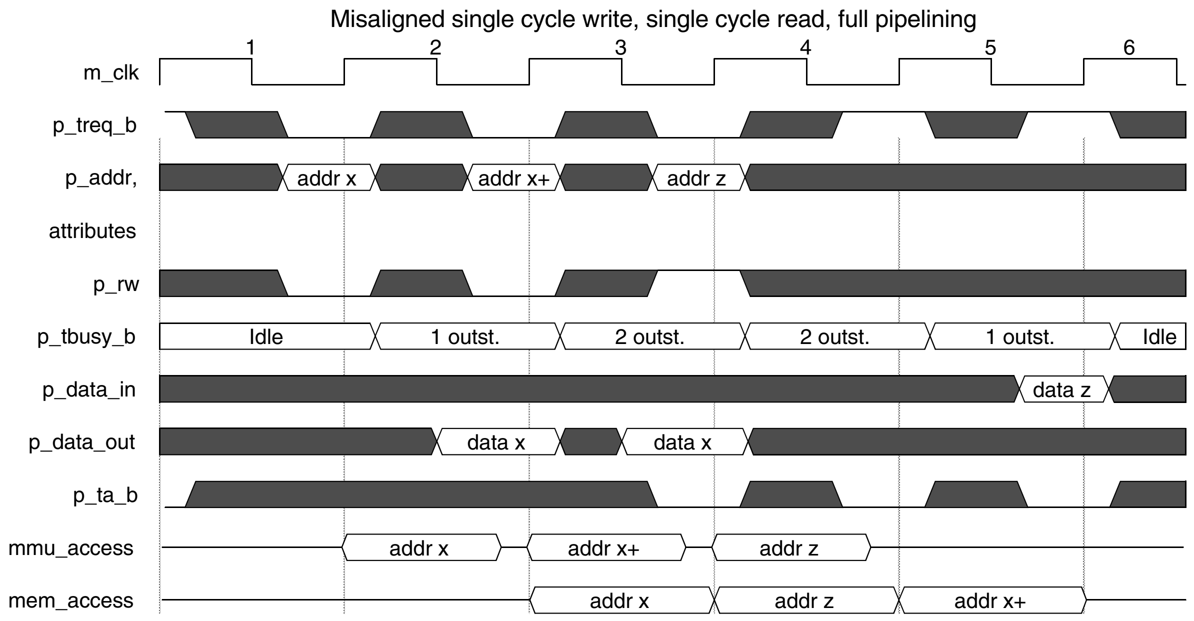


Figure 15-13. Misaligned write, single-cycle read transfer

The first portion of the misaligned write request (addr$_x$) is *taken* at the end of cycle C1 since the bus is idle.

The second portion of the misaligned write request (addr$_{x+}$) is *taken* at the end of cycle C2 since only one access is outstanding (addr$_x$).

Data for the addr$_x$ write cycle is driven in C2, the cycle after the access is *taken*. The MMU lookup occurs in C2 for the write to addr$_x$.

During C3, the write to addr$_x$ occurs using the information driven during C1 and C2. **p_ta_b** is asserted during C3 for the first write access (addr$_x$).

Data for the addr$_{x+}$ write cycle is driven in C3, the cycle after the access is *taken*. The MMU lookup occurs in C3 for the write to addr$_{x+}$. The data value driven remains unchanged from the previous access.

Also during C3, the request for a read to addr$_z$ is taken since the first access is terminating.

The MMU lookup and a memory access occurs in C4 for the read to addr$_z$. Also during C4, **p_ta_b** is asserted to terminate the write cycle to addr$_{x+}$, which has been buffered. This write does not occur during C4 since a read request is being serviced (assumes no hazard has been detected).

During C5, **p_ta_b** is asserted to complete the read cycle to $addr_z$. The buffered write for $addr_{x+}$ is now performed since no read is occurring.

15.3.1.7 Abort operation

Under certain circumstances, the CPU may abort an access in the clock cycle following a valid (*taken*) **p_treq_b** in the previous clock. In this event, the previous *taken* access address is an invalid one and must not be used to access devices. Circumstances that may cause aborted accesses include:

- When an exception is detected on a taken request
- When **p_tea_b** occurs on a previous data access
- Internal exception conditions requiring an access to be aborted

Aborted accesses are indicated by the assertion of the **p_abort_b** output early in the clock cycle following a taken access.

For certain external interfaces, this may be too late to cancel a pending access. An indication that an abort will occur on a pending request if the current access is terminated with error is provided via the **p_err_kill** output signal. This signal allows the BIU to interface to an external bus that does not provide an abort function directly, but instead uses a two-cycle error response protocol, such as AMBA AHB. The **p_err_kill** output signal is asserted along with **p_treq_b** for every access which, if terminated with error, will cause a next pending access request to be aborted.

For AMBA AHB, if an access that was requested having **p_err_kill** asserted receives HRESP=ERROR, this will cause assertion of **p_tea_b** in the first cycle of the two-cycle error response protocol, and changes the next AHB cycle to IDLE. If an access request is pending, or becomes pending in the cycle **p_tea_b** is first generated, **p_abort_b** will assert in the following clock cycle, and will allow **p_tea_b** to be generated for the second cycle of the two-cycle error response protocol. In either case, the second cycle of the two-cycle error response protocol will be an AHB IDLE cycle. If no access request is pending, and none becomes pending in the cycle **p_tea_b** is first generated, **p_abort_b** will not assert in the following clock cycle, and **p_tea_b** will not be generated for the second cycle of the two-cycle error response protocol.

The first cycle of the two-cycle error response protocol does not result in a **p_tea_b** if **p_err_kill** was negated when the access was requested. In this case, **p_tea_b** is not generated until the second cycle of the two-cycle error response, and any pending access will be allowed to occur normally on the AHB.

Figure 15-14 is an example of **p_abort_b** operation. The access for $addr_z$is requested and taken, then aborted. No termination response is expected for an aborted access. Note that the following access to $addr_w$ is not affected by the abort of $addr_z$.

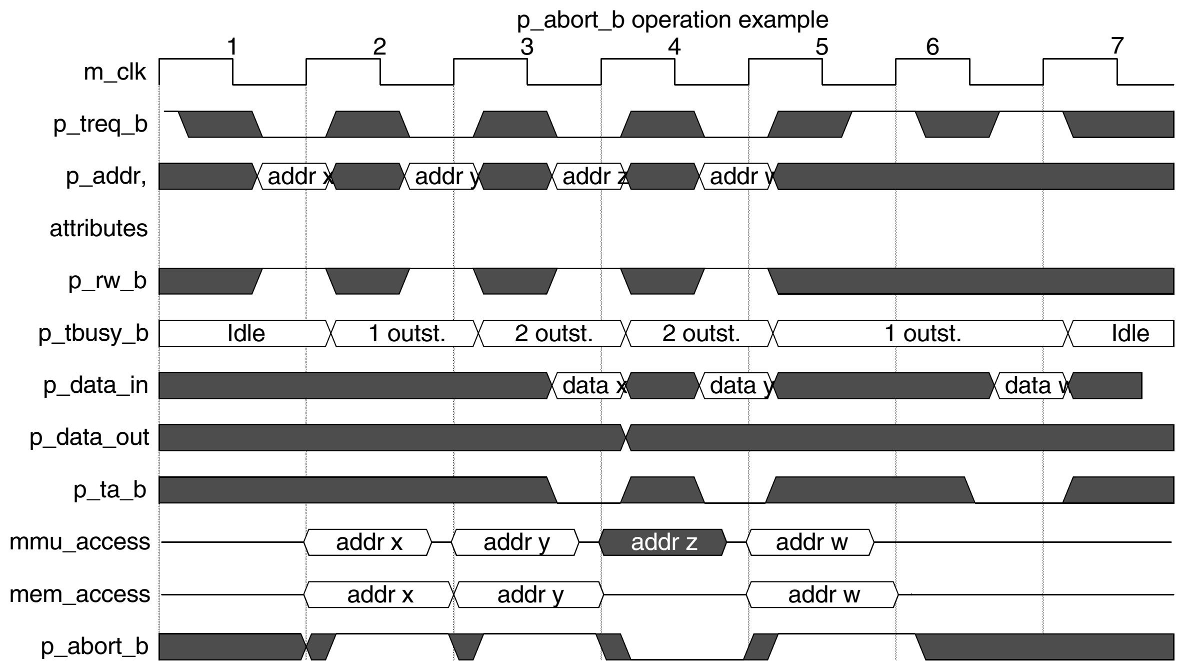


Figure 15-14. p_abort_b operation

15.3.1.8 Error termination and abort operation

The **p_tea_b** input is used to signal an error termination for an access in progress. **p_tea_b** has priority over the **p_ta_b** input.

Figure 15-15 shows an example of error termination with the **p_tea_b** input signal.

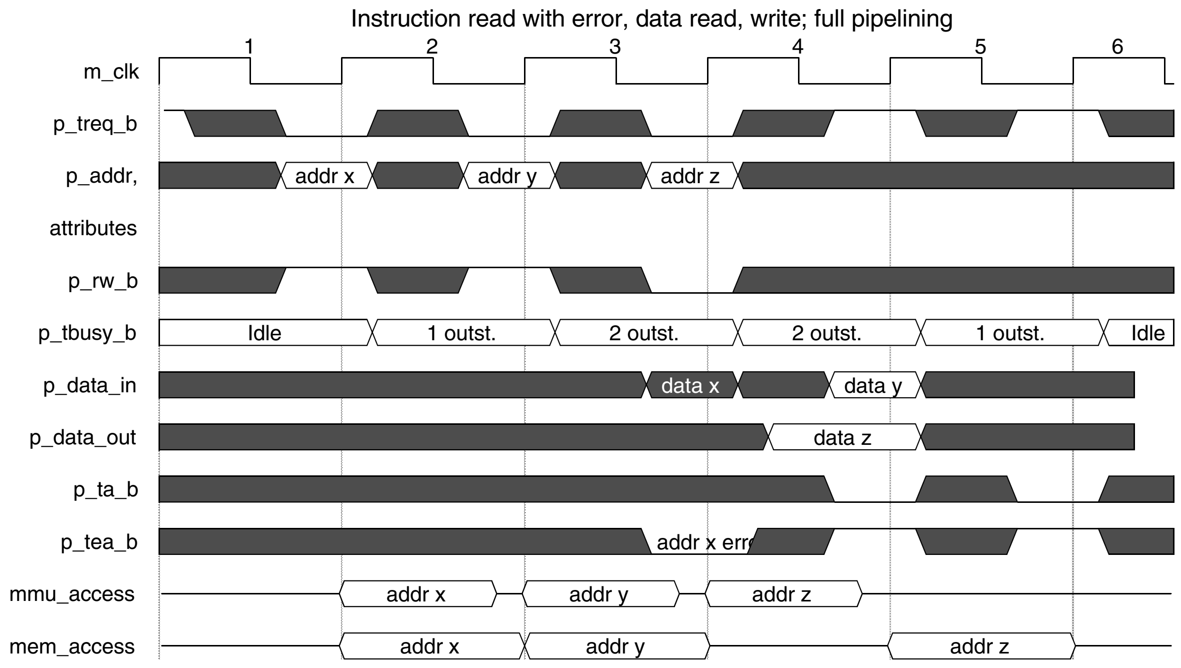


Figure 15-15. Read and write transfers, instruction read error termination

The first read request (addr$_x$) is *taken* at the end of cycle C1 since the bus is idle. It is an instruction prefetch.

The second read request (addr$_y$) is *taken* at the end of C2 since only one access is outstanding (addr$_x$).

p_tea_b is asserted during C3 for the first read access (addr$_x$), signaling an error condition. Since **p_tea_b** terminates the access, the data input bus is undefined. Also during C3, a request is generated for a write to addr$_z$, which is taken at the end of C3 since the first access is terminating.

Data for the addr$_z$ write cycle is driven in C4, the cycle after the access is *taken*. During C4, read data is supplied for the addr$_y$ read, and the access is terminated.

During C5, **p_ta_b** is asserted to complete the write cycle to addr$_z$.

In this example of error termination, subsequent accesses must be allowed to complete. This is not always allowable, and the memory system is responsible for preventing accesses from occurring when certain types of transfers are terminated with error. The following figures outline cases where an error termination for a given cycle must cause another cycle to be aborted by the memory controller prior to initiation.

Figure 15-16 shows another example of error termination with the **p_tea_b** input signal.

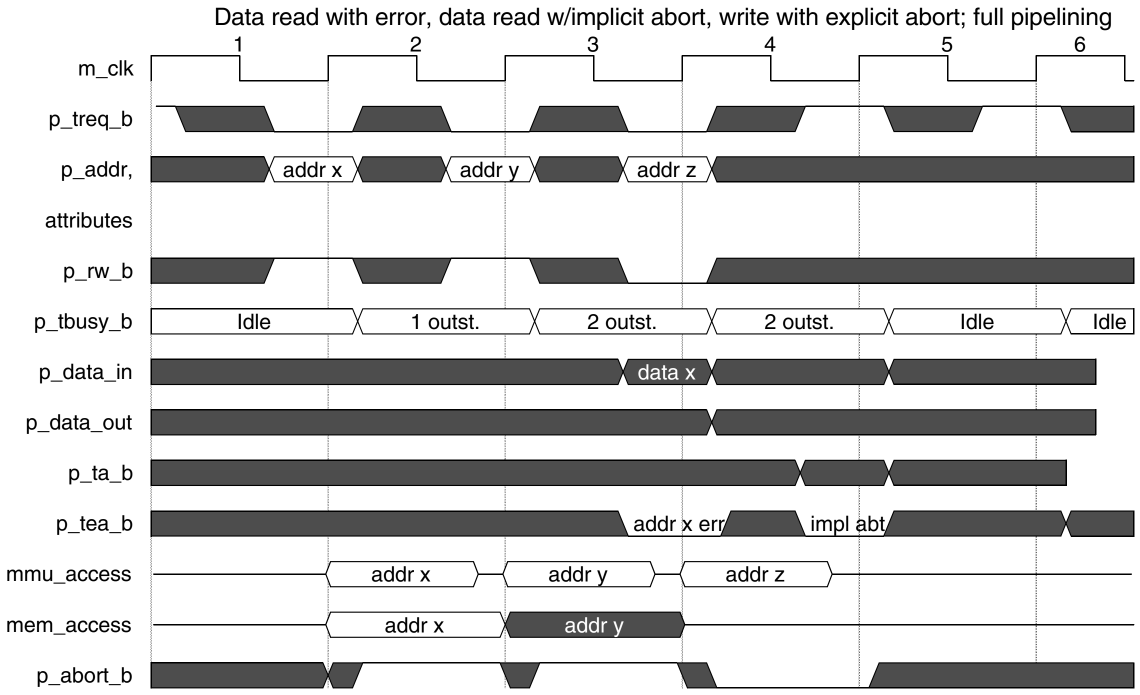


Figure 15-16. Data read error termination with implicit abort

The first read request (addr$_x$) is *taken* at the end of cycle C1 since the bus is idle. It is a data read.

The second read request (addr$_y$) is *taken* at the end of C2 since only one access is outstanding (addr$_x$).

p_tea_b is asserted during C3 for the first read access (addr$_x$), signaling an error condition. Since **p_tea_b** terminates the access, the data input bus is undefined. Also during C3, a request is generated for a write to addr$_z$, which is taken at the end of C3 since the first access is terminating.

In this example the read to addr$_x$ is a data access, therefore the memory controller is responsible for an implicit abort of another data access that is in the pipeline, in this case the data read to addr$_y$. If the access to addr$_y$ had been a write, it would be required to implicitly abort it without the write taking place. As seen in C4, implicit aborts are terminated with assertion of **p_tea_b**.

Data for the addr$_z$ write cycle is driven in C4, the cycle after the access is *taken*. During C4, read data is supplied for the addr$_y$ read, and the access is terminated.

During C5, **p_abort_b** is asserted by the CPU to abort the write cycle to addr$_z$. Since the access is aborted, no response is expected from the memory controller.

In this example of error termination, a subsequent access must not be allowed to complete. The memory system is responsible for preventing a subsequent pipelined access from occurring when data transfers are terminated with error.

Figure 15-17 shows another example of error termination with the **p_tea_b** input signal, this time on the initial portion of a misaligned write.

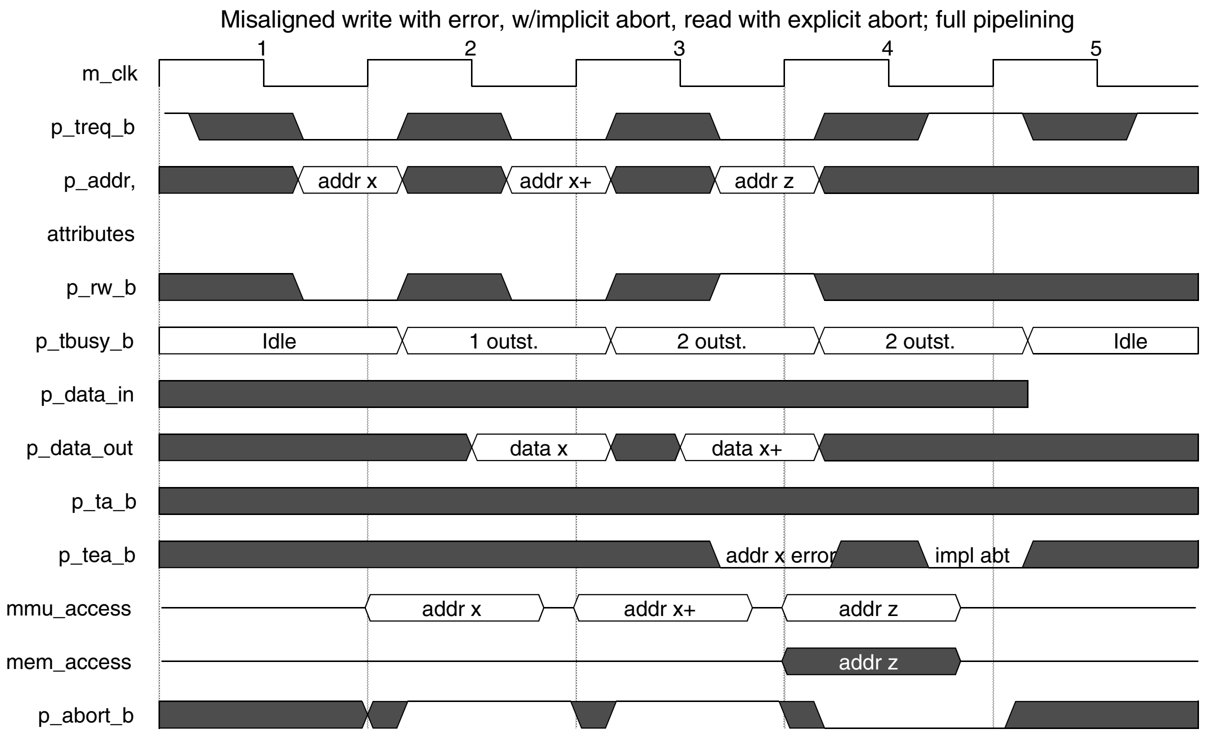


Figure 15-17. Misaligned write error termination with implicit abort

The first portion of the misaligned write request (addr$_x$) is *taken* at the end of cycle C1 since the bus is idle.

The second portion of the misaligned write (addr$_{x+}$) is *taken* at the end of C2 since only one access is outstanding (addr$_x$).

p_tea_b is asserted during C3 for the write read access (addr$_x$), signaling an error condition. Since **p_tea_b** terminates the access, the memory access must not occur. Also during C3, a request is generated for a read to addr$_z$, which is taken at the end of C3 since the first access is terminating.

In this example the write to addr$_x$ is a data access, therefore the memory controller is responsible for an implicit abort of another data access that is in the pipeline, in this case the data write to addr$_{x+}$. Since it is a write, the memory controller is required to implicitly abort it without the write taking place. As seen in C4, this implicit abort is terminated with assertion of **p_tea_b** for addr$_{x+}$, the second half of the misaligned write.

During C4, **p_abort_b** is asserted by the CPU to abort the read cycle to addr$_z$. Since the access is aborted, no response is expected from the memory controller. The memory access is aborted in C4, and does not proceed.

In this example of error termination, a subsequent access must not be allowed to complete. The memory system is responsible for preventing a subsequent pipelined access from occurring when data transfers are terminated with error.

15.3.2 SPR interface operation

An interface protocol is defined for Special Purpose Registers that exist external to the base e200z759n3 core, such as SPRs for the Cache, MMU, and SPE/EFPU. The protocol is a simple handshake, with fixed response timing. Accesses to an external SPR as a result of execution of a **mfspr** or **mtspr** instruction are initiated with assertion of either the **p_wr_spr** or **p_rd_spr** outputs. These outputs are mutually exclusive. Data for a **mtspr** is placed on the **p_spr_out[0:31]** bus the cycle following assertion of **p_wr_spr**. Read data for a **mfspr** instruction is sampled from **p_spr_in[0:31]** the cycle following assertion of **p_rd_spr**.

The following timing diagrams indicate operation of **mfspr** and **mtspr** instructions across the interface. Additional detail on mtspr/mfspr instruction timing may be found in Section 4.3.8, Move to/from SPR instruction pipeline operation.

Figure 15-18 shows an example of SPR read operations.

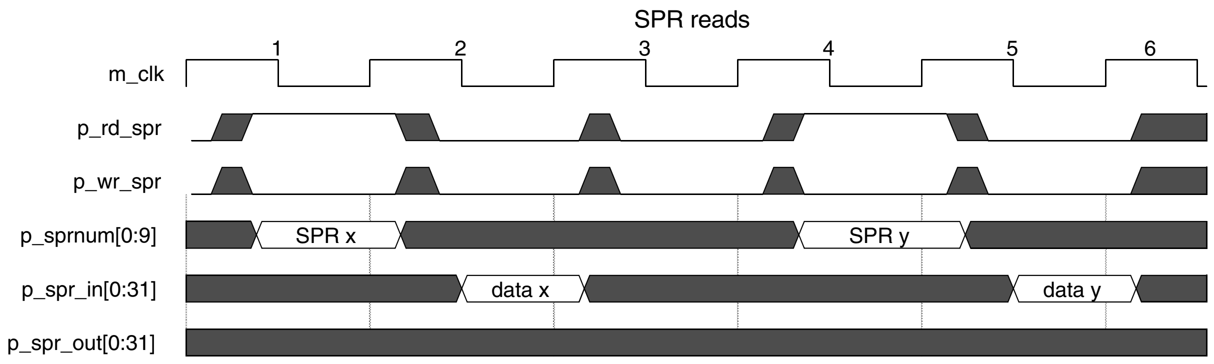


Figure 15-18. SPR reads

In this example, a read of an external SPR is initiated in cycle C1 with assertion of the **p_rd_spr** output. The SPR identifier is provided on **p_sprnum[0:9]**, and is an access to SPR$_x$. In cycle C2, read data for SPR$_x$ is provided on **p_spr_in[0:31]**.This access terminates in C3. A second SPR read is initiated in C4 to SPR$_y$, and data is provided on **p_spr_in[0:31]** in C5. This access terminates in C6.

Figure 15-19 shows an example of SPR write operations.

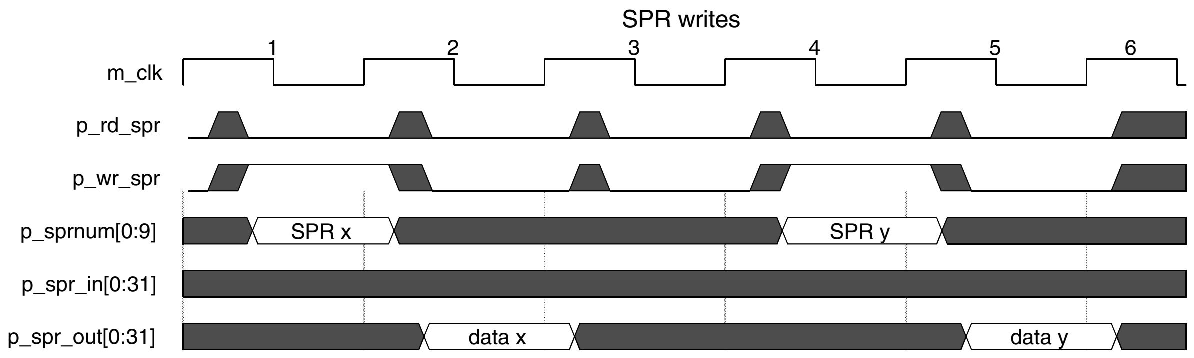


Figure 15-19. SPR writes

In this example, a write of an external SPR is initiated in cycle C1 with assertion of the **p_wr_spr** output. The SPR identifier is provided on **p_sprnum[0:9]**, and is an access to SPR$_x$. In cycle C2, write data for SPR$_x$ is provided on **p_spr_out[0:31]**. This access terminates in C3. A second SPR write is initiated in C4 to SPR$_y$. Data for the SPR$_y$ access is provided on **p_spr_out[0:31]** in C5. This access terminates in C6.

Figure 15-20 shows an example of SPR read and write operations.

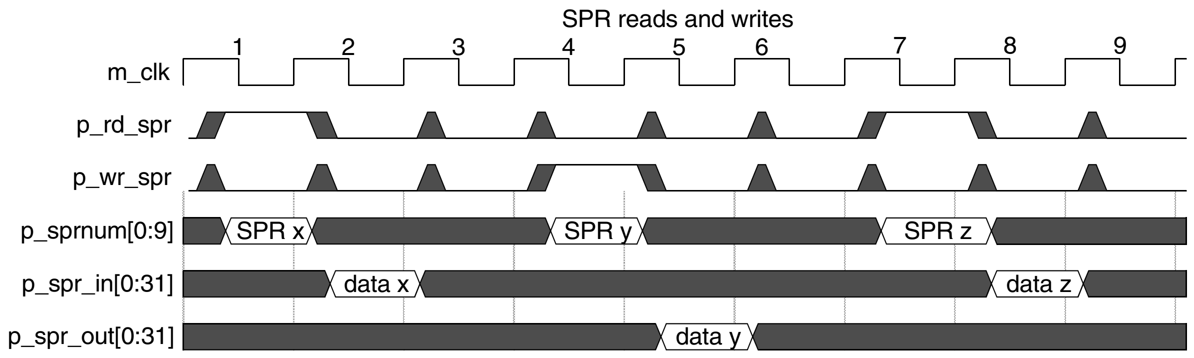


Figure 15-20. SPR reads/writes

In this example, a read of an external SPR is initiated in cycle C1 with assertion of the **p_rd_spr** output. The SPR identifier is provided on **p_sprnum[0:9]**, and is an access to SPR$_x$. In cycle C2, read data for SPR$_x$ is provided on **p_spr_in[0:31]**, and the access terminates. An SPR write is initiated in C4 to SPR$_y$ with assertion of **p_wr_spr**. Data is driven in C5 for the SPR$_y$ write access, and this access terminates in C6. In C7, another SPR read is initiated to SPR$_z$ and data is provided on **p_spr_in[0:31]** for this access in C8.

Appendix A
Register Summary

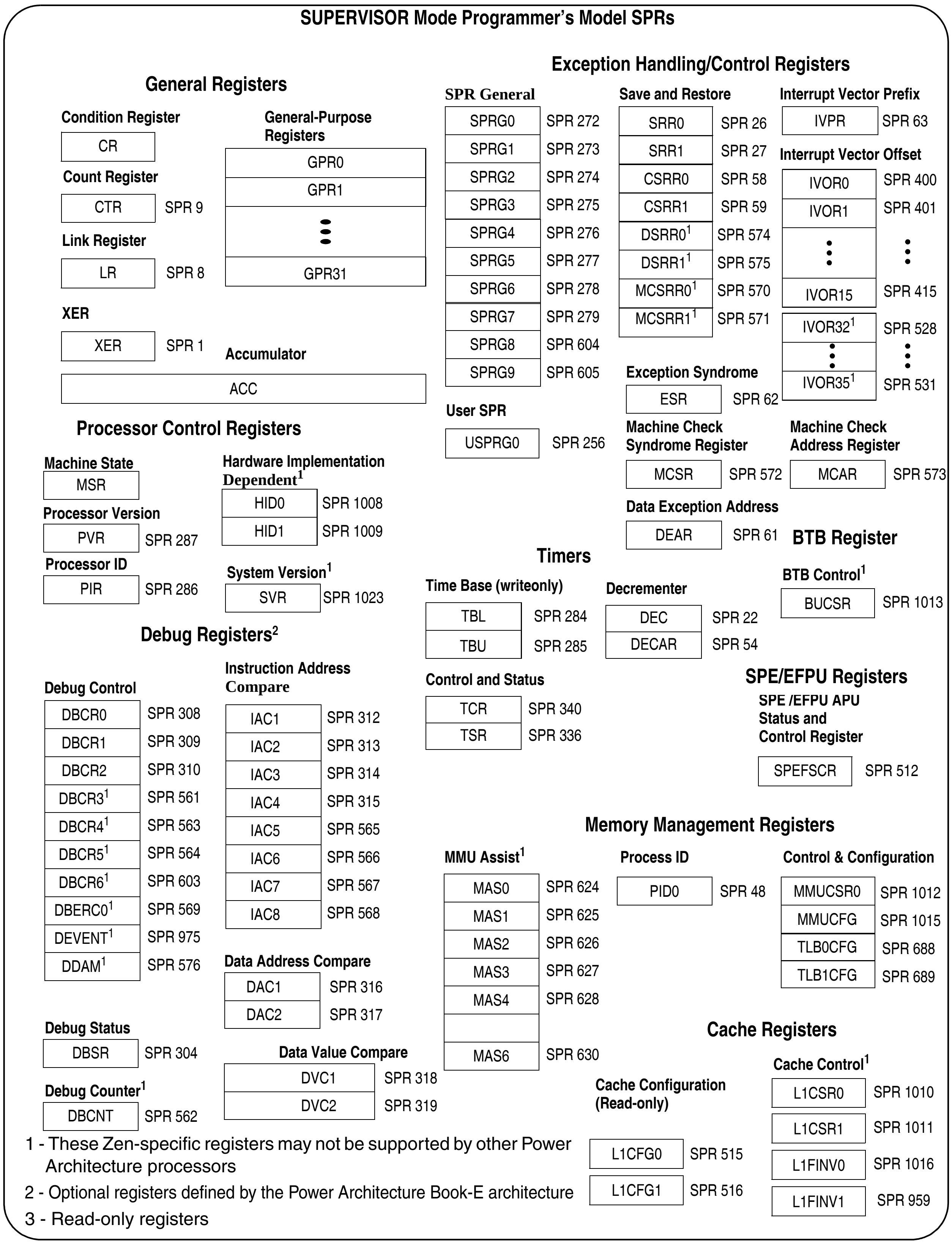


Figure 15-21. e200z759n3 Supervisor mode programmer's model SPRs

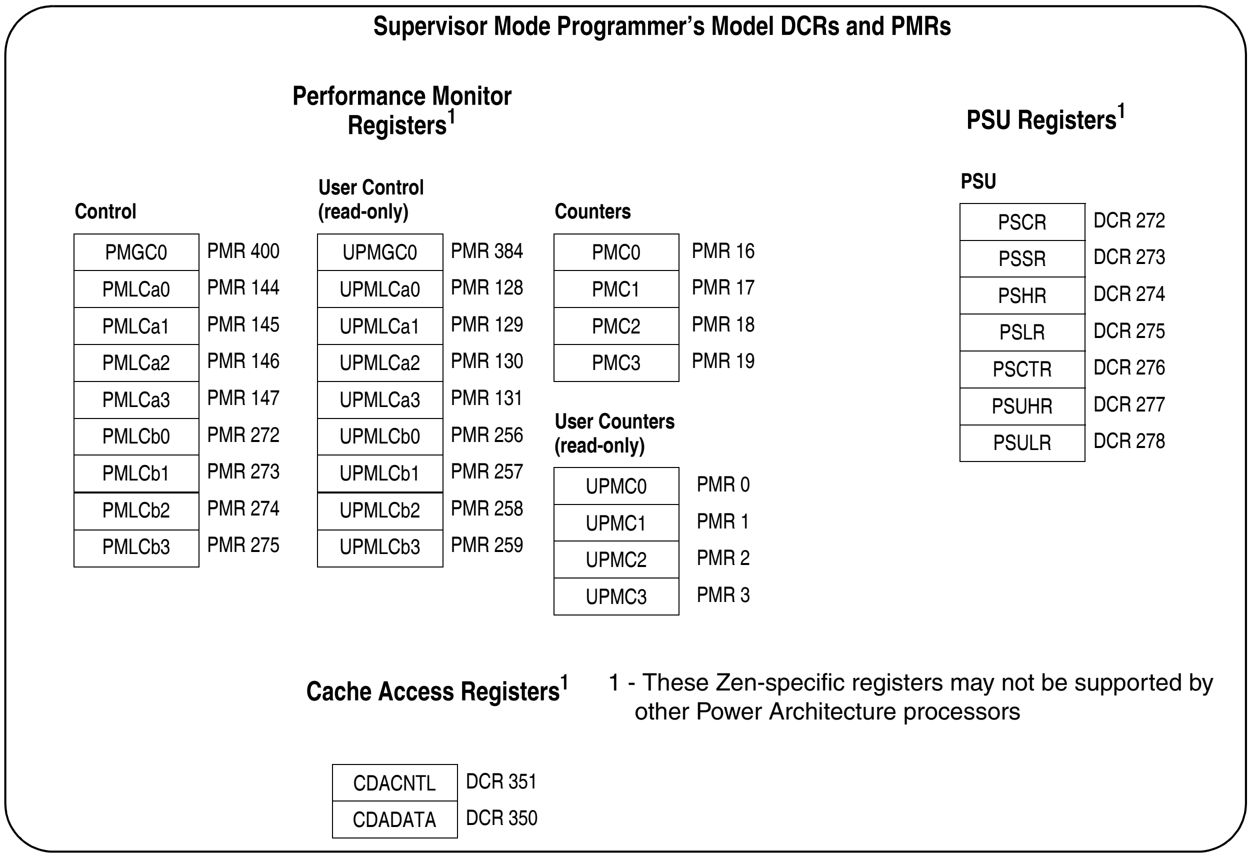


Figure 15-22. Zen Supervisor mode programmer's model DCRs and PMRs

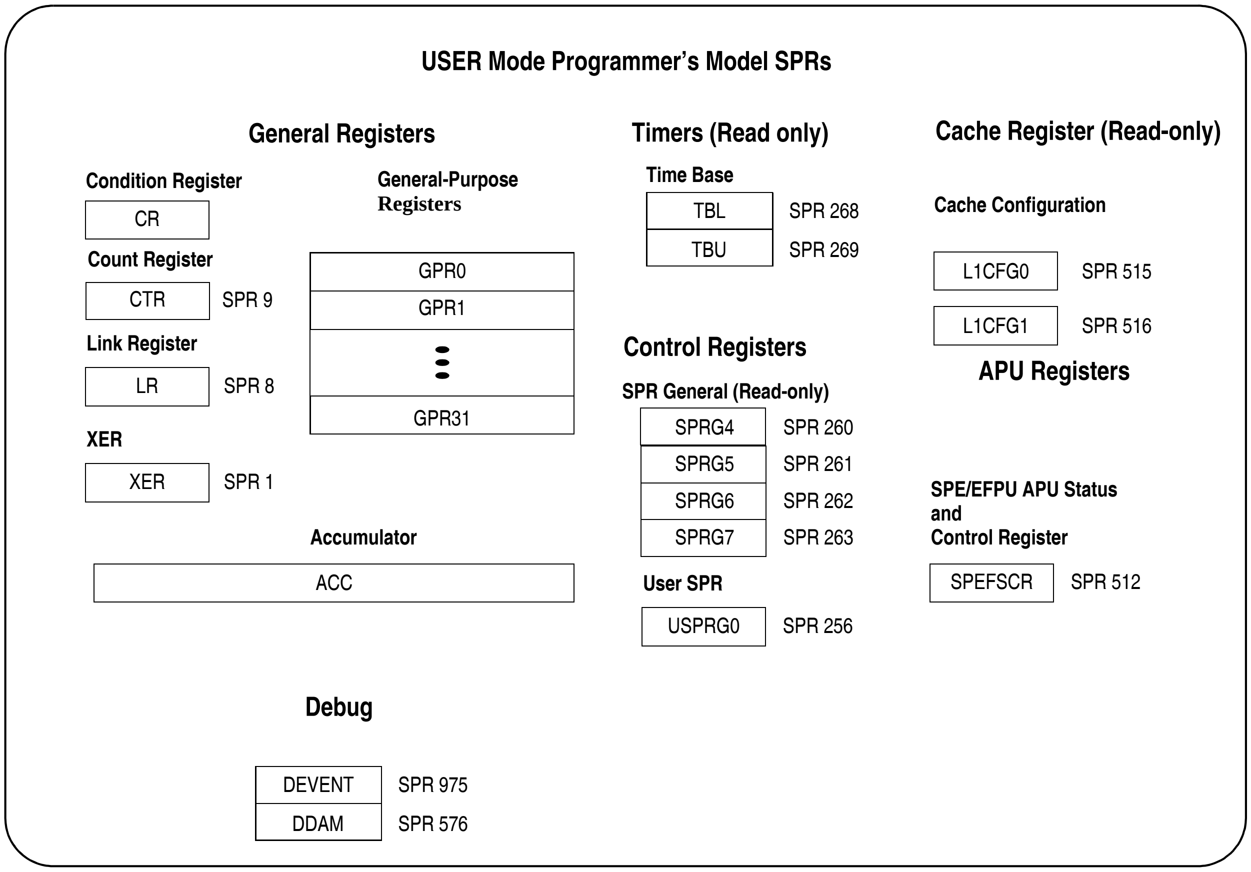


Figure 15-23. Zen User mode programmer's model SPRs

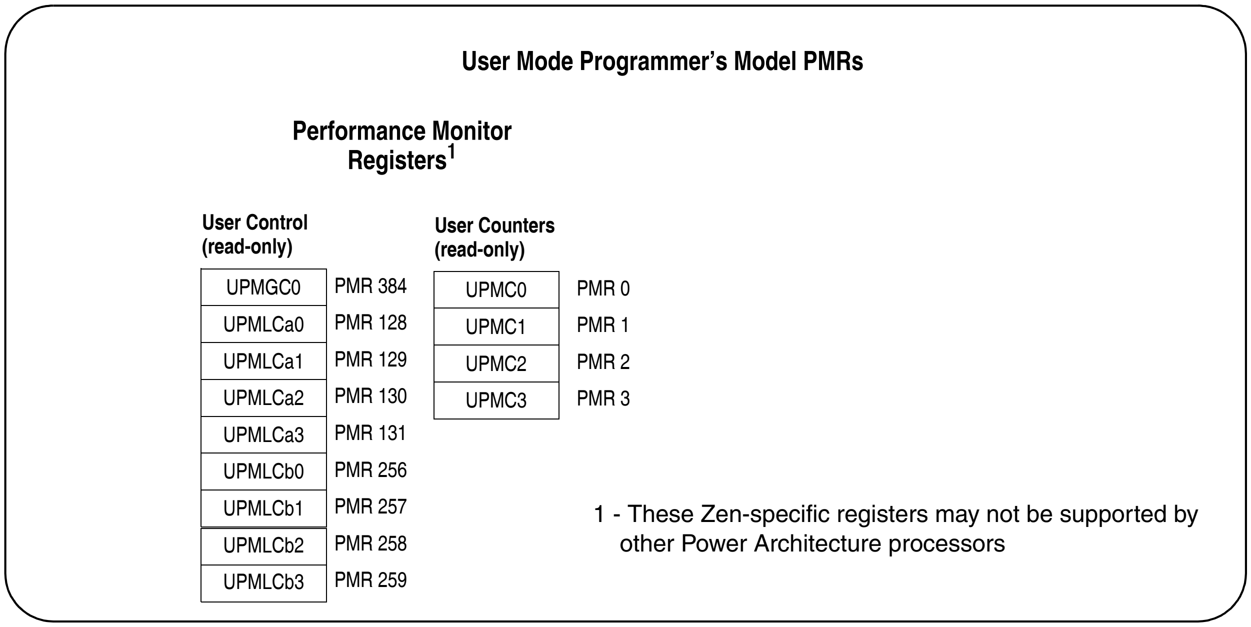


Figure 15-24. Zen User mode programmer's model PMRs

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

Figure A-1. Machine State Register (MSR)

ID
0–31

Figure A-2. Processor ID Register (PIR)

1	0	0	0	0	0	0	1	0	1	1	0	Version	MBG Reserved	Minor Rev	Major Rev	MBG ID
0	1	2	3	4	5	6	7	8	9	10	11	12–15	16–19	20–23	24–27	28–31

Figure A-3. Processor Version Register (PVR)

System Version
0–31

Figure A-4. System Version Register (SVR)

SO	OV	CA	0	Bytecnt
0	1	2	3–24	25–31

Figure A-5. Integer Exception Register (XER)

0	PIL	PPR	PTR	FP	ST	0	DLK	ILK	AP	PUO	BO	PIE	0	SPE	0	VLEMI	0	MIF	0
0–3	4	5	6	7	8	9	10	11	12	13	14	15	16–23	24	25	26	27–29	30	31

Figure A-6. Exception Syndrome Register (ESR)

MCP	IC_DPERR	CP_PERR	DC_DPERR	EXCP_ERR	IC_TPERR	DC_TPERR	IC_LKERR	DC_LKERR	0	NMI	MAV	MEA	0	IF	LD	ST	G	0	SNPERR	BUS_IRERR	BUS_DRERR	BUS_WRERR	0
0	1	2	3	4	5	6	7	8	9–10	11	12	13	14	15	16	17	18	19–25	26	27	28	29	30–31

Figure A-7. Machine Check Syndrome Register (MCSR)

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

Figure A-8. Timer Control Register (TCR)

0	1	2–3	4	5	6–31
ENW	WIS	WRS	DIS	FIS	0

Figure A-9. Timer Status Register (TSR)

0	1–7	8	9	10	11–13	14	15	16	17	18	19	20	21	22	23	24–30	31
EMCP	0	DOZE	NAP	SLEEP	0	ICR	NHR	0	TBEN	SELTBCLK	DCLREE	DCLRCE	CICLRDE	MCCLRDE	DAPUEN	0	NOPTI

Figure A-10. Hardware Implementation Dependent register 0 (HID0)

0–15	16–23	24	25–31
0	SYSCTL	ATS	0

Figure A-11. Hardware Implementation Dependent register 1 (HID1)

0–21	22	23–25	26–27	28	29–30	31
0	BBFI	0	BALLOC	0	BPRED	BPEN

Figure A-12. Branch Unit Control and Status Register (BUCSR)

0–15	16–27	28–31
0	Vector Offset	0

Figure A-13. e200z759n3 Interrupt Vector Offset Register (IVOR)

0	1	2	3–18	19–20	21–22	23	24–31
FAC	PMIE	FCECE	0	TBSEL	0	TBEE	0

Figure A-14. Performance Monitor Global Control register (PMGC0)

Bits	Field
0	FC
1	FCS
2	FCU
3	FCM1
4	FCM0
5	CE
6–8	0
9–15	EVENT
16–31	0

Figure A-15. Performance Monitor Local Control A registers (PMLCa0–PMLCa3)

Bits	Field
0–6	0
7–8	TRIGONSEL
9	TRIGGERED
10–11	0
12–13	TRIGOFFSEL
14	0
15–16	TRIGONCNTL
17	0
18–19	TRIGOFFCNTL
20	0
21–23	THRESHMUL
24–25	0
26–31	THRESHOLD

Figure A-16. Performance Monitor Local Control B registers (PMLCb0–PMLCb3)

Bits	Field
0–16	CNT1
17–31	CNT2

Figure A-17. Degug Counter register (DBCNT)

Bits	Field
0	EDM
1	IDM
2–3	RST
4	ICMP
5	BRT
6	IRPT
7	TRAP
8	IAC1
9	IAC2
10	IAC3
11	IAC4
12–13	DAC1
14–15	DAC2
16	RET
17	IAC5
18	IAC6
19	IAC7
20	IAC8
21	DEVT1
22	DEVT2
23	DCNT1
24	DCNT2
25	CIRPT
26	CRET
27–30	0
31	FT

Figure A-18. Debug Control 0 register (DBCR0)

Bits	Field
0–1	IAC1US
2–3	IAC1ER
4–5	IAC2US
6–7	IAC2ER
8–9	IAC12M
10–15	0
16–17	IAC3US
18–19	IAC3ER
20–21	IAC4US
22–23	IAC4ER
24–25	IAC34M
26–31	0

Figure A-19. Debug Control 1 register (DBCR1)

Bits	Field
0–1	DAC1US
2–3	DAC1ER
4–5	DAC2US
6–7	DAC2ER
8–9	DAC12M
10	DAC1LNK
11	DAC2LNK
12–13	DVC1M
14–15	DVC2M
16–23	DVC1BE
24–31	DVC2BE

Figure A-20. Debug Control 2 register (DBCR2)

Bits	Field
0	DEVT1C1
1	DEVT2C1
2	ICMPC1
3	IAC1C1
4	IAC2C1
5	IAC3C1
6	IAC4C1
7	DAC1RC1
8	DAC1WC1
9	DAC2RC1
10	DAC2WC1
11	IRPTC1
12	RETC1
13	DEVT1C2
14	DEVT2C2
15	ICMPC2
16	IAC1C2
17	IAC2C2
18	IAC3C2
19	IAC4C2
20	DAC1RC2
21	DAC1WC2
22	DAC2RC2
23	DAC2WC2
24	DEVT1T1
25	DEVT2T1
26	IAC1T1
27	IAC3T1
28	DAC1RT1
29	DAC1WT1
30	CNT2T1
31	CONFIG

Figure A-21. Debug Control 3 register (DBCR3)

0	DVC1C	0	DVC2C	0	DAC1XM	DAC2XM	0
0	1	2	3	4–15	16–19	20–23	24–31

Figure A-22. Debug Control 4 register (DBCR4)

IAC5US	IAC5ER	IAC6US	IAC6ER	IAC56M	0	IAC7US	IAC7ER	IAC8US	IAC8ER	IAC78M	0
0–1	2–3	4–5	6–7	8–9	10–15	16–17	18–19	20–21	22–23	24–25	26–31

Figure A-23. Debug Control 5 register (DBCR5)

IAC1XM	IAC2XM	IAC3XM	IAC4XM	IAC5XM	IAC6XM	IAC7XM	IAC8XM
0–3	4–7	8–11	12–15	16–19	20–23	24–27	28–31

Figure A-24. Debug Control 6 register (DBCR6)

IDE	UDE	MRR	ICMP	BRT	IRPT	TRAP	IAC1	IAC2	IAC3	IAC4	DAC1R	DAC1W	DAC2R	DAC2W	RET	0	DEVT1	DEVT2	DCNT1	DCNT2	CIRPT	CRET	VLES	DAC_OFST	CNT1TRG
0	1	2–3	4	5	6	7	8	9	10	11	12	13	14	15	16	17–20	21	22	23	24	25	26	27	28–30	31

Figure A-25. Debug Status Register (DBSR)

MCLK	ERR	0	RESET	HALT	STOP	DEBUG	WAIT	0	1
0	1	2	3	4	5	6	7	8	9

Figure A-26. OnCE Status Register (OSR)

R/W	GO	EX	RS[0:6]
0	1	2	3–9

Figure A-27. OnCE Command Register (OCMD)

0	I_DMDIS	0	I_DVLE	I_DI	I_DM	0	I_DE	D_DMDIS	0	D_DW	D_DI	D_DM	D_DG	D_DE	0	WKUP	FDB	DR
0–7	8	9–10	11	12	13	14	15	16	17–18	19	20	21	22	23	24–28	29	30	31

Figure A-28. OnCE Control Register (OCR)

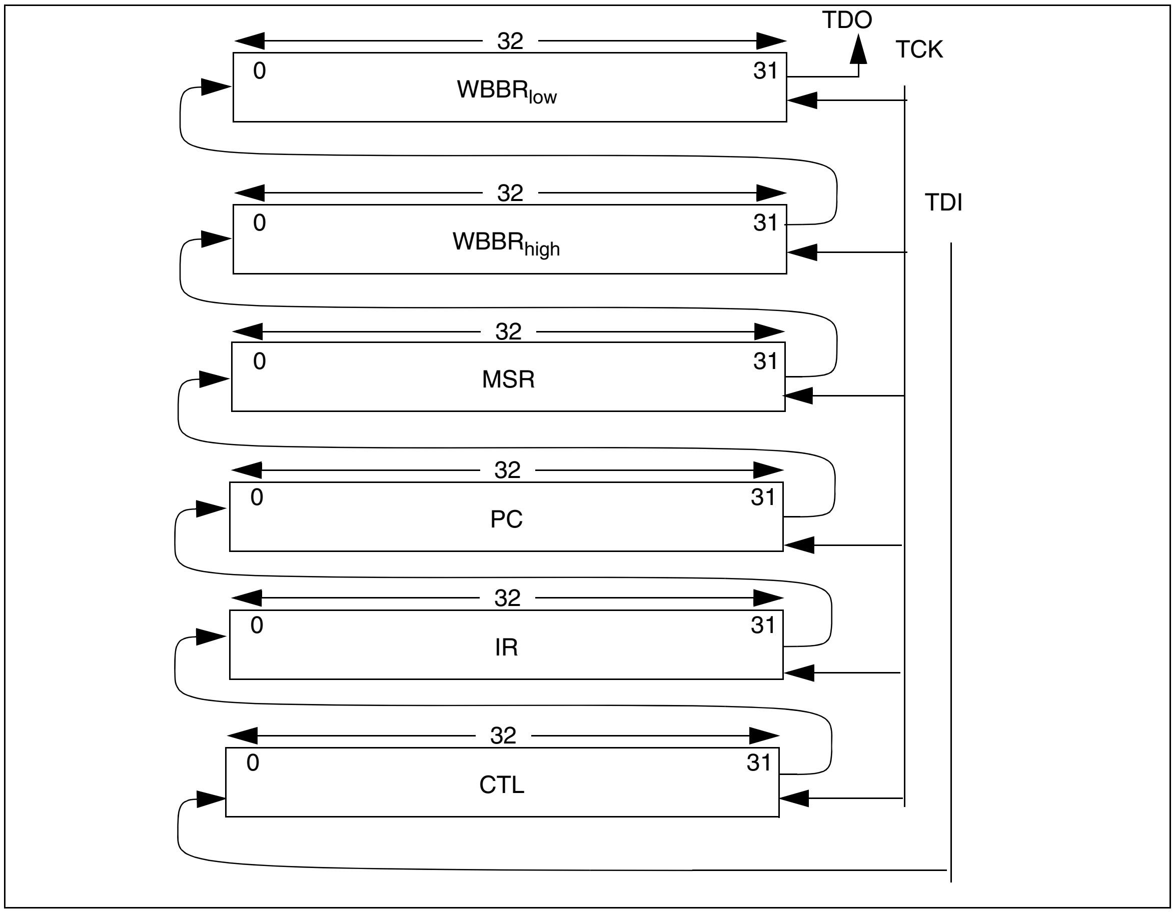


Figure A-29. CPU Scan Chain Register (CPUSCR)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
*											IRSTAT13	IRSTAT12	IRSTAT11	IRSTAT10	WAITING	PCOFST				PCINV	FFRA	IRSTAT0	IRSTAT1	IRSTAT2	IRSTAT3	IRSTAT4	IRSTAT5	IRSTAT6	IRSTAT7	IRSTAT8	IRSTAT9

Figure A-30. Control State register (CTL)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
SOVH	OVH	FGH	FXH	FINVH	FDBZH	FUNFH	FOVFH	0		FINXS	FINVS	FDBZS	FUNFS	FOVFS	MODE	SOV	OV	FG	FX	FINV	FDBZ	FUNF	FOVF	0	FINXE	FINVE	FDBZE	FUNFE	FOVFE	FRMC	

Figure A-31. SPE/EFPU Status and Control Register (SPEFSCR)

WID	WDD	0	DCWM	DCWA	0	DCECE	DCEI	0	DCEDT	DCSLC	DCUL	DCLO	DCLFC	DCLOA	DCEA	0	DCBZ32	DCABT	DCINV	DCE	

Figure A-32. L1 Cache Control and Status Register 0 (L1CSR0)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Figure A-32. L1 Cache Control and Status Register 0 (L1CSR0)

0–14	15	16	17	18–19	20	21	22	23	24	25–26	27–28	29	30	31
0	ICECE	ICEI	0	ICEDT	0	ICUL	ICLO	ICLFC	ICLOA	ICEA	0	ICABT	ICINV	ICE

Figure A-33. L1 Cache Control and Status Register 1 (L1CSR1)

0–1	2	3	4	5–6	7–8	9–10	11	12	13–20	21–31
CARCH	CWPA	CFAHA	DCFISWA	0	DCBSIZE	DCREPL	DCLA	DCECA	DCNWAY	DCSIZE

Figure A-34. L1 Cache Configuration register 0 (L1CFG0)

0–3	4	5–6	7–8	9–10	11	12	13–20	21–31
	ICFISWA	0	ICBSIZE	ICREPL	ICLA	ICECA	ICNWAY	ICSIZE

Figure A-35. L1 Cache Configuration register 1 (L1CFG1)

0–5	6–7	8–19	20–26	27–29	30–31
0	CWAY	0	CSET	0	CCMD

Figure A-36. L1 Flush/Invalidate register (L1FINV0,1)

0–7	8–14	15–16	17–20	21–25	26–27	28–29	30–31
0	RASIZE	0	NPIDS	PIDSIZE	0	NTLBS	MAVN

Figure A-37. MMU Configuration register (MMUCFG)

0–7	8–11	12–15	16	17	18	19	20–31
ASSOC	MINSIZE	MAXSIZE	IPROT	AVAIL	P2PSA	0	NENTRY

Figure A-38. TLB Configuration register (TLB0CFG, TLB1CFG)

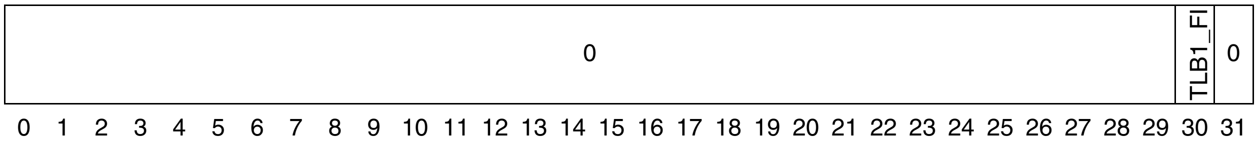


Figure A-39. MMU Control and Status Register 0 (MMUCSR0)

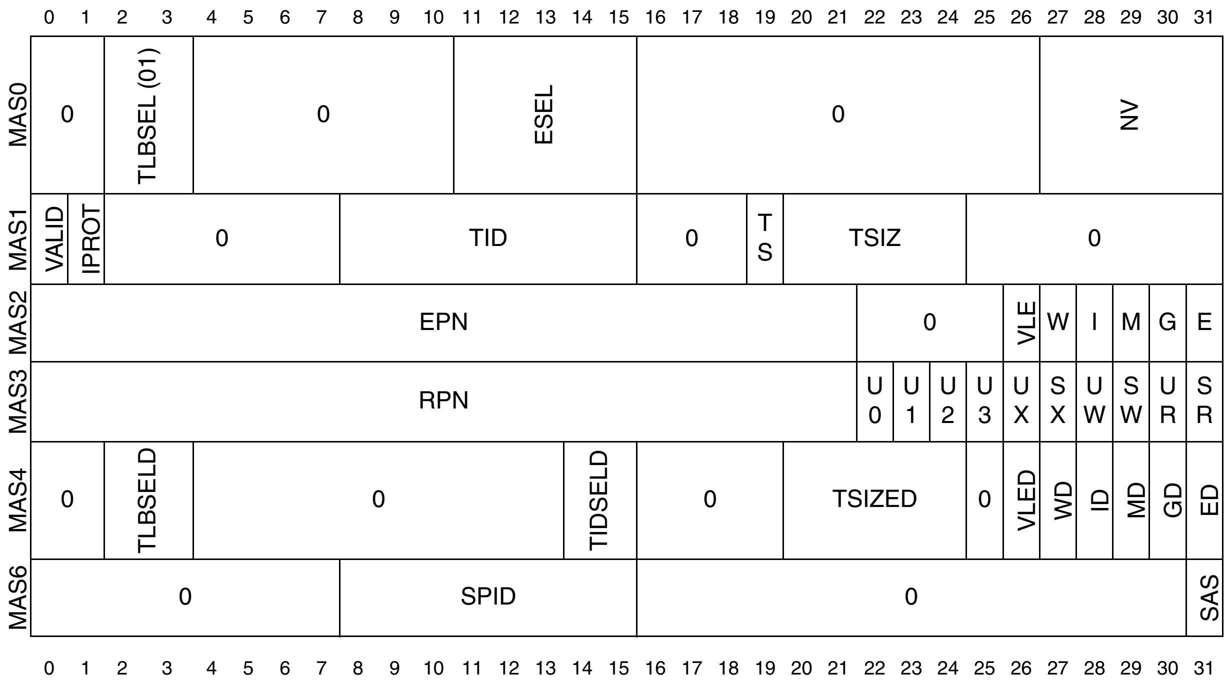
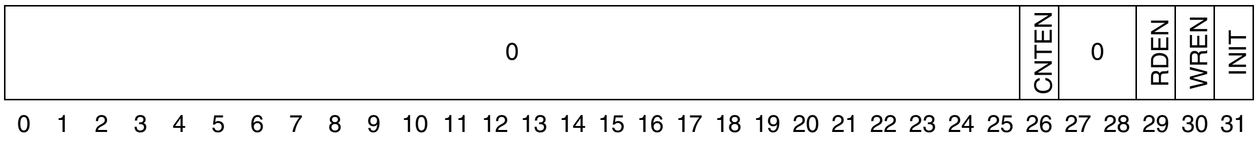
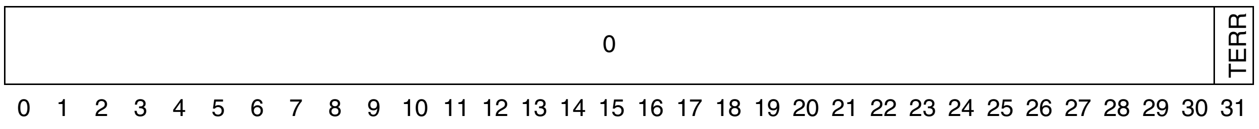


Figure A-40. MMU assist registers summary



DCR - 272; Read/Write; Reset - 0x0

Figure A-41. Parallel Signature Control Register (PSCR)



DCR - 273; Read/Write; Reset -Unaffected

Figure A-42. Parallel Signature Status Register (PSSR)

High Signature
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

DCR - 274; Read/Write; Reset -Unaffected

Figure A-43. Parallel Signature High Register (PSHR)

Low Signature
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

DCR - 275; Read/Write; Reset -Unaffected

Figure A-44. Parallel Signature Low Register (PSLR)

Counter
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

DCR - 276; Read/Write; Reset -Unaffected

Figure A-45. Parallel Signature Counter Register (PSCTR)

High Signature Update Data
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

DCR - 277; Write-only; Reset -Unaffected

Figure A-46. Parallel Signature Update High Register (PSUHR)

Low Signature Update Data
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

DCR - 278; Write-only; Reset -Unaffected

Figure A-47. Parallel Signature Update Low Register (PSULR)

Appendix B
Revision History

This appendix provides a list of the major differences between revisions of the *e200z759n3 Core Reference Manual*.

Table B-1. Revision history

Revision	Date	Description of changes
1	27 Nov 2012	initial release.
2	12 Dec 2014	In Figure 13-53 (Data Acquisition Message format), changed (6 bit) value from TCODE(011011) to TCODE(000111).

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: e200z759n3CRM
Rev. 2
January 2015