e200z1 Power Architecture™ Core Reference Manual

Supports
e200z1

e200z1RM
Rev. 0
09/2008

How to Reach Us:

Home Page:
www.freescale.com

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

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

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

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

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

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

Document Number: e200z1RM
Rev. 0, 09/2008

Contents

Contents

Contents

Contents

Contents

Paragraph Number	Title	Page Number

Contents

Contents

Contents

Contents

Contents

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

Figures

Figures

Figures

Figures

Tables

Tables

Tables

Tables

Table Number	Title	Page Number

Chapter 1
e200z1 Overview

1.1 Overview of the e200z1 Cores

The e200 processor family is a set of CPU cores that implement low-cost versions of the Power Architecture™ Book E architecture. e200 processors are designed for deeply embedded control applications, which require low cost solutions rather than maximum performance.

The processors integrate an integer execution unit, branch control unit, instruction fetch and load/store units, and a multi-ported register file capable of sustaining three read and two 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 some cases.

The e200z1 core is a single-issue, 32-bit Power Architecture Book E compliant design with 32-bit general purpose registers (GPRs). Power Architecture Book E floating-point instructions are not supported by e200 in hardware, but are trapped and may be emulated by software.All arithmetic instructions that execute in the core operate on data in the general purpose registers (GPRs).

In addition to the base Power Architecture Book E instruction set support, the core also implements the VLE (variable-length encoding) APU, providing improved code density. The VLE APU is further documented in the *PowerPC™ VLE APU Definition, Version 1.01*, a separate document.

In the remainder of this document, the e200z1 core is also referred to as the “e200z1 core” or “e200 core” when referring to the whole e200 family.

1.1.1 Features

The following is a list of some of the key features of the e200z1 core:

- 32-bit Power Architecture Book E programmer’s model
- Single issue, 32-bit 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 acceleration using Branch Target Buffer
- Supports independent instruction and data accesses to different memory subsystems, such as SRAM and Flash memory via independent Instruction and Data bus interface units (BIUs).
- Load/store unit

  - 1 cycle load latency
  - Fully pipelined
  - Big- and Little-endian support
  - Misaligned access support
  - Zero load-to-use pipeline bubbles for aligned transfers
- Power management
  - Low power design
  - Power saving modes: doze, nap, sleep, and wait
  - Dynamic power management of execution units
- Testability
  - Synthesizeable, full MuxD scan design
  - ABIST/MBIST for optional memory arrays

1.1.2 Microarchitecture Summary

The e200 processor utilizes a four stage pipeline for instruction execution. The Instruction Fetch (stage 1), Instruction Decode/Register file Read/Effective Address Calculation (stage 2), Execute/Memory Access (stage 3), and Register Writeback (stage 4) stages operate in an overlapped fashion, allowing single clock instruction execution for most instructions.

The integer execution unit consists of a 32-bit Arithmetic Unit (AU), a Logic Unit (LU), a 32-bit Barrel shifter (Shifter), a Mask-Insertion Unit (MIU), a Condition Register manipulation Unit (CRU), a Count-Leading-Zeros unit (CLZ), a 32x32 Hardware Multiplier array, result feed-forward hardware, and a hardware divider.

Arithmetic and logical operations are executed in a single cycle with the exception of the divide instructions. A Count-Leading-Zeros unit operates in a single clock cycle.

The Instruction Unit contains a PC incrementer and a dedicated Branch Address adder to minimize delays during change of flow operations. Sequential prefetching is performed to ensure a supply of instructions into the execution pipeline. Branch target prefetching from the BTB is performed to accelerate certain taken branches in the e200z31. Prefetched instructions are placed into an instruction buffer with 6 entries in e200z1, each capable of holding a single 32-bit instruction or a pair of 16-bit instructions.

Conditional branches which are not taken execute in a single clock. Branches with successful target prefetching have an effective execution time of 1 clock. All other taken branches have an execution time of two clocks.

Memory load and store operations are provided for byte, halfword, and word (32-bit) 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. Also, a load-to-use dependency does not incur any pipeline bubbles for most cases.

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.

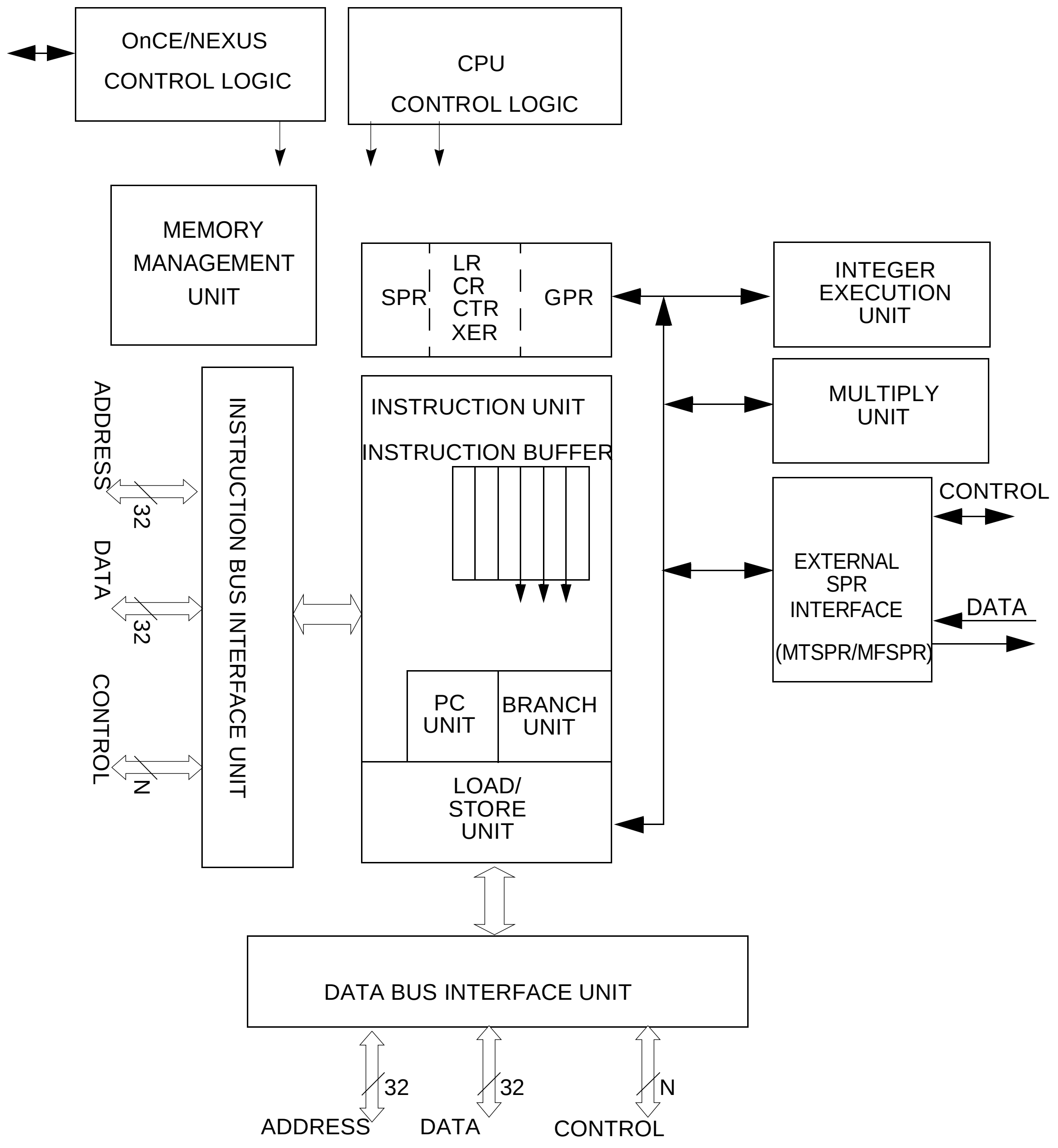


Figure 1-1. e200z1 Block Diagram

1.1.2.1 Instruction Unit Features

The features of the e200 Instruction unit are:

- 32-bit instruction fetch path supports fetching of one 32-bit instruction per clock, or up to two 16-bit VLE instructions per clock.

- Instruction buffer with 4 entries in e200z1, each holding a single 32-bit instruction, or a pair of 16-bit instructions
- Dedicated PC incrementer supporting instruction prefetches
- Branch unit with dedicated branch address adder and small branch target buffer logic supporting single cycle of execution of certain branches, two cycles for all others

1.1.2.2 Integer Unit Features

The e200 integer unit supports single cycle execution of most integer instructions:

- 32-bit AU for arithmetic and comparison operations
- 32-bit LU for logical operations
- 32-bit priority encoder for count leading zero's function
- 32-bit single cycle barrel shifter for shifts and rotates
- 32-bit mask unit for data masking and insertion
- Divider logic for signed and unsigned divide in 6-16 clocks with minimized execution timing
- 32x32 hardware multiplier array supports single-cycle 32x32->32 multiply

1.1.2.3 Load/Store Unit Features

The e200 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
- 32-bit interface to memory dedicated memory interface on e200z1)

1.1.2.4 e200z1 System Bus Features

The features of the e200z1 System Bus interface are as follows:

- Independent Instruction and Data Buses
- AMBA AHB Lite Rev 2.0 Specification with support for ARM v6 AMBA Extensions
  - Exclusive Access Monitor
  - Byte Lane Strobes
  - Cache Allocate Support
- 32-bit address bus plus attributes and control on each bus
- 32-bit read data bus for Instruction Interface
- Separate uni-directional 32-bit read data bus and 32-bit write data bus for Data Interface
- Overlapped, in-order accesses

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
- 8-entry Fully-associative TLB
- Support for multiple page sizes from 4-Kbyte to 4-Gbyte
- Entry Flush Protection

Chapter 2
Register Model

This section describes the registers implemented in the e200z1 core. It includes an overview of registers defined by the PowerPC Book E architecture, highlighting differences in how these registers are implemented in the e200 core, and provides a detailed description of e200-specific registers. Full descriptions of the architecture-defined register set are provided in Power Architecture Book E Specification.

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

Figure 2-1, Figure 2-2, and Figure 2-3 show the e200 register set including the registers which are accessible while in supervisor mode, and the registers which are accessible in user mode. The number to the right of the special-purpose registers (SPRs) is the decimal number used in the instruction syntax to access the register (for example, the integer exception register (XER) is SPR 1).

NOTE

e200z1 is a 32-bit implementation of the Power Architecture Book E specification. In this document, register bits are sometimes numbered from bit 0 (Most Significant Bit) to 31 (Least Significant Bit), rather than the Book E numbering scheme of 32:63, thus register bit numbers for some registers in Book E are 32 higher.

Where appropriate, the Book E defined bit numbers are shown in parentheses.

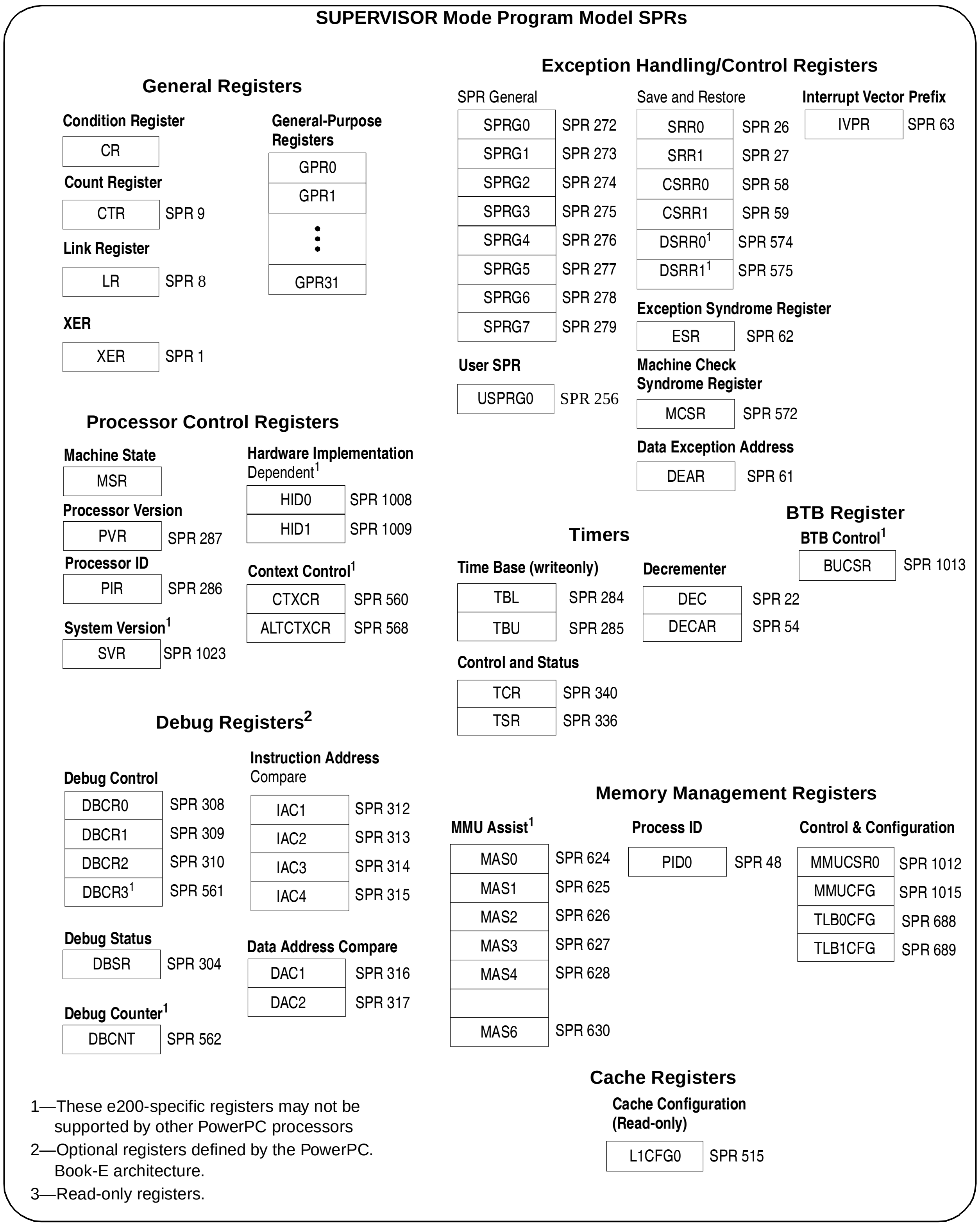


Figure 2-1. e200z1 Supervisor Mode Programmer's Model SPRs

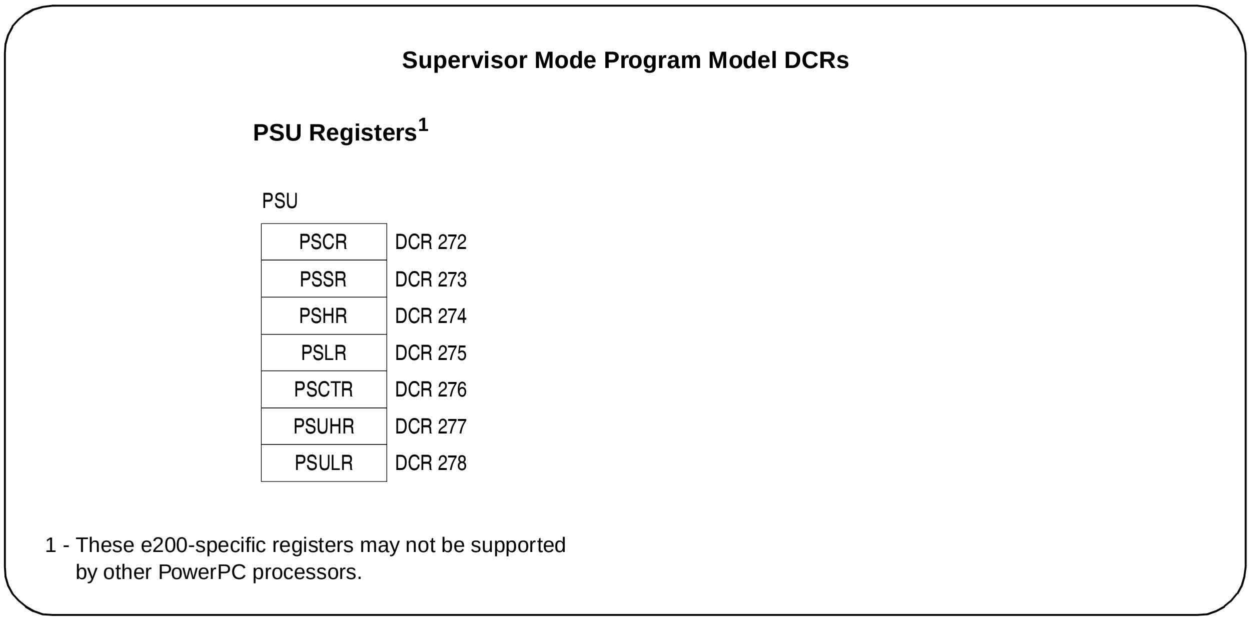

Supervisor Mode Program Model DCRs
PSU Registers[1]
PSU
PSCR
DCR 272
PSSR
DCR 273
PSHR
DCR 274
PSLR
DCR 275
PSCTR
DCR 276
PSUHR
DCR 277
PSULR
DCR 278
1 - These e200-specific registers may not be supported by other PowerPC processors.

Figure 2-2. e200 Supervisor Mode Program Model Device Control Registers (DCRs)

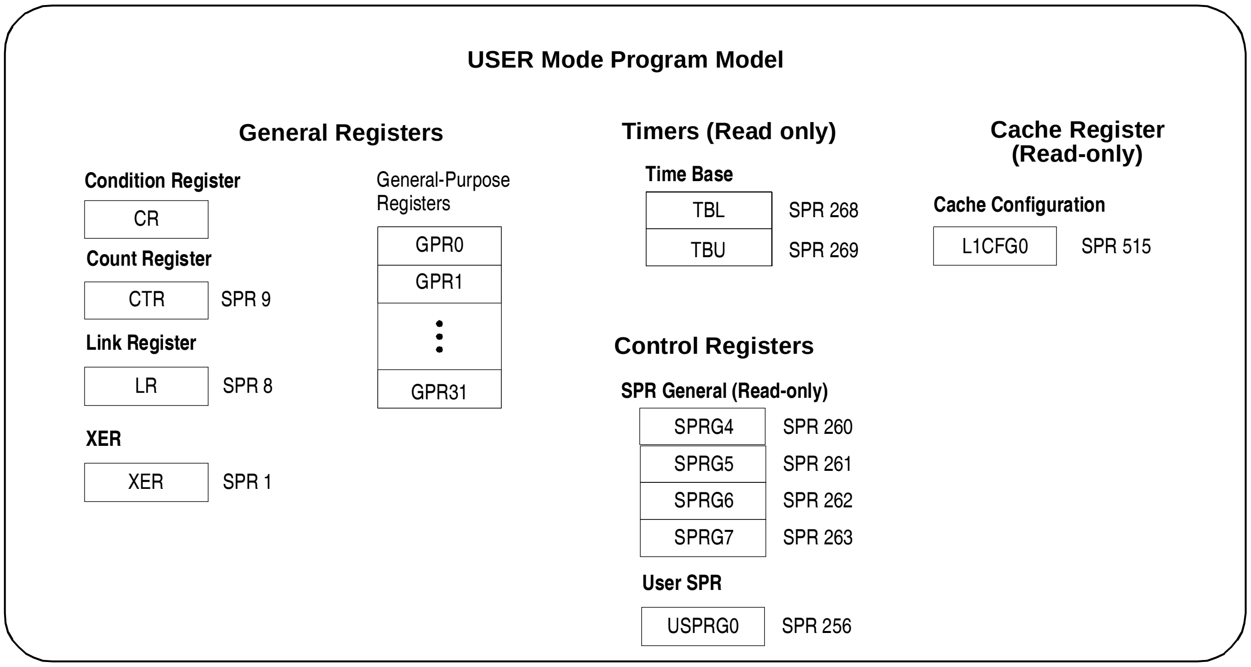


Figure 2-3. e200 User Mode Program Model

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

2.1 Power Architecture Book E Registers

e200 supports most of the registers defined by *Power Architecture™ Book E Specification*. Notable exceptions are the Floating Point registers FPR0–FPR31 and FPSCR. e200does not support the Book E floating-point architecture in hardware. The e200-supported Power Architecture Book E registers are described as follows (e200-specific registers are described in the Section 2.2, “e200-Specific Special Purpose Registers”).

- 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 32-bit GPRs (GPR0–GPR31) serve as data source or destination registers for integer instructions and provide data for generating addresses.
  - 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, *Power Architecture Book E Specification*.

  The remaining user-level registers are SPRs. Note that the Power Architecture Book E 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 *Power Architecture Book E Specification* 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 *Power Architecture Book E Specification*.
  - 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 *Power Architecture Book E Specification*.
  - 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 *Power Architecture Book E Specification*.
  - SPRG4–SPRG7. The Power Architecture Book E architecture defines Software-Use Special Purpose Registers (SPRGs). SPRG4 through SPRG7 are accessible in a read-only fashion by user-level software. e200 does not allow user mode access to the SPRG3 register (defined as implementation dependent by Book E).
  - USPRG0. The PowerPC 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 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, se_rfi**, **se_rfci**, **se_rfdi)** instructions. It can be read by the Move from Machine State Register (**mfmsr)** instruction. When an interrupt occurs, the contents of the MSR are saved to one of the machine state save/restore registers (SRR1, CSRR1, DSRR1).
  - Processor version register (PVR). This register is a read-only register that identifies the version (model) and revision level of the processor.
  - Processor Identification Register (PIR). This read-only 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 optional MMU as an extension to the effective address, and by the optional external Nexus2/3 modules for Ownership Trace message generation. Although the Power Architecture Book E allows for multiple PIDs, e200z1 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. e200 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 which can generate the same interrupt.
  - Interrupt Vector Prefix Register (IVPR). This register together with hardwired offsets which replace the IVOR0-15 registers 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 **rfi** or **se_rfi** executes.

    - Critical Save/Restore register 1 (CSRR1). The CSRR1 register is used to save machine state from the MSR on critical interrupts, and to restore machine state when **rfci** or **se_rfci** executes.
  - Debug facility registers
    - Debug Control Registers (DBCR0-DBCR2). These registers provide control for enabling and configuring debug events.
    - Debug Status Register (DBSR). This register contains debug event status.
    - Instruction Address Compare registers (IAC1-IAC4). These registers contain addresses and/or masks which are used to specify Instruction Address Compare debug events.
    - Data address compare registers (DAC1-2). These registers contain addresses and/or masks which are used to specify Data Address Compare debug events.
    - e200 does **not** implement the Data Value Compare registers (DVC1 and DVC2).
  - 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 e200-Specific Special Purpose Registers

The PowerPC Book E architecture allows implementation-specific special purpose registers. Those incorporated in the e200 core are as follows:

- User-level registers—The user-level registers can be accessed by all software with either user or supervisor privileges. They include the following:
  - The L1 Cache Configuration register (L1CFG0). This read-only register allows software to query the configuration of the L1 Cache. For the e200z1, this register returns all zeros indicating no cache is present.
- Supervisor-level registers—The following supervisor-level registers are defined in e200 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 Syndrome register (MCSR). This register provides a syndrome to differentiate between the different kinds of conditions which can generate a Machine Check.
  - 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 **rfdi** or **se_rfdi** 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 **rfdi** or **se_rfdi** executes.
- Debug Facility Registers
  - Debug Control Register 3 (DBCR3)—This register provides control for debug functions not described in PowerPC Book E architecture.
  - Debug Counter Register (DBCNT)—This register provides counter capability for debug functions.
- Branch Unit Control and Status Register (BUCSR) controls operation of the BTB in e200z1.
- L1 Cache Configuration Register (L1CFG0) is a read-only register that allows software to query the configuration of the L1 Cache. For the e200z1, this register returns all zeros.
- MMU Configuration Register (MMUCFG) is a read-only register that allows software to query the configuration of the MMU.
- Memory Management Registers
  - MMU Assist (MAS0–MAS4, MAS6) registers. These registers provide the interface to the e200 core from the Memory Management Unit.
  - MMU Control and Status Register (MMUCSR0) 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 SoC which includes an e200 Power Architecture processor.

Note that it is not guaranteed that the implementation of e200 core-specific registers is consistent among Power Architecture processors, although other processors may implement similar or identical registers. All e200 SPR definitions are compliant with the Freescale EIS specification definitions as documented in the *EREF: A Programmer's Reference Manual for Freescale Book E Processors*.

2.3 e200-Specific Device Control Registers

In addition to the SPRs described above, implementations may also choose to implement one or more Device Control Registers (DCRs). e200 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 9.7, "Parallel Signature Unit."

2.4 Special Purpose Register Descriptions

2.4.1 Machine State Register (MSR)

The Machine State Register defines the state of the processor. Chapter 5, “Interrupts and Exceptions,” describes how the MSR is affected when Interrupts occur. The e200 MSR is shown in Figure 2-4.

e200z1	0	UCLE	Allocated	0	WE	CE	0	EE	PR	FP	ME	FE0	0	DE	FE1	0	IS	DS	0
	0 1 2 3 4	5	6	7 8 9 10 11 12	13	14	15	16	17	18	19	20	21	22	23	24 25	26	27	28 29 30 31

Read/Write; Reset—0x0

Figure 2-4. Machine State Register (MSR)

The MSR bits are defined in Table 2-1.

Table 2-1. MSR Field Descriptions

Bit(s)	Name	Description
0:4 (32:36)	—	Reserved[1]
5 (37)	UCLE[2]	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)	Allocated	Allocated[3] - Allocated for SPE Not supported on e200z1
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)	—	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.

Table 2-1. MSR Field Descriptions (continued)

Bit(s)	Name	Description
17 (49)	PR	Problem State 0 The processor is in supervisor mode, can execute any instruction, and can access any resource (for example, GPRs, SPRs, MSR, etc.). 1 The processor is in user mode, cannot execute any privileged instruction, and cannot access any privileged resource.
18 (50)	FP	Floating-Point Available 0 Floating point unit is unavailable. The processor cannot execute floating-point instructions, including floating-point loads, stores, and moves. (A FP Unavailable interrupt is generated on attempted execution of floating point instructions). 1 Floating Point unit is available. The processor can execute floating-point instructions. Note that for e200, the Book E floating point unit is not supported in hardware, and an Unimplemented Operation exception will be generated for attempted execution of Power Architecture Book E floating point instructions when FP is set.
19 (51)	ME	Machine Check Enable 0 Machine Check interrupts are disabled. 1 Machine Check interrupts are enabled.
20 (52)	FE0	Floating-point exception mode 0 (not used by e200)
21 (53)	—	Reserved[1]
22 (54)	DE	Debug Interrupt Enable 0 Debug interrupts are disabled. 1 Debug interrupts are enabled.
23 (55)	FE1	Floating-point exception mode 1 (not used by e200)
24 (56)	—	Reserved[1]
25 (57)	—	Reserved[1]
26 (58)	IS	Instruction Address Space 0 The processor directs all instruction fetches to address space 0 (TS=0 in the relevant TLB entry). 1 The processor directs all instruction fetches to address space 1 (TS=1 in the relevant TLB entry).
27 (59)	DS	Data Address Space 0 The processor directs all data storage accesses to address space 0 (TS=0 in the relevant TLB entry). 1 The processor directs all data storage accesses to address space 1 (TS=1 in the relevant TLB entry).
28:29 (60:61)	—	Reserved[1]
30:31 (62:63)	—	Reserved[1]

[1] This bit is not implemented, is read as zero, and writes are ignored.
[2] This bit is implemented but ignored because no cache is implemented
[3] This bits is should be written with zero for future compatibility.

2.4.2 Processor ID Register (PIR)

The processor ID for the processor core is contained in the Processor ID Register (PIR), shown in Figure 2-5. The contents of the PIR register are a reflection of hardware input signals to the e200 core. This register is read-only.

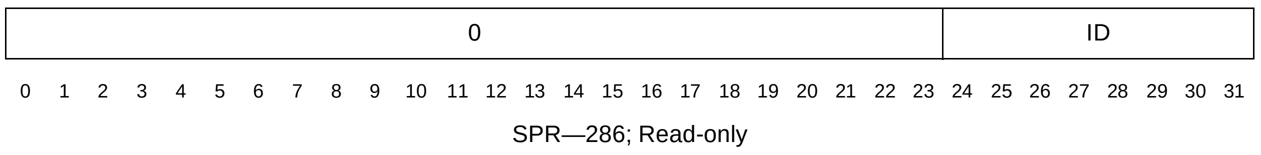


Figure 2-5. Processor ID Register (PIR)

The PIR fields are defined in Table 2-2.

Table 2-2. PIR Field Descriptions

Bits	Name	Description
0:23	—	These bits always reads 0.
24:31	ID	These bits are a reflection of the values provided on the *p_cpuid*[0:7] input signals.

2.4.3 Processor Version Register (PVR)

The Processor Version Register (PVR), shown in Figure 2-6, contains the processor version number for the processor core.

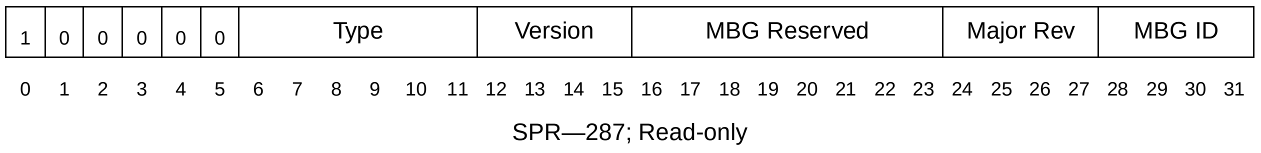


Figure 2-6. Processor Version Register (PVR)

The PVR bit fields are shown in Table 2-3.

Table 2-3. PVR Field Descriptions

Bits	Name	Description
0–5	Reserved	Reserved
6–11	Type	A 6-bit number that, together with the version number, uniquely identifies a particular processor version.
12–15	Version	A 4-bit number that, together with the version number, uniquely identifies a particular processor version.
16–23	Reserved	Reserved

Table 2-3. PVR Field Descriptions (continued)

Bits	Name	Description
24–27	Major Rev	A 4-bit number that, together with the ID number, distinguishes between various releases of a particular version (that is, an engineering change level). The value of the revision portion of the PVR is implementation-specific. The processor revision level is changed for each revision of the device.
28–31	MBG ID	A 4-bit number that, together with the major revision number, distinguishes between various releases of a particular version (that is, an engineering change level). The value of the revision portion of the PVR is implementation-specific. The processor revision level is changed for each revision of the device.

2.4.4 System Version Register (SVR)

The System Version Register (SVR), shown in Figure 2-7, contains system version information for an e200-based SoC.

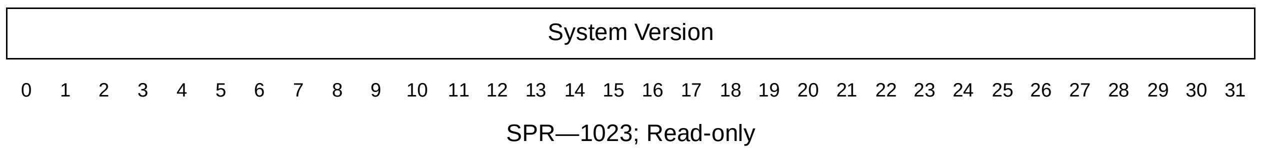


Figure 2-7. System Version Register (SVR)

This register is used to specify a particular implementation of an e200-based system. This register is read-only. The SVR bit fields are shown in Table 2-4.

Table 2-4. SVR Field Descriptions

Bits	Name	Description
0–31	Version	SVR number is SoC specific.

2.4.5 Integer Exception Register (XER)

The XER bit assignments are shown in Figure 2-8.

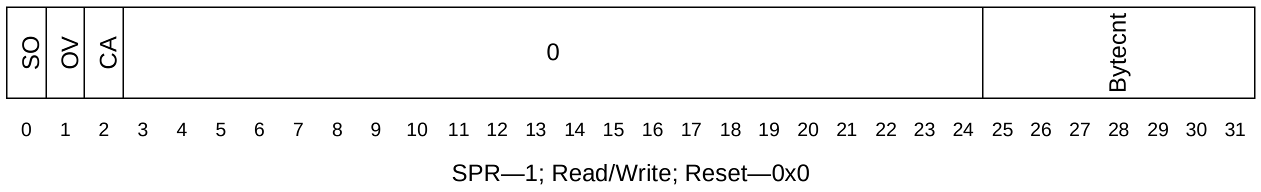


Figure 2-8. Integer Exception Register (XER)

The XER fields are defined in Table 2-5.

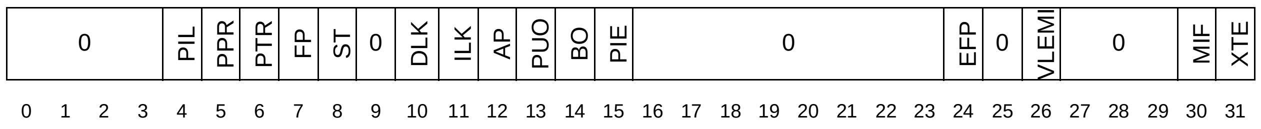
Table 2-5. XER Field Descriptions

Bits	Name	Description
0 (32)	SO	Summary Overflow (per 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

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

2.4.6 Exception Syndrome Register

The Exception Syndrome Register (ESR) provides a syndrome to differentiate between exceptions that can generate the same interrupt type. e200 adds some implementation-specific bits to this register, as seen in Figure 2-9.



SPR—62; Read/Write; Reset—0x0

Figure 2-9. Exception Syndrome Register (ESR)

The ESR fields are defined in Table 2-6.

Table 2-6. ESR Field Descriptions

Bit(s)	Name	Description	Associated Interrupt Type
0:3 (32:35)	—	Allocated[1]	—
4 (36)	PIL	Illegal Instruction exception	Program
5 (37)	PPR	Privileged Instruction exception	Program
6 (38)	PTR	Trap exception	Program
7 (39)	FP	Floating-point operation	Alignment Data Storage Data TLB Program

Table 2-6. ESR Field Descriptions (continued)

Bit(s)	Name	Description	Associated Interrupt Type
8 (40)	ST	Store operation	Alignment Data Storage Data TLB
9 (41)	—	Reserved[1]	—
10 (42)	DLK[2]	Data Cache Locking	Data Storage
11 (43)	ILK[2]	Instruction Cache Locking	Data Storage
12 (44)	AP	Auxiliary Processor operation (Currently unused in e200)	Alignment Data Storage Data TLB Program
13 (45)	PUO	Unimplemented Operation exception	Program
14 (46)	BO	Byte Ordering exception Mismatched Instruction Storage exception	Data Storage Instruction Storage
15 (47)	PIE	Program Imprecise exception (Reserved)	Currently unused in e200
16:23 (48:55)	—	Reserved[1]	—
24 (56)	EFP	Embedded Floating-point APU Operation	Allocated, not set by hardware
25 (57)	—	Allocated[1]	—
26 (58)	VLEMI	VLE Mode Instruction	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)	XTE	External Termination Error (Precise)	Data Storage Instruction Storage

[1] These bits are not implemented and should be written with zero for future compatibility.
[2] e200z1.

2.4.6.1 Power Architecture VLE Mode Instruction Syndrome

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

2.4.6.2 Misaligned Instruction Fetch Syndrome

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

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

2.4.6.3 Precise External Termination Error Syndrome

The ESR[XTE] bit is provided to indicate that a precise external termination error DSI or ISI interrupt was caused by an instruction. This syndrome bit is set on an external termination error exception that is reported in a precise manner via a DSI or ISI as opposed to a machine check.

2.4.7 Machine Check Syndrome Register (MCSR)

When the core complex takes a machine check interrupt, it updates the Machine Check Syndrome register (MCSR) to differentiate between machine check conditions. The MCSR is shown in Figure 2-10.

e200z1	MCP	0	CP_PERR	CPERR	EXCP_ERR	0	BUS_IRERR	BUS_DRERR	BUS_WRERR	0
	0	1	2	3	4	5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26	27	28	29	30 31

SPR—572; Read/Write; Reset —0x0

Figure 2-10. Machine Check Syndrome Register (MCSR)

Table 2-7 describes MCSR fields. The MCSR indicates the source of a machine check condition is recoverable. When a syndrome bit in the MCSR is set, the core complex asserts *p_mcp_out* for system information.

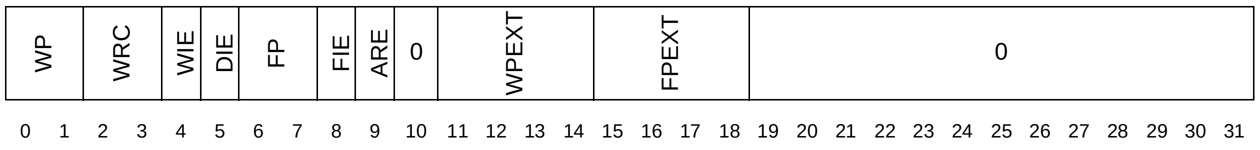
Table 2-7. Machine Check Syndrome Register (MCSR)

Bit	Name	Description
0 (32)	MCP	Machine check input pin
1 (33)	—	Reserved, should be cleared.
2 (34)	CP_PERR[1]	Cache push parity error
3 (35)	CPERR[1]	Cache parity error
4 (36)	EXCP_ERR	ISI, ITLB, or Bus Error on first instruction fetch for an exception handler
5:26 (37:58)	—	Reserved. Should be cleared.
27 (59)	BUS_IRERR	Read bus error on Instruction fetch
28 (60)	BUS_DRERR	Read bus error on data load
29 (61)	BUS_WRERR	Write bus error on buffered store
30:31 (62:63)	—	Reserved, should be cleared.

[1] Unused on e200z1.

2.4.8 Timer Control Register (TCR)

The timer control register (TCR) provides control information for the CPU timer facilities. The TCR[WRC] field functions are defined to be implementation-dependent and are described below. In addition, the e200 core implements two fields not specified in Book E, TCR[WPEXT] and TCR[FPEXT]. The TCR is shown in Figure 2-11.



SPR—340; Read/Write; Reset—0x0

Figure 2-11. Timer Control Register (TCR)

The TCR fields are defined in Table 2-8

Table 2-8. Timer Control Register Field Descriptions

Bits	Name	Description
0:1 (32:33)	WP	Watchdog Timer Period When concatenated with WPEXT, specifies one of 64 bit locations of the time base used to signal a watchdog timer exception on a transition from 0 to 1. TCRwpext[0–3],TCRwp[0–1] == 6’b000000 selects TBU[0] TCRwpext[0–3],TCRwp[0–1] == 6’b111111 selects TBL[31]
2:3 (34:35)	WRC	Watchdog Timer Reset Control 00 No Watchdog Timer reset will occur 01 Force processor checkstop on second time-out of Watchdog Timer 10 Assert processor reset output (*p_resetout_b*) on second time-out of Watchdog Timer 11 Reserved 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]

[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. The TSR[WRS] field is defined to be implementation-dependent and is described below. The TSR is shown in Figure 2-12.

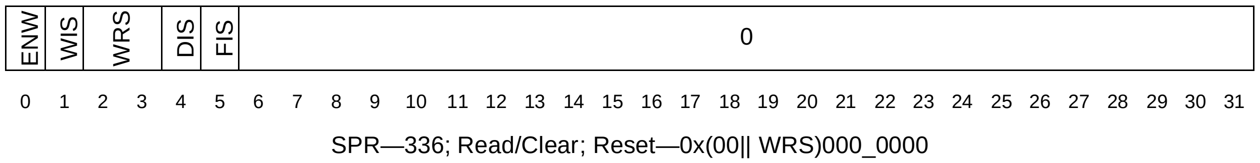


Figure 2-12. Timer Status Register (TSR)

The TSR fields are defined in Table 2-9.

Table 2-9. Timer Status Register Field Descriptions

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

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

NOTE

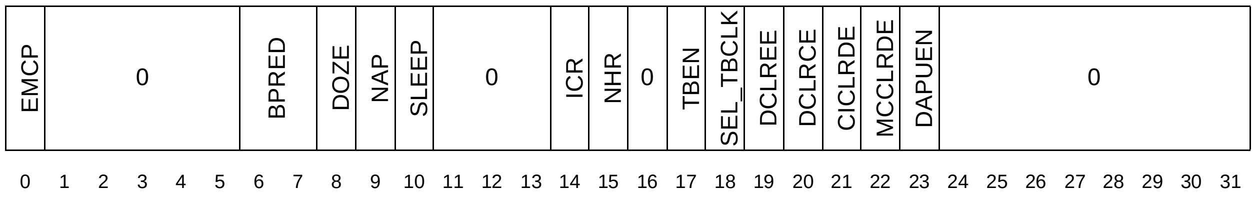
The Timer Status Register can be read using **mfspr** *RT,TSR*. The Timer Status Register cannot be directly written to. Instead, bits in the Timer Status Register corresponding to 1 bits in GPR(RS) can be cleared using **mtspr** *TSR,RS*.

2.4.10 Debug Registers

The Debug facility registers are described in Chapter 9, “Debug Support.”

2.4.11 Hardware Implementation Dependent Register 0 (HID0)

The HID0 register is an e200 implementation dependent register used for various configuration and control functions.The HID0 register is shown in Figure 2-13.



SPR—1008; Read/Write; Reset—0x0

Figure 2-13. Hardware Implementation Dependent Register 0 (HID0)

The HID0 fields are defined in Table 2-10.

Table 2-10. Hardware Implementation Dependent Register 0

Bits	Name	Description
0	EMCP	Enable machine check pin (*p_mcp_b*) 0 *p_mcp_b* pin is disabled. 1 *p_mcp_b* pin is enabled. If MSR[ME] = 0, asserting *p_mcp_b* causes checkstop. If MSR[ME] = 1, asserting *p_mcp_b* causes a machine check interrupt. The primary purpose of this bit is to mask out further machine check exceptions caused by assertion of *p_mcp_b*.
1:5	—	Reserved[1]
6:7	BPRED	Branch Prediction (Acceleration) Control 00 - Branch acceleration is enabled. 01 - Branch acceleration is disabled for backward branches. 10 - Branch acceleration is disabled for forward branches. 11 - Branch acceleration is disabled for both branch directions. This field controls instruction buffer lookahead for branch acceleration. Note that for branches with "AA' = '1', the MSB of the displacement field is still used to indicate forward/backward, even though the branch is absolute. This field is used in conjunction with the BUCSR.
8	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	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	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	—	Reserved[1]

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

Bits	Name	Description
14	ICR	Interrupt Inputs Clear Reservation 0 External Input and Critical Input Interrupts do not affect reservation status 1 External Input and Critical Input Interrupts clear an outstanding reservation
15	NHR	Not hardware reset 0 Indicates to a reset exception handler that a reset occurred if software had previously set this bit 1 Indicates to a reset exception handler that no reset occurred if software had previously set this bit Provided for software use - set anytime by software, cleared by reset.
16	—	Reserved[1]
17	TBEN	TimeBase Enable 0 TimeBase is disabled 1 TimeBase is enabled
18	Reserved	Reserved
19	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	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	CICLRDE	Critical Interrupt Clears MSR[DE] 0 MSR[DE] unaffected by Critical class interrupt 1 MSR[DE] cleared by Critical class interrupt This bit controls whether certain Critical interrupts (Critical Input, Watchdog Timer) force Debug interrupts to be disabled, or whether they remain unaffected. Machine Check interrupts have a separate control bit. Note that if Critical Interrupt Debug events are enabled (DBCR0[CIRPT] set (which should only be done when the Debug APU is enabled), and MSR[DE] is set at the time of a (Critical Input, Watchdog Timer) Critical interrupt, a debug event is generated after the Critical Interrupt Handler has been fetched, and the Debug handler is executed first. In this case, DSRR0[DE] is cleared, such that after returning from the debug handler, the Critical interrupt handler is not run with MSR[DE] enabled.
22	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. Note that if Critical Interrupt Debug events are enabled (DBCR0[CIRPT] set (which should only be done when the Debug APU is enabled), and MSR[DE] is set at the time of a Machine Check interrupt, a debug event is generated after the Machine Check interrupt handler has been fetched, and the Debug handler is executed first. In this case, DSRR0[DE] is cleared, such that after returning from the Debug handler, the Machine Check handler is not run with MSR[DE] enabled.

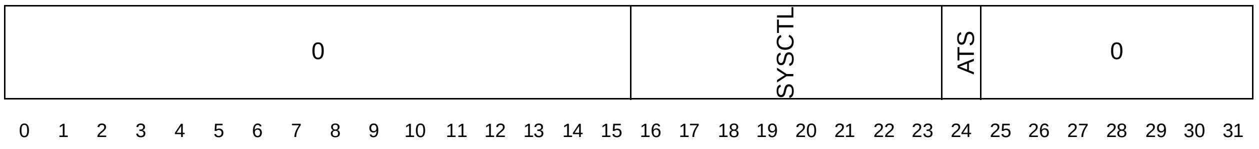
Table 2-10. Hardware Implementation Dependent Register 0 (continued)

Bits	Name	Description
23	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** and **se_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** and **se_rfci** instruction is used for returning from a debug interrupt, and the **rfdi** and **se_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.
24:31	—	Reserved[1]

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

2.4.12 Hardware Implementation Dependent Register 1 (HID1)

The HID1 register is used for bus configuration and system control. HID1 is shown in Figure 2-14.



SPR—1009; Read/Write; Reset—0x0

Figure 2-14. Hardware Implementation Dependent Register 1 (HID1)

The HID1 fields are defined in Table 2-11.

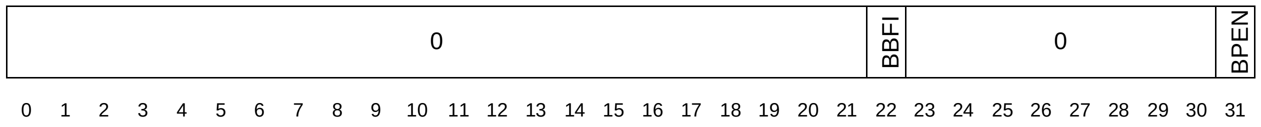
Table 2-11. Hardware Implementation Dependent Register 1

Bits	Name	Description
0:15	—	Reserved[1]
16:23	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	ATS	Atomic status (read-only) Indicates state of the reservation bit in the load/store unit. See Section 3.6, "Memory Synchronization and Reservation Instructions," for more detail.
25:31	—	Reserved[1]

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

2.4.13 Branch Unit Control and Status Register (BUCSR)

The BUCSR register is used for general control and status of the branch target buffer (BTB). BUCSR is shown in Figure 2-15.



SPR—1013; Read/Write; Reset—0x0

Figure 2-15. Branch Unit Control and Status Register (BUCSR)

The BUCSR fields are defined in Table 2-12.

Table 2-12. Branch Unit Control and Status Register

Bits	Name	Description
0:21 [32:53]	—	Reserved[1]
22 [54]	BBFI	Branch target buffer flash invalidate. When written to a ‘1’, BBFI flash clears the valid bit of all entries in the branch buffer; clearing occurs regardless of the value of the enable bit (BPEN). Note: BBFI is always read as 0.
25:30 [55:62]	—	Reserved[1]
31 [63]	BPEN	Branch target buffer 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 are generated from the BTB, and no new entries are allocated. Entries are not automatically invalidated when BPEN is cleared, the BBFI bit controls entry invalidation.

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

2.4.14 L1 Cache Configuration Register (L1CFG0)

The L1CFG0 register provides configuration information for an L1 cache supplied with this version of the e200 CPU core. For e200z1 reads of this register return a value of all zeros.

2.4.15 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 6, “Memory Management Unit.”

2.4.16 MMU Configuration Register (MMUCFG)

The MMUCFG register provides configuration information for the MMU supplied with this version of the e200 CPU core. A description of the MMUCFG register can be found in Chapter 6, “Memory Management Unit.”

2.4.17 TLB Configuration Registers (TLB0CFG, TLB1CFG)

The TLB0CFG and TLB1CFG registers provide configuration information for the optional MMU TLBs supplied with this version of the e200 CPU core. A description of these registers can be found in Chapter 6, "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 core is not in supervisor mode (MSR[PR] = 1), then a privilege exception is generated.

Table 2-13. System Response to Invalid SPR Reference

SPR Address Bit 5	Mode	MSR[PR]	Response
0	—	—	Illegal exception
1	supervisor	0	Illegal exception
1	user	1	Privilege exception

References to the SPRs associated with an optional unit (Cache, MMU, EFPU) when the unit is not present are treated as references to an invalid SPR unless otherwise defined.

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 Power Architecture Book E. Software requirements for synchronization before/after accessing these registers are shown in Table 2-14. The notation CSI in the table refers to a Context Synchronizing instruction which include **sc**, **isync**, **rfi**, **rfci**, and **rfdi**.

Table 2-14. Additional Synchronization Requirements for SPRs

Context Altering Event or Instruction		Required Before	Required After	Notes
mfspr				
DBCNT	Debug Counter register	**msync**	none	1
DBSR	Debug Status Register	**msync**	none	—
HID0	Hardware implementation dependent reg 0	none	none	—

Table 2-14. Additional Synchronization Requirements for SPRs (continued)

Context Altering Event or Instruction		Required Before	Required After	Notes
HID1	Hardware implementation dependent reg 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	—
DBCR0	Debug Control Register 0	none	CSI	—
DBCR1	Debug Control Register 1	none	CSI	—
DBCR2	Debug Control Register 2	none	CSI	—
DBCR3	Debug control register 3	none	CSI	—
DBSR	Debug Status Register	**msync**	none	—
HID0	Hardware implementation dependent reg 0	CSI	CSI	—
HID1	Hardware implementation dependent reg 1	none	CSI	—
MASx	MMU MAS registers	none	CSI	—
MMUCSR	MMU control and status register 0	CSI	CSI	—
PID	PID0 register	none	CSI	—

Note:

1. Not required if counter is not currently enabled

2.5.3 Special Purpose Register Summary

Power Architecture Book E and implementation-specific SPRs for the e200 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). 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 Table 2-15.

Table 2-15. 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

Table 2-15. Special Purpose Registers (continued)

Mnemonic	Name	SPR Number	Access	Privileged	Zen Specific
DBCR0	Debug Control Register 0	308	R/W	Yes	No
DBCR1	Debug Control Register 1	309	R/W	Yes	No
DBCR2	Debug Control Register 2	310	R/W	Yes	No
DBCR3	Debug control register 3	561	R/W	Yes	Yes
DBSR	Debug Status Register	304	Read/Clear[1]	Yes	No
DEAR	Data Exception Address Register	61	R/W	Yes	No
DEC	Decrementer	22	R/W	Yes	No
DECAR	Decrementer Auto-Reload	54	R/W	Yes	No
DSRR0	Debug save/restore register 0	574	R/W	Yes	Yes
DSRR1	Debug save/restore register 1	575	R/W	Yes	Yes
ESR	Exception Syndrome Register	62	R/W	Yes	No
HID0	Hardware implementation dependent reg 0	1008	R/W	Yes	Yes
HID1	Hardware implementation dependent reg 1	1009	R/W	Yes	Yes
IAC1	Instruction Address Compare 1	312	R/W	Yes	No
IAC2	Instruction Address Compare 2	313	R/W	Yes	No
IAC3	Instruction Address Compare 3	314	R/W	Yes	No
IAC4	Instruction Address Compare 4	315	R/W	Yes	No
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
MAS0	MMU assist register 0	624	R/W	Yes	Yes
MAS1	MMU assist register 1	625	R/W	Yes	Yes
MAS2	MMU assist register 2	626	R/W	Yes	Yes
MAS3	MMU assist register 3	627	R/W	Yes	Yes
MAS4	MMU assist register 4	628	R/W	Yes	Yes
MAS6	MMU assist register 6	630	R/W	Yes	Yes
MCSR	Machine Check Syndrome Register	572	R/W	Yes	Yes
MMUCFG	MMU configuration register	1015	Read-only	Yes	Yes
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	Read-only	Yes	No
PVR	Processor Version Register	287	Read-only	Yes	No

Table 2-15. Special Purpose Registers (continued)

Mnemonic	Name	SPR Number	Access	Privileged	Zen Specific
SPRG0	SPR General 0	272	R/W	Yes	No
SPRG1	SPR General 1	273	R/W	Yes	No
SPRG2	SPR General 2	274	R/W	Yes	No
SPRG3	SPR General 3	275	R/W	Yes	No
SPRG4	SPR General 4	260	Read-only	No	No
		276	R/W	Yes	No
SPRG5	SPR General 5	261	Read-only	No	No
		277	R/W	Yes	No
SPRG6	SPR General 6	262	Read-only	No	No
		278	R/W	Yes	No
SPRG7	SPR General 7	263	Read-only	No	No
		279	R/W	Yes	No
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[2]	Yes	No
USPRG0	User SPR General 0	256	R/W	No	No
XER	Integer Exception Register	1	R/W	No	No

[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 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*.

2.5.4 Reset Settings

Table 2-16 shows the state of the Power Architecture Book E architected registers and other optional resources immediately following a system reset.

Table 2-16. Reset Settings for e200 Resources

Resource	System Reset Setting
Program Counter	p_rstbase[0:29] || 2'b00
GPRs	Unaffected[1]
CR	Unaffected[1]
BUCSR	0x0000_0000
CSRR0	Unaffected[1]
CSRR1	Unaffected[1]
CTR	Unaffected[1]
DAC1	0x0000_0000
DAC2	0x0000_0000
DBCNT	Unaffected[1]
DBCR0	0x0000_0000
DBCR1	0x0000_0000
DBCR2	0x0000_0000
DBCR3	0x0000_0000
DBSR	0x1000_0000
DEAR	Unaffected[1]
DEC	Unaffected[1]
DECAR	Unaffected[1]
DSRR0	Unaffected[1]
DSRR1	Unaffected[1]
ESR	0x0000_0000
HID0	0x0000_0000
HID1	0x0000_0000
IAC1	0x0000_0000
IAC2	0x0000_0000
IAC3	0x0000_0000
IAC4	0x0000_0000
IVPR	Unaffected[1]
LR	Unaffected[1]
L1CFG0[2]	—

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

Resource	System Reset Setting
MCSR	0x0000_0000
MMUCFG[2]	—
MSR	0x0000_0000
PID0	0x0000_0000
PIR[2]	—
PVR[2]	—
SPRG0	Unaffected[1]
SPRG1	Unaffected[1]
SPRG2	Unaffected[1]
SPRG3	Unaffected[1]
SPRG4	Unaffected[1]
SPRG5	Unaffected[1]
SPRG6	Unaffected[1]
SPRG7	Unaffected[1]
USPRG0	Unaffected[1]
SRR0	Unaffected[1]
SRR1	Unaffected[1]
SVR[2]	—
TBL	Unaffected[1]
TBU	Unaffected[1]
TCR	0x0000_0000
TSR	Undefined on POR, 0x(00|| WRS)000_0000
XER	0x0000_0000

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

2 Read-only register

Chapter 3
Instruction Model

This chapter provides additional information about the Power Architecture Book E as it relates specifically to e200.

The e200 cores are a 32-bit implementation of the Power Architecture Book E as defined in Power Architecture Book E Specification v 2.0. This architecture specification includes a recognition that different processor implementations may require clarifications, extensions or deviations from the architectural descriptions.

3.1 Unsupported Instructions and Instruction Forms

Because e200 is a 32-bit Power Architecture Book E core, all of the instructions defined for 64-bit implementations of the Power Architecture Book E architecture are illegal on e200. See Appendix A of Power Architecture Book E Specification for more information on 64-bit instructions. e200 takes an illegal instruction exception type program interrupt upon encountering a 64-bit Power Architecture Book E instruction.

The e200 core does not support the instructions listed in Table 3-1. An unimplemented instruction or FP unavailable exception is generated if the processor attempts to execute one of these instructions.

Table 3-1. List of Unsupported Instructions

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

3.2 Optionally Supported Instructions and Instruction Forms

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

Table 3-2. List of Optionally Supported Instructions

Type/Name	Mnemonics	Unit
Cache Management Instructions[1]	**dcba, dcbf, dcbi, dcbt, dcbtst, dcbst, dcbz, icbi, icbt**	Data Cache/ Unified Cache Instruction Cache/Unified Cache
Cache Locking Instructions[2]	**dcbtls, dcbtstls, dcblc, icbtls, icblc**	Data Cache/ Unified Cache Instruction Cache/Unified Cache

Table 3-2. List of Optionally Supported Instructions (continued)

Type/Name	Mnemonics	Unit
TLB Management Instructions[3]	**tlbivax, tlbre, tlbsx, tlbsync, tlbwe**	TLB
DCR Management[4]	**mfdcr, mtdcr**	DCR

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

2 These instructions are not supported and are treated as no-ops.

3 These instructions are supported by e200z1

4 These instructions are supported by e200z1

3.3 Implementation Specific Instructions

Several Power Architecture Book E defined instructions are implementation specific. Table 3-3 summarizes the e200 implementation-specific instructions.

Table 3-3. Implementation-Specific Instruction Summary

Mnemonic	Implementation Details
mfapidi	Unimplemented instructions
mfdcrx, mtdcrx	
stwcx.	Address match with prior **lwarx** not req'd for store to be performed
mfdcr, mtdcr[1]	Optionally supported instructions
tlbivax	
tlbre	
tlbsx	
tlbsync	
tlbwe	

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

3.4 Book E Instruction Extensions

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

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

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

3.5 Memory Access Alignment Support

The e200 core provides hardware support for unaligned memory accesses; however, there is a performance degradation for accesses that cross a 32-bit (4-byte) boundary. For these cases, the throughput of the load/store unit is degraded to 1 misaligned load every 2 cycles. Stores that are misaligned across a 32-bit (4-byte) boundary can be translated at a rate of 2 cycles per store. Frequent use of unaligned memory accesses result in an impact on performance.

NOTE

Accesses that cross a 32-bit boundary may be restarted. A misaligned access which crosses a page boundary is restarted in its entirety in the event of a TLB miss of the second portion of the access. This may result in the first portion being accessed twice.

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

3.6 Memory Synchronization and Reservation Instructions

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

On the e200 core, the **mbar** instruction behaves identically to the **msync** instruction. The **mbar** instruction MO field is ignored by the e200 core.

The e200 core implements the **lwarx** and **stwcx.** instructions as described in Book E. If the EA is not a multiple of 4 for either instruction, an alignment interrupt is invoked. e200 allows **lwarx** and **stwcx.** 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 Power Architecture Book E, the e200 core does not require that for a **stwcx.** instruction to succeed, the EA of the **stwcx.** instruction must be to the same reservation granule as the EA of a preceding **lwarx** instruction. Reservation granularity is implementation-dependent. The e200 core does not define a reservation granule explicitly; reservation granularity is defined by external logic. When no external logic is provided, the e200 core performs no address comparison checking, thus the effective implementation granularity is “null”.

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

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

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

The e200 core treats **lwarx** and **stwcx.** accesses as though they were both cache inhibited and guarded, regardless of page attributes. The e200 core provides the input signal *p_hresp*[2:0], which is sampled at termination of a **stwcx.** store transfer to allow an external agent or mechanism to indicate that the **stwcx.** instruction has failed to update memory, even though a reservation existed for the store at the time it was issued. This is not considered an error, and causes the condition codes for the **stwcx.** instruction to be written as if a reservation did not exist for the **stwcx.** instruction. In addition, any outstanding reservation is cleared.

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

3.7 Branch Prediction

In the e200z1, the instruction fetching mechanism uses a branch target buffer to detect branch instructions early. This branch instruction lookahead scheme allows branch targets to be fetched early, thereby hiding some taken branch bubbles.

3.8 Interruption of Instructions by Interrupt Requests

In general, the e200 core samples pending external input and critical input interrupt requests at instruction boundaries. However, in order to reduce interrupt latency, long running instructions may be interrupted prior to completion. Instructions in this class include divides (**divw[uo][.]**), load multiple word and store multiple word. When interrupted prior to completion, the value saved in SRR0/CSRR0 is the address of the interrupted instruction. The instruction is restarted from the beginning after returning from the interrupt handler.

3.9 New e200 Instructions

The e200 core implements the Freescale EIS **isel** APU as described below which extends the Power Architecture Book E instruction set. The e200 **wait** instruction implements a wait for interrupt function and is described below. The e200 **rfdi** or **se_rfdi** instruction returns from a Debug interrupt and is also described below.

3.9.1 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

3.9.2 Debug APU

e200 implements the EIS Debug APU, as documented in the *EREF: A Programmer's Reference Manual for Freescale Book E Processors,* 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** or **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 forms and definition are as follows.

rfdi rfdi

Return From Debug Interrupt

rfdi

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

se_rfdi se_rfdi

Return From Debug Interrupt

se_rfdi

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

```
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 (such as 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.

3.9.3 WAIT APU

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

wait wait

Wait for Interrupt

wait

0–5	6–10	11–15	16–20	21–30	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 points to the next sequential instruction. The saved value in xSRR0 when the processor re-initiates activity points 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 is negated after leaving the "waiting" state.

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

3.10 Unimplemented SPRs and Read-Only SPRs

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

3.11 Invalid Forms of Instructions

3.11.1 Load and Store with Update Instructions

Power Architecture 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 e200 core performs the instruction and update the register with the load data. In addition, if RA=0 for any load or store with update instruction, the e200 core updates RA (GPR0).

3.11.2 Load Multiple Word (lmw, e_lmw) Instruction

Power Architecture 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 e200, invalid forms of the **lmw** or **e_lmw** instruction is 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 in is done using the architectural value of RA which existed when beginning execution of this **lmw** or **e_lmw** instruction. RA is overwritten with a value fetched from memory as if it had not been the base register. Note that if the instruction is interrupted and restarted, the base address may be different if RA has been overwritten.

- Case 2: *RA=0 and RT=0.* In this case, address generation for all loads to register targets RT=0 to RT=31 is done substituting the value of 0 for the RA operand.

3.11.3 Branch Conditional to Count Register Instructions

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

3.11.4 Instructions with Reserved Fields Non-Zero

Power Architecture 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, e200 ignores the value of the reserved field (bit 31) in X-form integer load and store instructions. e200 ignores the value of the reserved ‘z’ bits in the BO field of branch instructions. For all other instructions, e200 generates an illegal instruction exception if a reserved field is non-zero.

3.12 Instruction Summary

Table 3-4 and Table 3-5 list all 32-bit instructions in Power Architecture Book E, as well as certain e200 specific instructions, sorted by mnemonic. Format, Opcode, Mnemonic, Instruction name, and page number are included in the Book E: Enhanced PowerPC Architecture v0.99. Implementation dependent instructions are noted with a footnote. Instructions which are optionally supported (when an optional function is added to the base core) are shown with shaded entries.

Note that the following specific APU are not included in the table below:

- VLE APU

3.12.1 Instruction Index Sorted by Mnemonic

Table 3-4. 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
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

Legend:

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

Table 3-4. Instructions Sorted by Mnemonic (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
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
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

Legend:

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

Table 3-4. Instructions Sorted by Mnemonic (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
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	00001 10110 /	dcbst	Data Cache Block Store	245
X	011111	01000 10110 /	dcbt	Data Cache Block Touch	246
X	011111	00111 10110 /	dcbtst	Data Cache Block Touch for Store	247
X	011111	11111 10110 /	dcbz	Data Cache Block set to Zero	248
X	011111	01111 01011 0	divw	Divide Word	251
X	011111	01111 01011 1	divw.	Divide Word & record CR	251
X	011111	11111 01011 0	divwo	Divide Word & record OV	251
X	011111	11111 01011 1	divwo.	Divide Word & record OV & CR	251
X	011111	01110 01011 0	divwu	Divide Word Unsigned	252
X	011111	01110 01011 1	divwu.	Divide Word Unsigned & record CR	252
X	011111	11110 01011 0	divwuo	Divide Word Unsigned & record OV	252
X	011111	11110 01011 1	divwuo.	Divide Word Unsigned & record OV & CR	252
X	011111	01000 11100 0	eqv	Equivalent	253
X	011111	01000 11100 1	eqv.	Equivalent & record CR	253
X	011111	11101 11010 0	extsb	Extend Sign Byte	254
X	011111	11101 11010 1	extsb.	Extend Sign Byte & record CR	254
X	011111	11100 11010 0	extsh	Extend Sign Halfword	254
X	011111	11100 11010 1	extsh.	Extend Sign Halfword & record CR	254
X	111111	01000 01000 0	[1]	—	255
X	111111	01000 01000 1	[1]	—	255
A	111111	----- 10101 0	[1]	—	256
A	111111	----- 10101 1	[1]	—	256
A	111011	----- 10101 0	[1]	—	256

Legend:

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

/ Reserved bit, invalid instruction form if encoded as 1

? Allocated for implementation-dependent use. See User’ Manual for the implementation

Table 3-4. Instructions Sorted by Mnemonic (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
A	111011	----- 10101 1	[1]	—	256
X	111111	11010 01110 /	[1]	—	257
X	111111	00001 00000 /	[1]	—	259
X	111111	00000 00000 /	[1]	—	259
X	111111	11001 01110 /	[1]	—	260
X	111111	11001 01111 /	[1]	—	260
X	111111	00000 01110 0	[1]	—	262
X	111111	00000 01110 1	[1]	—	262
X	111111	00000 01111 0	[1]	—	262
X	111111	00000 01111 1	[1]	—	262
A	111111	----- 10010 0	[1]	—	264
A	111111	----- 10010 1	[1]	—	264
A	111011	----- 10010 0	[1]	—	264
A	111011	----- 10010 1	[1]	—	264
A	111111	----- 11101 0	[1]	—	265
A	111111	----- 11101 1	[1]	—	265
A	111011	----- 11101 0	[1]	—	265
A	111011	----- 11101 1	[1]	—	265
X	111111	00010 01000 0	[1]	—	266
X	111111	00010 01000 1	[1]	—	266
A	111111	----- 11100 0	[1]	—	267
A	111111	----- 11100 1	[1]	—	267
A	111011	----- 11100 0	[1]	—	267
A	111011	----- 11100 1	[1]	—	267
A	111111	----- 11001 0	[1]	—	268
A	111111	----- 11001 1	[1]	—	268
A	111011	----- 11001 0	[1]	—	268
A	111011	----- 11001 1	[1]	—	268

Legend:

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

Table 3-4. 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 0	[1]	—	269
X	111111	00100 01000 1	[1]	—	269
X	111111	00001 01000 0	[1]	—	269
X	111111	00001 01000 1	[1]	—	269
A	111111	----- 11111 0	[1]	—	270
A	111111	----- 11111 1	[1]	—	270
A	111011	----- 11111 0	[1]	—	270
A	111011	----- 11111 1	[1]	—	270
A	111111	----- 11110 0	[1]	—	271
A	111111	----- 11110 1	[1]	—	271
A	111011	----- 11110 0	[1]	—	271
A	111011	----- 11110 1	[1]	—	271
A	111011	----- 11000 0	[1]	—	272
A	111011	----- 11000 1	[1]	—	272
X	111111	00000 01100 0	[1]	—	273
X	111111	00000 01100 1	[1]	—	273
A	111111	----- 11010 0	[1]	—	276
A	111111	----- 11010 1	[1]	—	276
A	111111	----- 10111 0	[1]	—	277
A	111111	----- 10111 1	[1]	—	277
A	111111	----- 10110 0	[1]	—	278
A	111111	----- 10110 1	[1]	—	278
A	111011	----- 10110 0	[1]	—	278
A	111011	----- 10110 1	[1]	—	278
A	111111	----- 10100 0	[1]	—	279
A	111111	----- 10100 1	[1]	—	279
A	111011	----- 10100 0	[1]	—	279
A	111011	----- 10100 1	[1]	—	279

Legend:

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

Table 3-4. Instructions Sorted by Mnemonic (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary ($Inst_{0:5}$)	Extended ($Inst_{21:31}$)			
X	011111	11110 10110 /	icbi	Instruction Cache Block Invalidate	280
X	011111	00000 10110 /	icbt	Instruction Cache Block Touch	281
??	011111	----- 01111 /	isel[2]	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	----- ----- -	[1]	—	286
D	110011	----- ----- -	[1]	—	286
X	011111	10011 10111 /	[1]	—	286
X	011111	10010 10111 /	[1]	—	286
D	110000	----- ----- -	[1]	—	287
D	110001	----- ----- -	[1]	—	287
X	011111	10001 10111 /	[1]	—	287
X	011111	10000 10111 /	[1]	—	287
D	101010	----- ----- -	lha	Load Halfword Algebraic	288
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 /	[3]	—	292
X	011111	10000 10101 /	[3]	—	292

Legend:

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

Table 3-4. 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 10100 /	lwarx[4]	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[4]	Memory Barrier	298
XL	010011	00000 00000 /	mcrf	Move Condition Register Field	299
X	111111	00010 00000 /	[1]	—	300
X	011111	10000 00000 /	mcrxr	Move to Condition Register from XER	300
X	011111	01000 10011 /	[3]	—	301
X	011111	00000 10011 /	mfcr	Move From Condition Register	301
XFX	011111	01010 00011 /	mfdcr[3]	Move From Device Control Register	302
X	011111	01000 00011 /	[3]	—	302
X	111111	10010 00111 0	[1]	—	303
X	111111	10010 00111 1	[1]	—	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[4]	Memory Synchronize	305
XFX	011111	00100 10000 /	mtcrf	Move To Condition Register Fields	306
XFX	011111	01110 00011 /	mtdcr[3]	Move To Device Control Register	307
X	011111	01100 00011 /	[3]	—	307
X	111111	00010 00110 0	[1]	—	308
X	111111	00010 00110 1	[1]	—	308
X	111111	00001 00110 0	[1]	—	308
X	111111	00001 00110 1	[1]	—	308
XFL	111111	10110 00111 0	[1]	—	309
XFL	111111	10110 00111 1	[1]	—	309

Legend:

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

Table 3-4. 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 00110 0	[1]	—	310
X	111111	00100 00110 1	[1]	—	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
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

Legend:

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

Table 3-4. 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 00111 /	rfdi[5]	Return From Debug Interrupt	—
XL	010011	00001 10010 /	rfi	Return From Interrupt	322
M	010100	----- ----- 0	rlwimi	Rotate Left Word Immed then Mask Insert	327
M	010100	----- ----- 1	rlwimi.	Rotate Left Word Immed then Mask Insert & record CR	327
M	010101	----- ----- 0	rlwinm	Rotate Left Word Immed then AND with Mask	328
M	010101	----- ----- 1	rlwinm.	Rotate Left Word Immed 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	----- ----- -	[1]	—	340
D	110111	----- ----- -	[1]	—	340
X	011111	10111 10111 /	[1]	—	340
X	011111	10110 10111 /	[1]	—	340
X	011111	11110 10111 /	[1]	—	341
D	110100	----- ----- -	[1]	—	342
D	110101	----- ----- -	[1]	—	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 3-4. Instructions Sorted by Mnemonic (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary (Inst$_{0:5}$)	Extended (Inst$_{21:31}$)			
X	011111	10101 10111 /	[1]	—	342
X	011111	10100 10111 /	[1]	—	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 /	[3]	—	346
X	011111	10100 10101 /	[3]	—	346
D	100100	----- ----- -	stw	Store Word	347
X	011111	10100 10110 /	stwbrx	Store Word Byte-Reverse Indexed	348
X	011111	00100 10110 1	stwcx.[4]	Store Word Conditional Indexed & record CR	349
D	100101	----- ----- -	stwu	Store Word with Update	347
X	011111	00101 10111 /	stwux	Store Word with Update Indexed	347
X	011111	00100 10111 /	stwx	Store Word Indexed	347
X	011111	00001 01000 0	subf	Subtract From	351
X	011111	00001 01000 1	subf.	Subtract From & record CR	351
X	011111	00000 01000 0	subfc	Subtract From Carrying	352
X	011111	00000 01000 1	subfc.	Subtract From Carrying & record CR	352
X	011111	10000 01000 0	subfco	Subtract From Carrying & record OV	352
X	011111	10000 01000 1	subfco.	Subtract From Carrying & record OV & CR	352
X	011111	00100 01000 0	subfe	Subtract From Extended with CA	353
X	011111	00100 01000 1	subfe.	Subtract From Extended with CA & record CR	353
X	011111	10100 01000 0	subfeo	Subtract From Extended with CA & record OV	353
X	011111	10100 01000 1	subfeo.	Subtract From Extended with CA & record OV & CR	353
D	001000	----- ----- -	subfic	Subtract From Immediate Carrying	354
X	011111	00111 01000 0	subfme	Subtract From Minus One Extended with CA	355

Legend:

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

Table 3-4. Instructions Sorted by Mnemonic (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary (Inst$_{0:5}$)	Extended (Inst$_{21:31}$)			
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
X	011111	00000 00100 /	tw	Trap Word	363
D	000011	----- ----- -	twi	Trap Word Immediate	363
X	011111	00100 00011 /	wrtee	Write External Enable	364
X	011111	00101 00011 /	wrteei	Write External Enable Immediate	364
X	011111	01001 11100 0	xor	XOR	365
X	011111	01001 11100 1	xor.	XOR & record CR	365
D	011010	----- ----- -	xori	XOR Immediate	365
D	011011	----- ----- -	xoris	XOR Immediate Shifted	365

Legend:

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

[1] Attempted execution causes an unimplemented exception if MSR[FP]=1, or a FP Unavailable exception if MSR[FP]=0.

[2] EIS **isel** APU, refer to Section 3.9.1, "ISEL APU" on page 3-4. EIS specifications can be found in the *EREF: A Programmer's Reference Manual for Freescale Book E Processors*

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

[4] See Section 3.6, "Memory Synchronization and Reservation Instructions" on page 3-3

[5] See Section 3.9.2, "Debug APU" on page 3-5

3.12.2 Instruction Index Sorted by Opcode

Table 3-5. Instructions Sorted by Opcode

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary (Inst0:5)	Extended (Inst21:31)			
D	000011	----- ----- -	twi	Trap Word Immediate	363
D	000111	----- ----- -	mulli	Multiply Low Immediate	315
D	001000	----- ----- -	subfic	Subtract From Immediate Carrying	354
D	001010	----- ----- -	cmpli	Compare Logical Immediate	236
D	001011	----- ----- -	cmpi	Compare Immediate	235
D	001100	----- ----- -	addic	Add Immediate Carrying	227
D	001101	----- ----- -	addic.	Add Immediate Carrying & record CR	227
D	001110	----- ----- -	addi	Add Immediate	226
D	001111	----- ----- -	addis	Add Immediate Shifted	226
B	010000	----- ----0 0	bc	Branch Conditional	232
B	010000	----- ----0 1	bcl	Branch Conditional & Link	232
B	010000	----- ----1 0	bca	Branch Conditional Absolute	232
B	010000	----- ----1 1	bcla	Branch Conditional & Link Absolute	232
SC	010001	///// ////1 /	sc	System Call	330
I	010010	----- ----0 0	b	Branch	231
I	010010	----- ----0 1	bl	Branch & Link	231
I	010010	----- ----1 0	ba	Branch Absolute	231
I	010010	----- ----1 1	bla	Branch & Link Absolute	231
XL	010011	00000 00000 /	mcrf	Move Condition Register Field	299
XL	010011	00000 10000 0	bclr	Branch Conditional to Link Register	234
XL	010011	00000 10000 1	bclrl	Branch Conditional to Link Register & Link	234
XL	010011	00001 00001 /	crnor	Condition Register NOR	239
XL	010011	00001 00111 /	rfdi	Return From Debug Interrupt	----
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

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

Table 3-5. Instructions Sorted by Opcode (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary (Inst0:5)	Extended (Inst21:31)			
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 Immed then Mask Insert	327
M	010100	----- ----- 1	rlwimi.	Rotate Left Word Immed then Mask Insert & record CR	327
M	010101	----- ----- 0	rlwinm	Rotate Left Word Immed then AND with Mask	328
M	010101	----- ----- 1	rlwinm.	Rotate Left Word Immed then AND with Mask & record CR	328
M	010111	----- ----- 0	rlwnm	Rotate Left Word then AND with Mask	328
M	010111	----- ----- 1	rlwnm.	Rotate Left Word then AND with Mask & record CR	328
D	011000	----- ----- -	ori	OR Immediate	320
D	011001	----- ----- -	oris	OR Immediate Shifted	320
D	011010	----- ----- -	xori	XOR Immediate	365
D	011011	----- ----- -	xoris	XOR Immediate Shifted	365
D	011100	----- ----- -	andi.	AND Immediate & record CR	230
D	011101	----- ----- -	andis.	AND Immediate Shifted & record CR	230
??	011111	----- 01111 /	isel	Integer Select	----
X	011111	00000 00000 /	cmp	Compare	235
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

Legend:

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

Table 3-5. Instructions Sorted by Opcode (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary (Inst0:5)	Extended (Inst21:31)			
X	011111	/0000 01011 0	mulhwu	Multiply High Word Unsigned	314
X	011111	/0000 01011 1	mulhwu.	Multiply High Word Unsigned & record CR	314
X	011111	00000 10011 /	mfcr	Move From Condition Register	301
X	011111	00000 10100 /	lwarx	Load Word & Reserve Indexed	294
X	011111	00000 10110 /	icbt	Instruction Cache Block Touch	281
X	011111	00000 10111 /	lwzx	Load Word & Zero Indexed	297
X	011111	00000 11000 0	slw	Shift Left Word	332
X	011111	00000 11000 1	slw.	Shift Left Word & record CR	332
X	011111	00000 11010 0	cntlzw	Count Leading Zeros Word	237
X	011111	00000 11010 1	cntlzw.	Count Leading Zeros Word & record CR	237
X	011111	00000 11100 0	and	AND	230
X	011111	00000 11100 1	and.	AND & record CR	230
X	011111	00001 00000 /	cmpl	Compare Logical	236
X	011111	00001 01000 0	subf	Subtract From	351
X	011111	00001 01000 1	subf.	Subtract From & record CR	351
X	011111	00001 10110 /	dcbst	Data Cache Block Store	245
X	011111	00001 10111 /	lwzux	Load Word & Zero with Update Indexed	297
X	011111	00001 11100 0	andc	AND with Complement	230
X	011111	00001 11100 1	andc.	AND with Complement & record CR	230
X	011111	/0010 01011 0	mulhw	Multiply High Word	314
X	011111	/0010 01011 1	mulhw.	Multiply High Word & record CR	314
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

Legend:

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

Table 3-5. Instructions Sorted by Opcode (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary (Inst0:5)	Extended (Inst21:31)			
X	011111	00011 11100 1	nor.	NOR & record CR	319
X	011111	00100 00011 /	wrtee	Write External Enable	364
X	011111	00100 01000 0	subfe	Subtract From Extended with CA	353
X	011111	00100 01000 1	subfe.	Subtract From Extended with CA & record CR	353
X	011111	00100 01010 0	adde	Add Extended with CA	225
X	011111	00100 01010 1	adde.	Add Extended with CA & record CR	225
XFX	011111	00100 10000 /	mtcrf	Move To Condition Register Fields	306
X	011111	00100 10010 /	mtmsr	Move To Machine State Register	311
X	011111	00100 10110 1	stwcx.	Store Word Conditional Indexed & record CR	349
X	011111	00100 10111 /	stwx	Store Word Indexed	347
X	011111	00101 00011 /	wrteei	Write External Enable Immediate	364
X	011111	00101 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 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

Legend:

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

Table 3-5. Instructions Sorted by Opcode (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary (Inst0:5)	Extended (Inst21:31)			
X	011111	01000 10011 /	—	—	301
X	011111	01000 10110 /	dcbt	Data Cache Block Touch	246
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 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
X	011111	01110 11100 0	nand	NAND	317
X	011111	01110 11100 1	nand.	NAND & record CR	317
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’ Manual for the implementation

Table 3-5. Instructions Sorted by Opcode (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary (Inst0:5)	Extended (Inst21:31)			
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
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

Legend:

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

Table 3-5. Instructions Sorted by Opcode (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary (Inst0:5)	Extended (Inst21:31)			
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
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

Legend:

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

Table 3-5. Instructions Sorted by Opcode (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary (Inst0:5)	Extended (Inst21:31)			
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
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

Legend:

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

Table 3-5. Instructions Sorted by Opcode (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary (Inst0:5)	Extended (Inst21:31)			
D	110001	----- ----- -	—	—	287
D	110010	----- ----- -	—	—	286
D	110011	----- ----- -	—	—	286
D	110100	----- ----- -	—	—	342
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
A	111111	----- 10010 0	—	—	264

Legend:

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

Table 3-5. Instructions Sorted by Opcode (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary (Inst0:5)	Extended (Inst21:31)			
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
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

Legend:

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

Table 3-5. Instructions Sorted by Opcode (continued)

Format	Opcode		Mnemonic	Instruction	BooK E 0.99 Page
	Primary (Inst0:5)	Extended (Inst21:31)			
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
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

3.13 Optionally Supported APU Instructions

e200 cores optionally support several APUs. If a core does not implement a particular APU, it may treat these instructions as illegal, or may treat them as unimplemented. e200z1 treats the Embedded

Floating-Point APU instructions (efs$_{xxx}$, brinc) as unimplemented instructions because other cores of the e200 family implement this APU. All other non-supported APUs are treated as illegal instructions.

Chapter 4
Instruction Pipeline and Execution Timing

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

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

4.1 Overview of Operation

A block diagram of the e200 cores are shown in Figure 4-1. The instruction fetch unit prefetches instructions from memory into the instruction buffers. The decode unit decodes each instruction and generates information needed by the branch unit and the execution units.

The instruction fetch unit attempts to supply a constant stream of instructions to the execution pipeline. In the e200z1 it does so by decoding and detecting branches early in the instruction buffer, making branch predictions, and prefetching their branch targets into the instruction buffer. By prefetching the branch targets early, some or all of the branch pipeline bubbles can be hidden from the execution pipeline.

The instruction issue unit attempts to issue a single instruction each cycle to one of 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 which cause instruction issue to be stalled for one or more cycles until the hazard is eliminated.

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

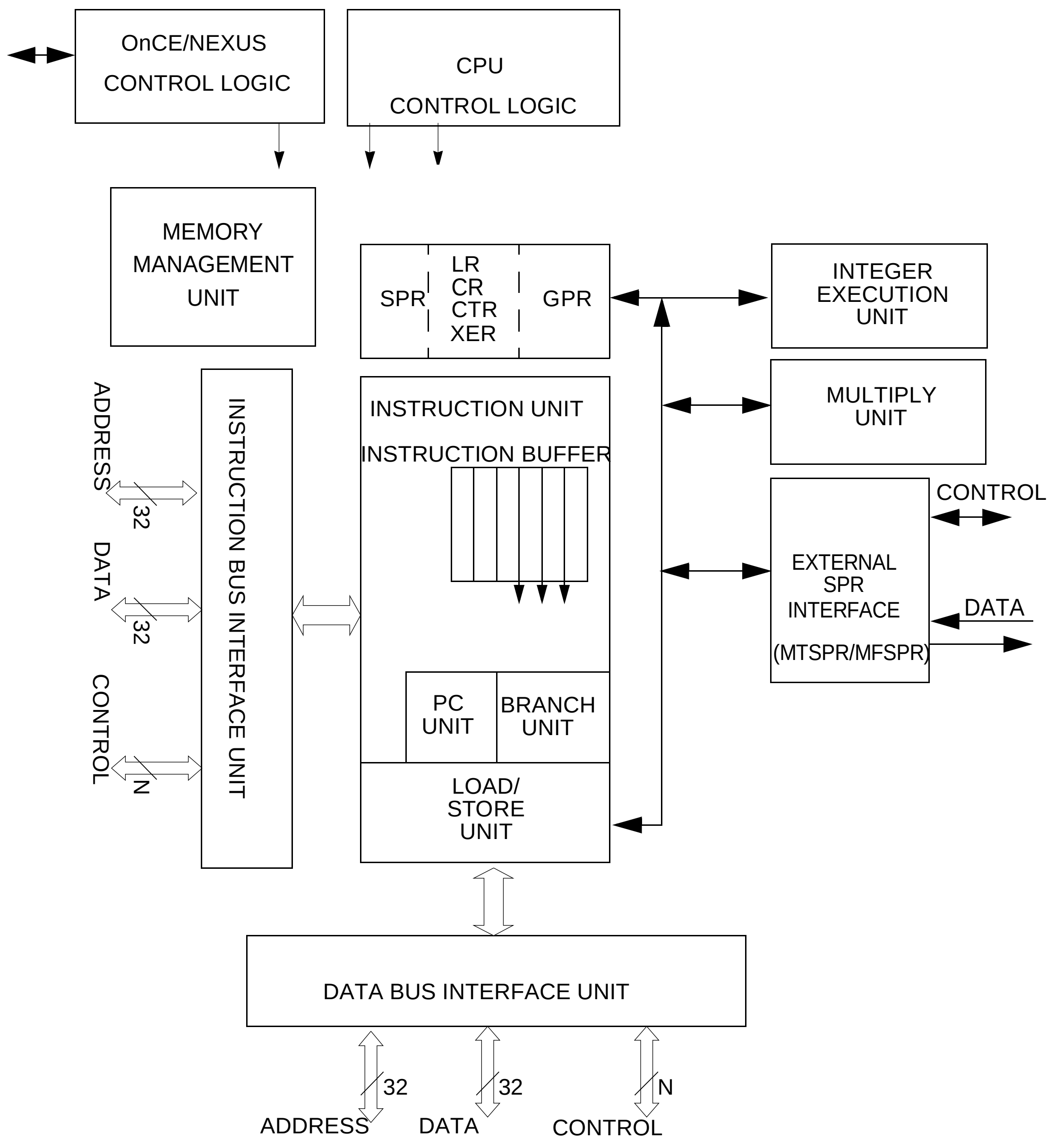


Figure 4-1. e200z1 Block Diagram

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 to the instruction buffers and decode unit. A set of 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

In the e200z1 the branch unit contains an four entry Branch Target Buffer (BTB) to accelerate execution of branch instructions.

Conditional branches which are not taken execute in a single clock on e200. Branches with successful target prefetching have an effective execution time of one clock in the e200z1. All other taken branches have an execution time of two clocks on e200.

4.1.4 Instruction Decode Unit

The decode unit includes the instruction buffers. A single instruction 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 unit, and the load/store unit. Included in the execution units section are the 32 general purpose registers (GPRs). Instructions with data dependencies begin execution when all such dependencies are resolved.

4.2.1 Integer Execution Unit

The integer execution unit is used to process arithmetic and logical instructions. Adds, subtracts, compares, count leading zeros, shifts and rotates execute in a single cycle.

Multiply instructions have a latency and throughput rate of 1 cycle.

Divide instructions have a variable latency (6–16 cycles) depending on the operand data. The worst case integer divide requires 16 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. A load followed by a dependent instruction does not incur any pipeline stall, except when the dependent instruction is a load/store instruction, and the latter instruction is using the previous load data for its effective address (EA) calculation, in which case a 1 cycle “register-busy” pipeline stall is incurred.

Loads, when free of the above effective address calculation dependency, execute with a maximum throughput of one per cycle and one cycle latency. Store data can be fed-forward from an immediately preceding load with no stall.

4.3 Instruction Pipeline

The four stage processor pipeline consists of stages for instruction fetch (IFETCH), instruction decode (DECODE), execution (EXECUTE), and result writeback (WB). For memory operations, the effective address generation occurs in the decode stage, while the memory access occurs in the execute stage.

The processor also contains an instruction prefetch buffer to allow buffering of instructions prior to the decode stage. Instructions proceed from this buffer to the instruction decode stage by entering the instruction decode register IR.

Table 4-1. Pipeline Stages

Stage	Description
IFETCH	Instruction Fetch From Memory
DECODE/EA	Instruction Decode / Register Read/ Operand Forwarding / EA Calculation
EXECUTE/MEM	Instruction Execution / Memory Access
WB	Write Back to Registers

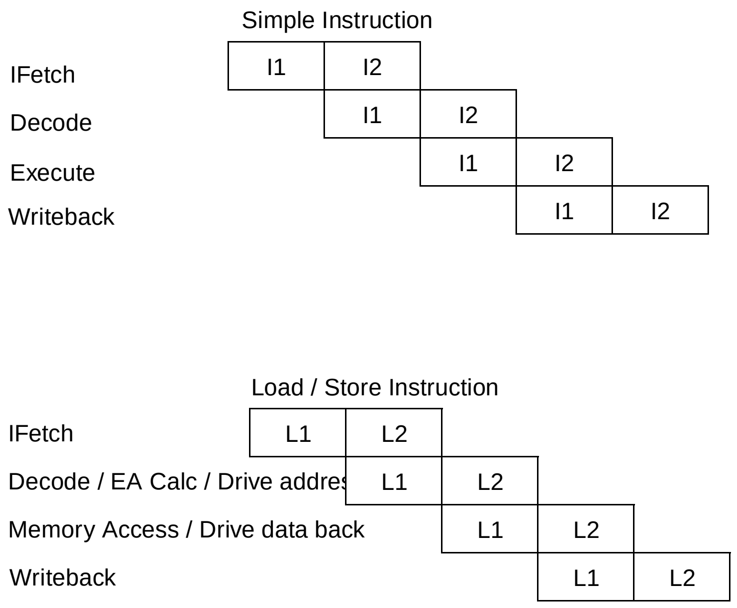


Figure 4-2. Pipeline Diagram

4.3.1 Description of Pipeline Stages

The IFetch pipeline stage retrieve instructions from the memory system and determine where the next instruction fetch is performed. Up to two 16-bit instructions are sent from memory to the instruction buffers every cycle.

The Decode pipeline stage decodes instructions and performs dependency checking. Simple integer instructions complete execution in the Execute stage of the pipeline.

Execution of load/store instructions is pipelined. The effective address calculations for load/store instructions are performed in the decode stage. This effective address is driven out to the data memory in the same stage. The actual memory access occurs in the execute stage.

Load-to-use dependencies do not incur pipeline bubbles except when the dependent instruction is a load or store instruction, and the latter instruction is dependent on its previous load data for EA calculation. If an ALU instruction is dependent on a load instruction, the data is fed directly into the ALU for execution. No pipeline bubble is incurred in this case.

Multiply instructions require 1 clock to execute.

All condition-setting instructions complete in the Execute stage of the pipeline.

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.

4.3.2 Instruction Buffers

e200 contains a set of instruction buffers which supply instructions into the Instruction Register (IR) for decoding.

Instruction prefetches request a 64-bit doubleword and the buffer is filled with a pair of instructions at a time, except for the case of a change of flow fetch where the target is to the second (odd) word. In that case only a 32-bit prefetch is performed to load the instruction 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 Slot 0, 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 buffer empties, instruction issue stalls, and the buffer is refilled. The first returned instruction is forwarded directly to the IR.

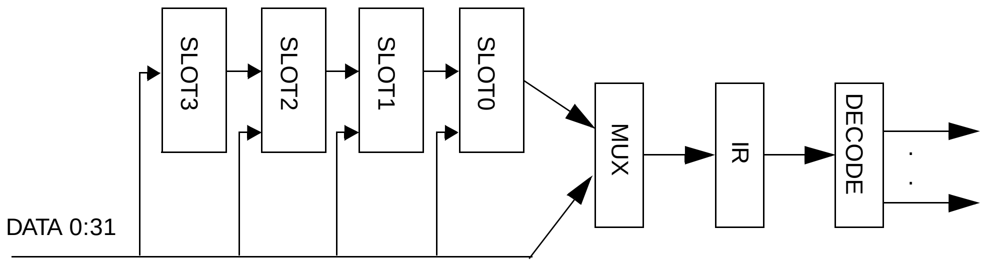


Figure 4-3. e200 Instruction Buffers

4.3.2.1 Branch Prediction in e200z1

In e200z1 the HID0[BPRED] field is used to control whether prediction will be made for forward or backward branches (or both).

To resolve branch instructions and improve the accuracy of branch predictions, e200 implements a dynamic branch prediction mechanism using a 4-entry branch target buffer (BTB), a fully associative address cache of branch target addresses. The BTB on e200 is purposefully small to reduce cost and power. It is expected to accelerate the execution of loops with some potential change of flow within the loop body.

An entry is allocated in the BTB whenever a branch resolves as taken and the BTB is enabled. Branches that have not been allocated are always predicted as not taken. Entries in the BTB are allocated on taken branches using a FIFO replacement algorithm.

Each BTB entry holds a 2-bit branch history counter, whose value is incremented or decremented on a BTB hit, depending on whether the branch was taken. The counter can assume four different values: strongly taken, weakly taken, weakly not taken, and strongly not taken.

A branch 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 mispredicted branch, the instruction fetch stream returns to the sequential instruction stream after the branch has been resolved.

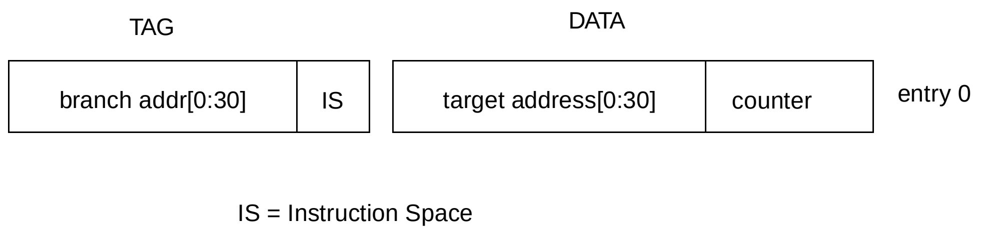
When a branch is predicted taken and the branch is later resolved (in the branch decode stage), the value of the counter is updated. A branch whose counter indicates weakly taken is resolved as taken, the counter increments so that the prediction becomes strongly taken. If the branch resolves as not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

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

Dynamic branch prediction is enabled by setting BUCSR[BPEN]. Clearing BUCSR[BPEN] disables dynamic branch prediction, in which case e200 predicts every branch as not taken. Additional control is available in the HID0[BPRED] 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, HID0[BPRED] has no further effect on that branch entry.

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

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



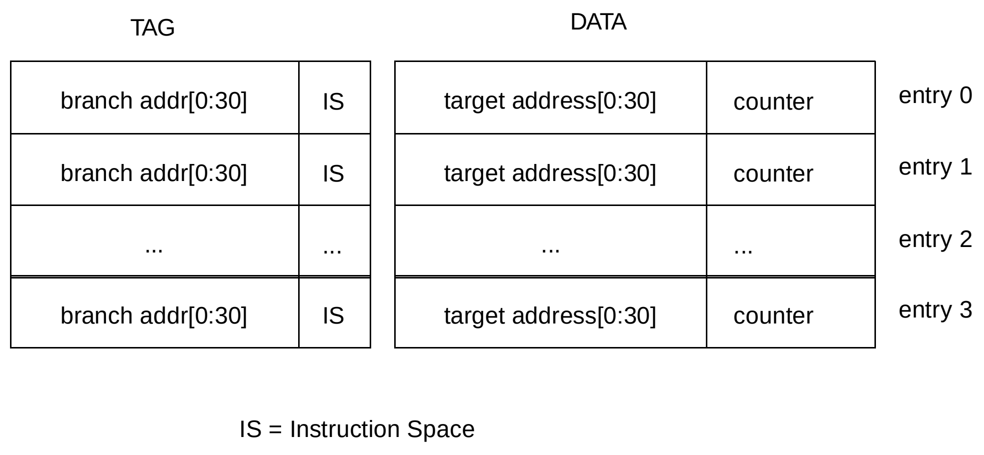


Figure 4-4. e200 Branch Target Buffer

4.3.3 Single-Cycle Instruction Pipeline Operation

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

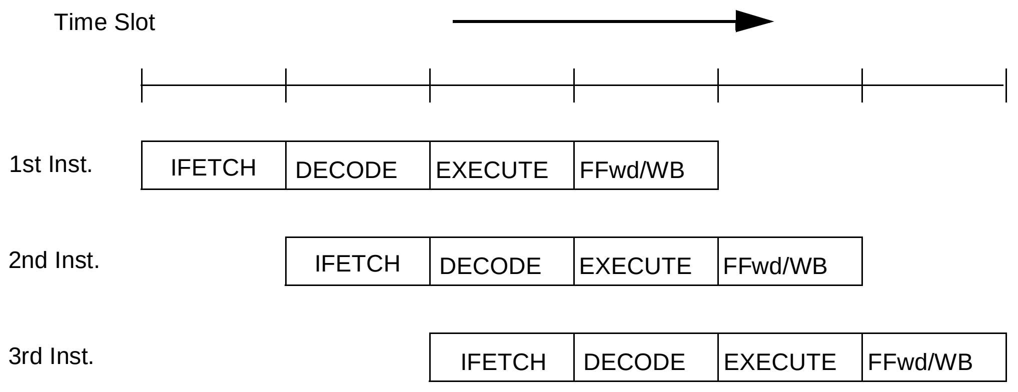


Figure 4-5. Basic Pipeline Flow, Single Cycle Instructions

4.3.4 Basic Load and Store Instruction Pipeline Operation

The effective address (EA) calculations for load and store instructions are performed in the decode stage. The memory access occurs in the execution stage.

If a load instruction is followed by an dependent ALU instruction, the load data is driven from the memory in the MEM stage and feed-forwarded into the dependent ALU instruction in the following cycle. As a result, the is no load-to-use pipeline bubble. Figure 4-7 shows the instruction flow for a load instruction followed by a dependent add instruction.

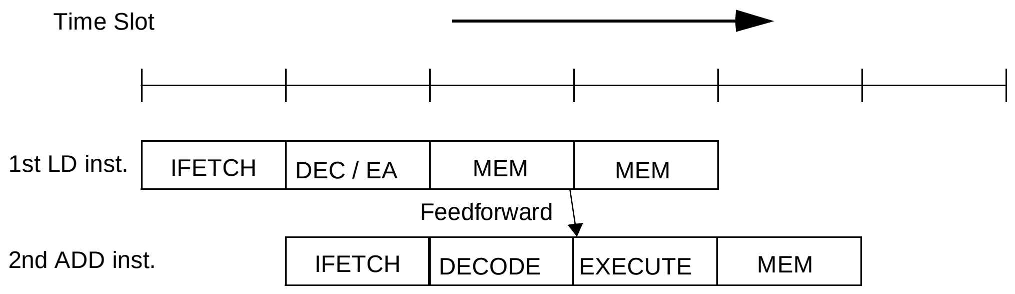


Figure 4-6. A Load Followed By A Dependent Add Instruction

Back-to-back load/store instructions are executed in a pipelined fashion, provided that their effective address calculations are not dependent on their previous load instructions. Figure 4-7 shows the basic pipe line flow for two back-to-back load instructions. In this case, the 2nd load does not depend on its previous load data for its EA calculation. Notice that the memory access of the first load instruction overlaps in time with the EA calculation of the second load instruction.

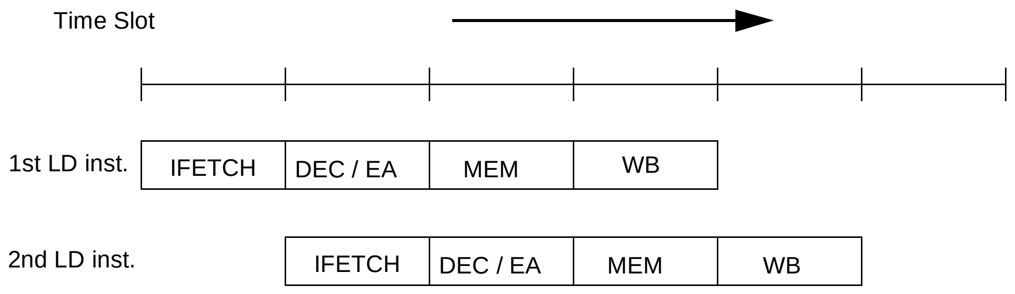


Figure 4-7. Back-to-back Load Instructions

When a load is followed by a load or a store instruction that depends on the first load data for EA calculation, a pipeline stall is incurred. Figure 4-8 shows the instruction flow for a load instruction followed by a dependent store instruction through EA calculation. The second store instruction, in this case, is dependent on the first load instruction for its EA calculation.

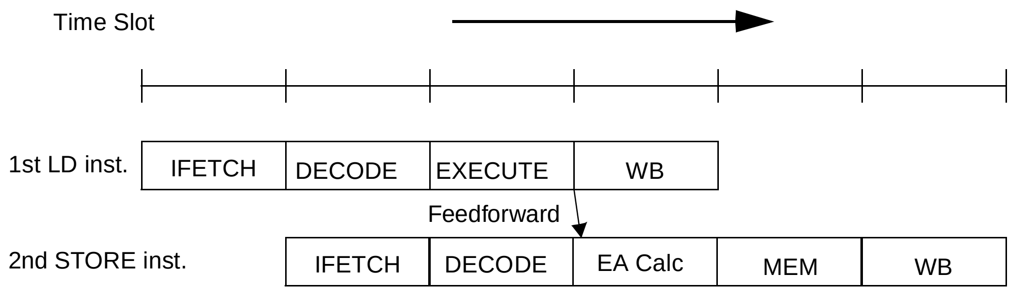


Figure 4-8. A Load Followed By A Dependent Store Instruction

A store instruction that depends on its previous load for its store data does not stall the pipeline.

4.3.5 Change-of-Flow Instruction Pipeline Operation

A branch instruction takes either one or two cycles to execute. Simple change of flow instructions require 2 cycles to refill the pipeline with the target instruction for taken branches and branch and link instructions with no prediction.

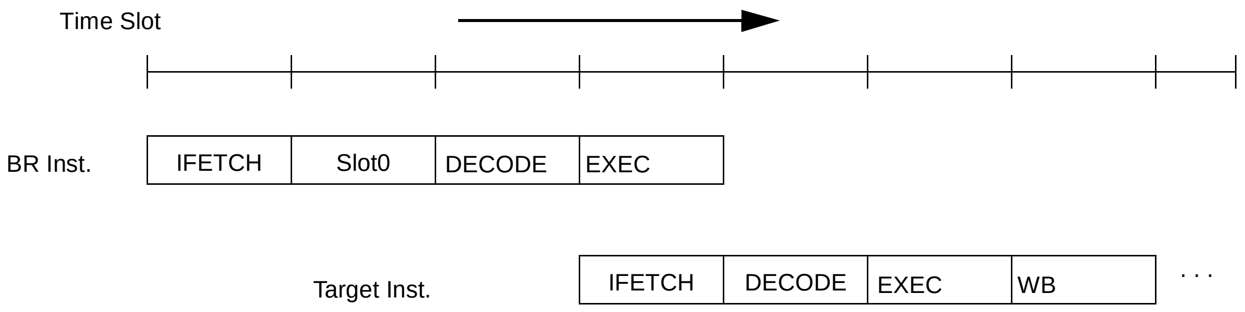


Figure 4-9. Basic Pipeline Flow, Branch Instructions

For branch type instructions in e200z1, in some situations this 2 cycle timing may be reduced by performing the target fetch speculatively while the branch instruction is still being fetched into the instruction buffer. The branch target address is obtained from the BTB. The resulting branch timing reduces to a single clock when the target fetch is initiated early enough and the branch is taken.

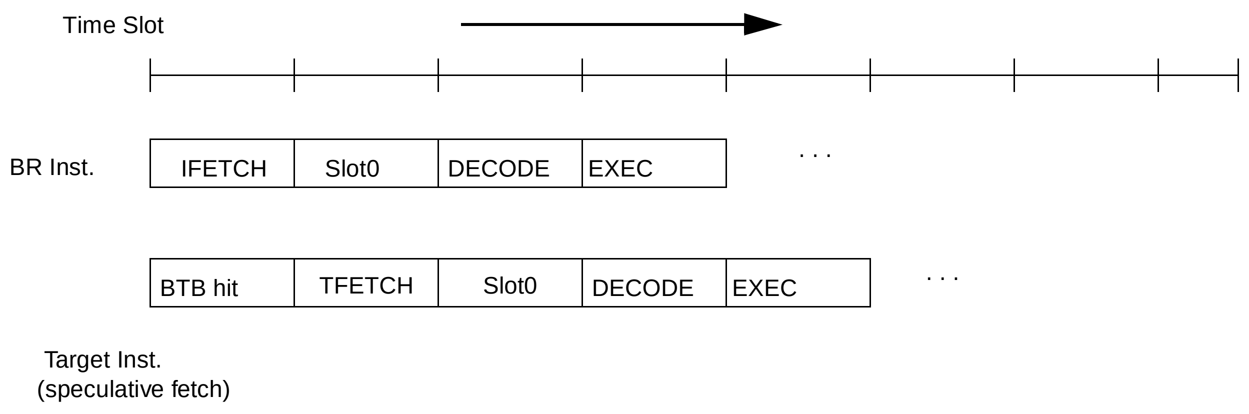


Figure 4-10. Basic Pipeline Flow, Branch Speculation

4.3.6 Basic Multi-Cycle Instruction Pipeline Operation

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

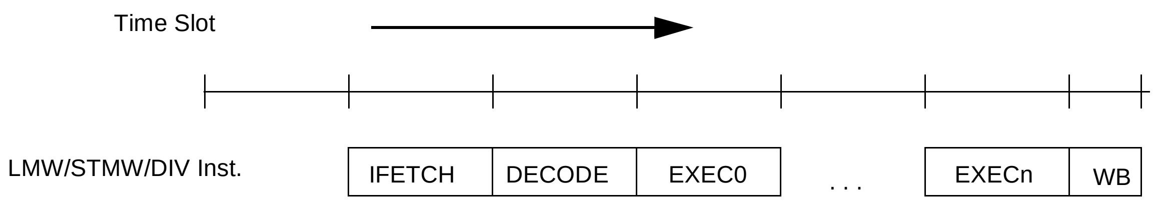


Figure 4-11. Basic Pipeline Flow, Multi-cycle Instructions

Instructions must complete and write back results in order. A single cycle instruction which follows a multi-cycle instruction must wait for completion of the multi-cycle instruction prior to its writeback in order to meet the in-order requirement. Result feed-forward paths are provided so that execution may continue prior to result writeback.

4.3.7 Additional Examples of Instruction Pipeline Operation for Load and Store

Figure 4-12 shows an example of pipelining two non-data dependent load or store instructions with a following data dependent single cycle instruction. While the first load or store begins accessing memory in the MEM stage, the next load or store can be calculating a new effective address in the DEC/EA stage. The **add** in this example does not stall even though there is a data dependency on its preceding load instruction.

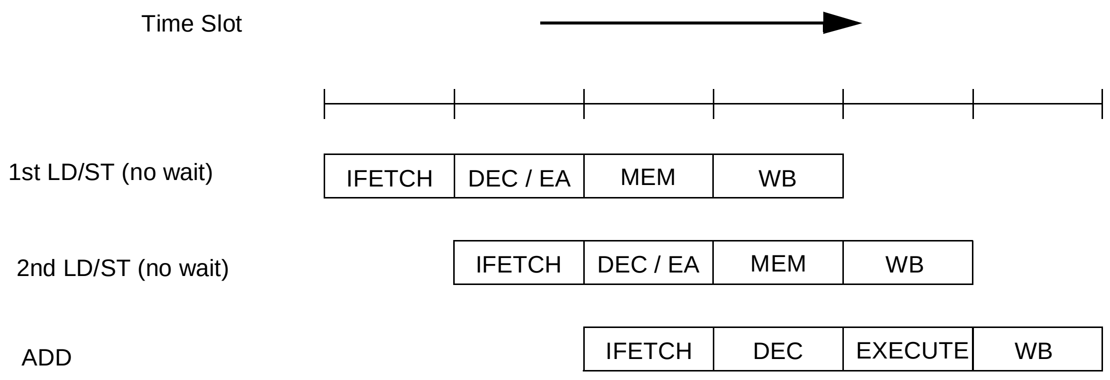


Figure 4-12. Pipelined Load/Store Instructions

For memory access instructions, wait-states may occur. This causes a following memory access instruction to stall because the following memory access may not be initiated as shown in Figure 4-13. Here, the first **ld/st** instruction incurs a wait-state on the bus interface, causing succeeding instructions to stall.

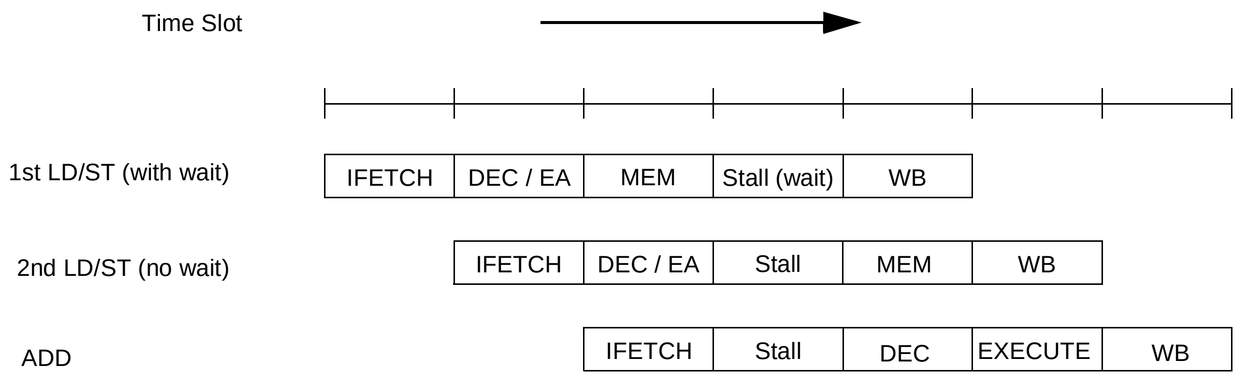


Figure 4-13. Pipelined Load/Store Instructions with Wait-state

4.3.8 Move to/from SPR Instruction Pipeline Operation

Most **mtspr** and **mfspr** instructions are treated like single cycle instructions in the pipeline, and do not cause stalls. Exceptions are for the MSR, the Debug SPRs, the Embedded Floating Point Unit, and MMU SPRs that do cause stalls. The following figures show examples of **mtspr** and **mfspr** instruction timing.

Figure 4-14 applies to the Debug SPRs. These instructions do not begin execution until all previous instructions have finished their execute stage. If the previous instruction of **mfspr** or **mtspr** is a multicycle instruction, the **mfspr** and **mtspr** instructions do not begin execution until its previous instruction moves into the WB stage as shown in Figure 4-14. In addition, execution of subsequent instructions is stalled until the **mfspr** and **mtspr** instructions complete.

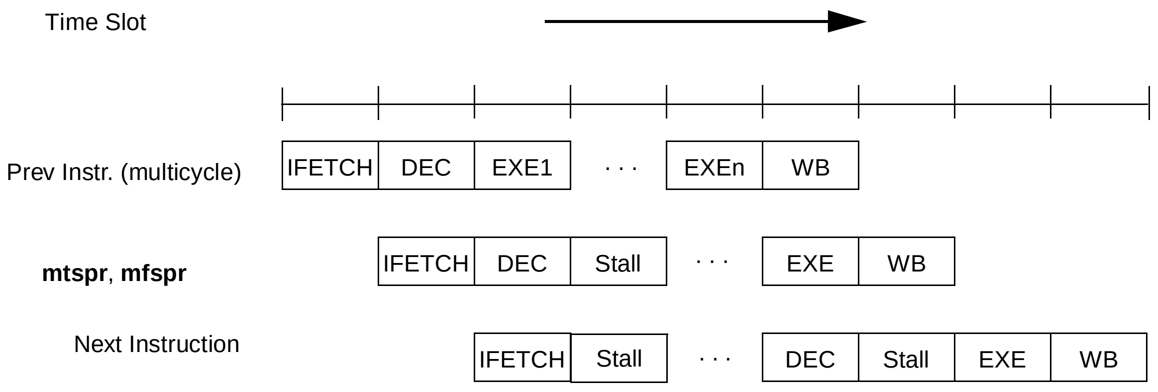


Figure 4-14. mtspr, mfspr Instruction Execution—(1)

Figure 4-15 applies to the **mtmsr** instruction and the **wrtee** and **wrteei** instructions. Execution of subsequent instructions is stalled until these instructions writeback.

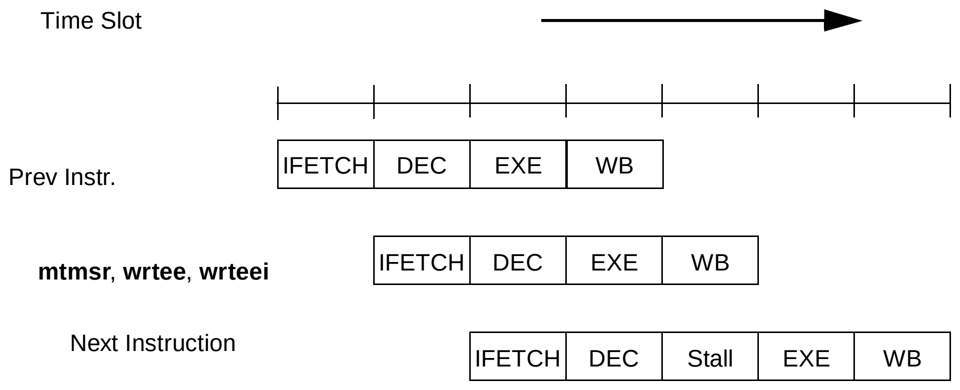


Figure 4-15. mtmsr, wrtee, wrteei Instruction Execution

Access to MMU SPRs are stalled until all outstanding bus accesses have completed and the MMU is idle (*p_[i,d]_cmbusy* negated) to allow an access window where no translations or cache cycles are required. Figure 4-16 shows an example where an outstanding bus access causes **mtspr**/**mfspr** execution to be delayed until the bus becomes idle. Processor access requests will be held off during execution of a MMU SPR instruction. A subsequent access request may be generated in the WB cycle. This same protocol applies to MMU management instructions (for example, **tlbre**, **tlbwe**, etc.) as well as to the DCRs. See Section 7.2, “Internal Interface Signals,” for a definition of internal signals used in Figure 4-16.

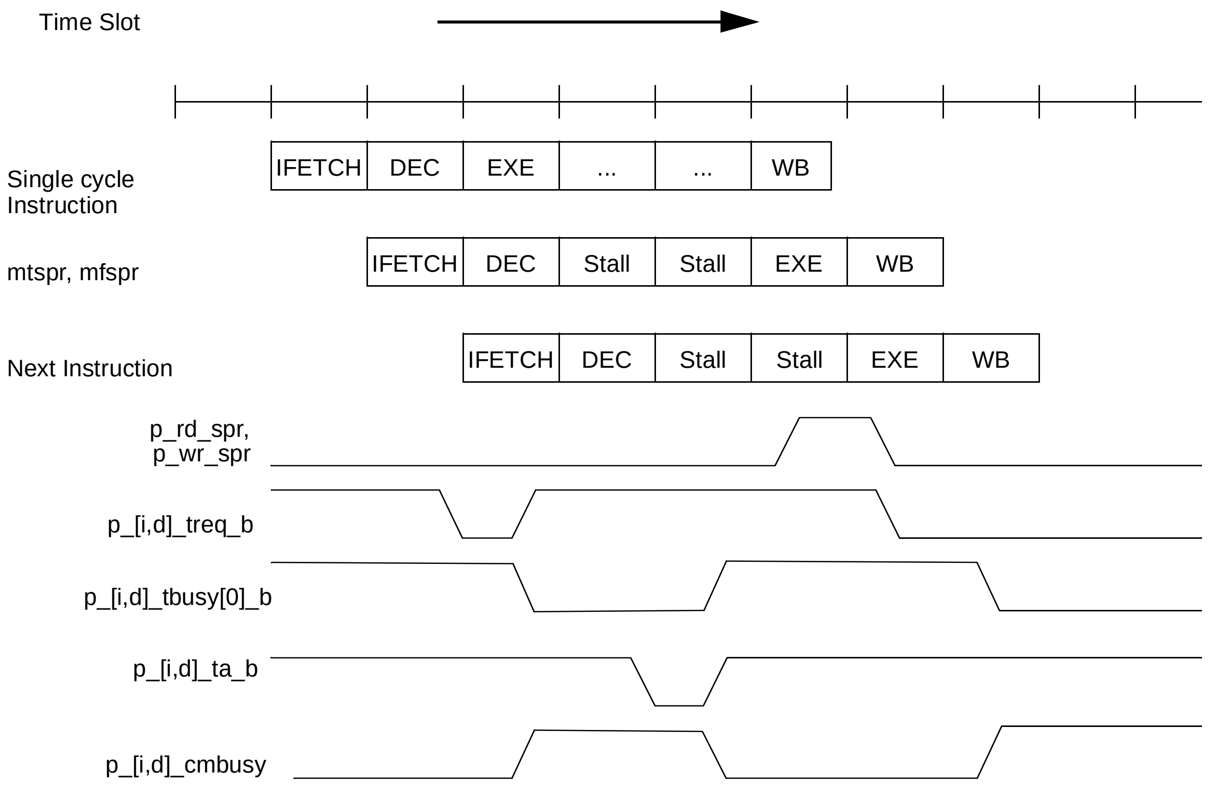


Figure 4-16. DCR, MMU mtspr, mfspr and MMU Management Instruction Execution

4.4 Control Hazards

Several internal control hazards exist in e200 which can cause certain instruction sequences to incur one or more stall cycles. These include the following:

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

4.5 Instruction Serialization

The types of serialization required by the core are as follows:

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

4.5.1 Completion Serialization

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

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

4.5.2 Dispatch Serialization

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

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 instruction **isync**.
- The **rfi**, **rfci**, **rfdi**, **se_rfi**, **se_rfci**, **se_rfdi**, **sc**, and **se_sc** instructions.

4.6 Interrupt Recognition and Exception Processing

Figure 4-17 shows timing for interrupt recognition and exception processing overhead. This example shows best-case response timing when an interrupt is received and processed during execution of a sequence of single-cycle instructions.

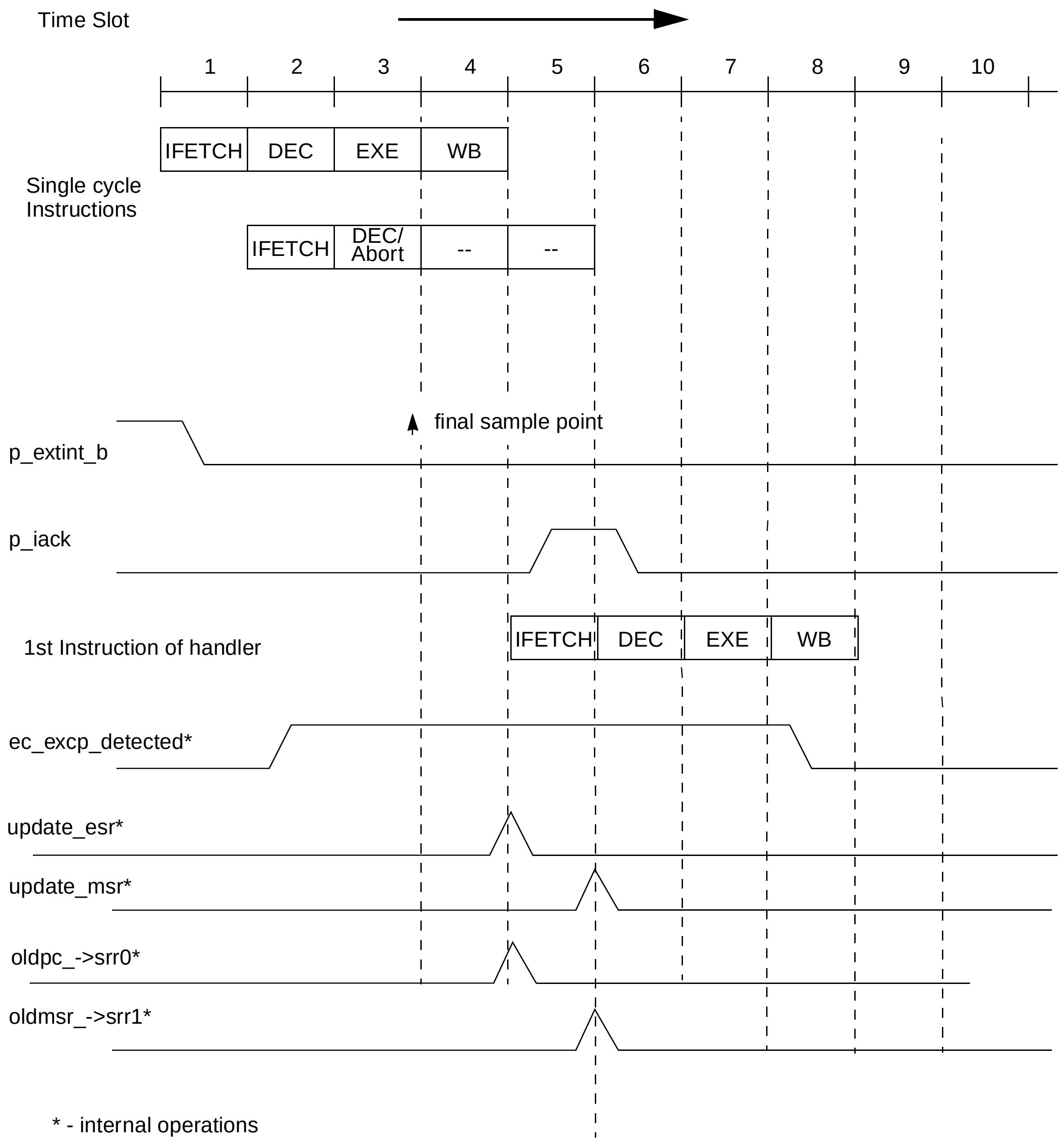


Figure 4-17. Interrupt Recognition and Handler Instruction Execution

Figure 4-18 shows timing for interrupt recognition and exception processing overhead. This example shows best-case response timing when an interrupt is received and processed during execution of a load or store instruction. The fetch for the handler is delayed until completion of the load or store, regardless of the number of wait-states.

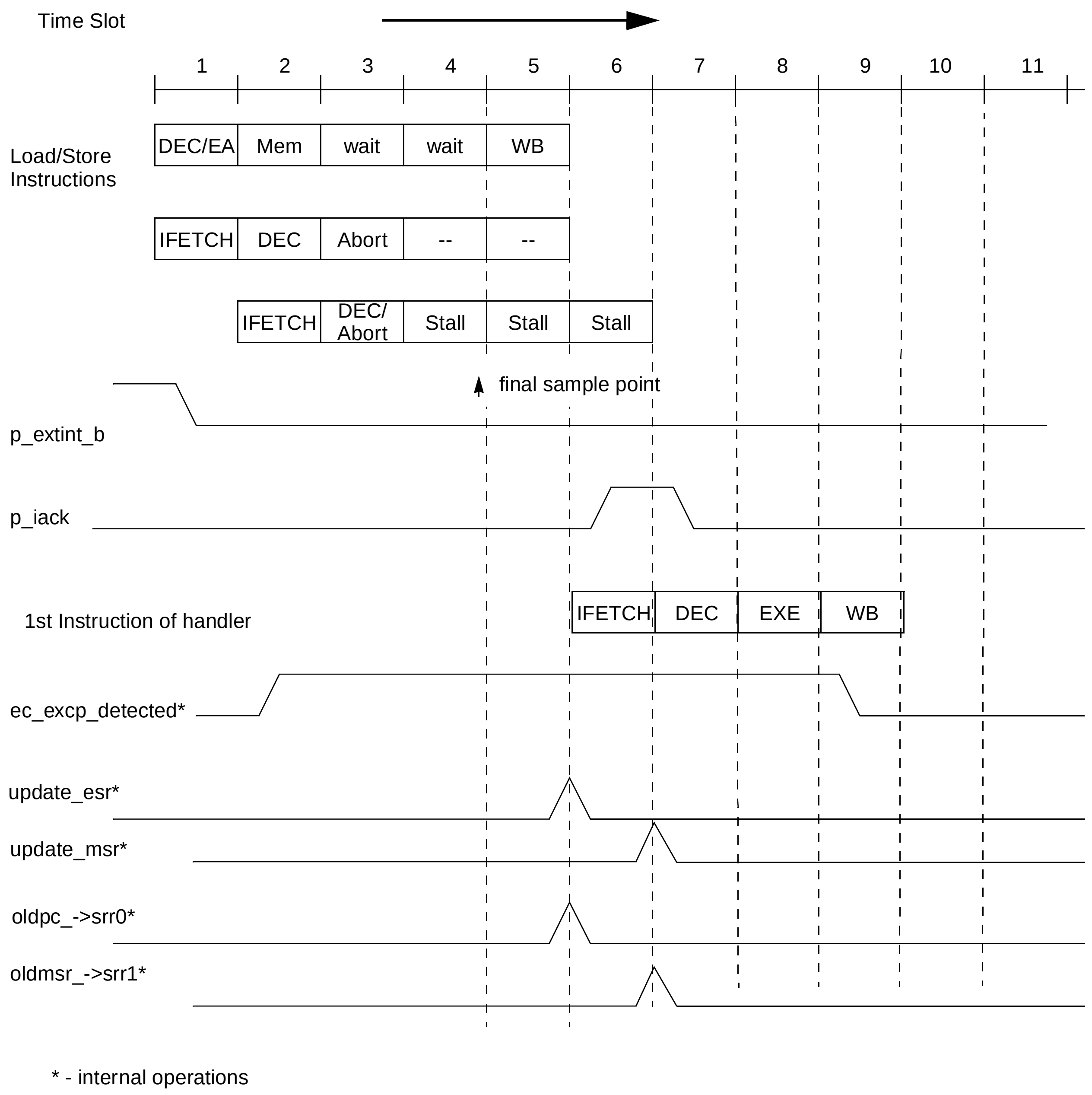


Figure 4-18. Interrupt Recognition and Handler Instruction Execution—Load/Store in Progress

Figure 4-19 shows timing for interrupt recognition and exception processing overhead. This example shows best-case response timing when an interrupt is received and processed during execution of a multicycle interruptible instruction.

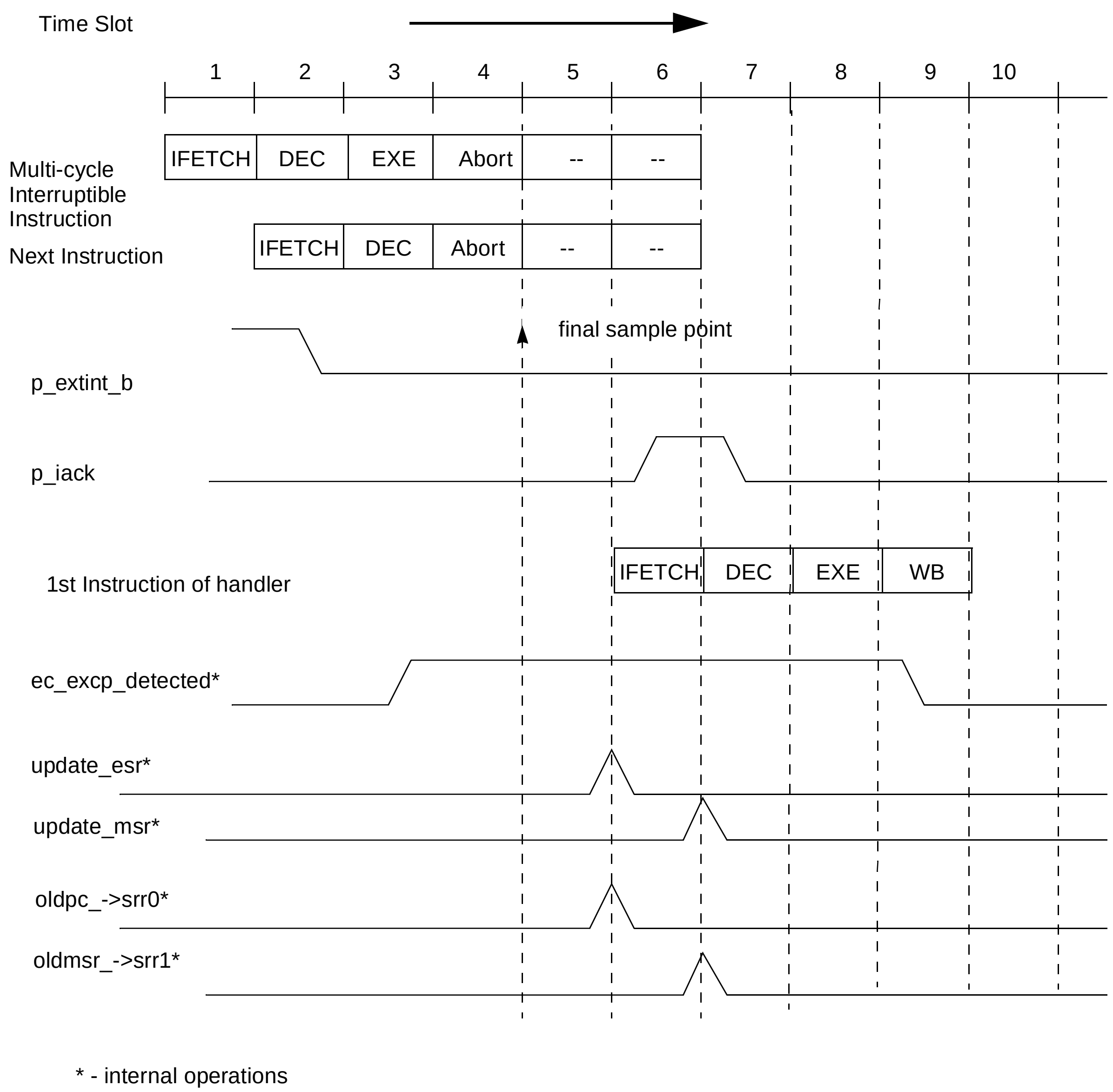


Figure 4-19. Interrupt Recognition and Handler Instruction Execution—Multi-Cycle Instruction Abort

4.7 Instruction Timings

Instruction timing in number of processor clock cycles for various instruction classes is shown in Table 4-2. Pipelined instructions are shown with cycles of total latency and throughput cycles. Divide instructions are not pipelined and block other instructions from executing during execution.

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

Table 4-2. Instruction Class Cycle Counts

Class of Instructions	Latency	Throughput	Special Notes
integer: **add**, **sub**, **shift**, **rotate**, **logical**, **cntlzw class** instructions	1	1	—
integer: compare	1	1	—
Branch	2/1	2/1	Branches take either 2 or 1 cycles to execute.
multiply	1	1	—
divide	6-16	6-16	data dependent timing
CR logical	1	1	—
loads (non-multiple)	1	1	—
load multiple	1 + n	1 + n	Actual timing depends on n and address alignment. (n = number of registers transferred)
stores (non-multiple)	1	1	—
store multiple	1 + n	1 + n	Actual timing depends on n and address alignment. (n = number of registers transferred)
mtmsr, **wrtee**, **wrteei**	2&	2	—
mcrf	1	1	—
mfspr, **mtspr**	2&	2&	applies to Debug SPRs, optional unit SPRS
mfspr, **mfmsr**	1	1	applies to internal, non Debug SPRs
mfcr, mtcr	1	1	—
rfi, **rfci**, **rfdi, se_rfi, se_rfci, se_rfdi**	3	—	—
sc, se_sc	3	—	—
tw, **twi**	3	—	Trap taken timing

4.8 Operand Placement on Performance

The placement (location and alignment) of operands in memory affects relative performance of memory accesses, and in some cases, affects it significantly. Table 4-3 indicates the effects for the e200 core.

In Table 4-3, optimal means that one effective address (EA) calculation occurs during the memory operation. Good means that multiple EA calculations occur during the memory operation which may cause additional bus activities with multiple bus transfers. Poor means that an alignment interrupt is generated by the storage operation.

Table 4-3. Performance Effects of Storage Operand Placement

Operand		
Size	**Byte Align**	**Performance**
4 Byte	4 <4	optimal good
2 Byte	2 <2	optimal good
1 Byte	1	optimal
lmw, stmw	4 <4	good poor
string	N/A	—

Notes:

Optimal: One EA calculation occurs.

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

Poor: Alignment Interrupt occurs.

Chapter 5
Interrupts and Exceptions

The Power Architecture Book E document defines the mechanisms by which the e200 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 which, 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) and the processor begins execution at an address (interrupt vector) determined by the Interrupt Vector Prefix register (IVPR), and one of the hardwired Interrupt Vector Offset values. 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. Three sets of these registers are implemented; SRR0 and SRR1 for non-critical interrupts, CSRR0 and CSRR1 for critical interrupts, and DSRR0 and DSRR1 for debug interrupts (when the Debug APU is enabled). Hardware supports nesting of critical interrupts within non-critical interrupts, and debug interrupts within both critical and non-critical 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**, or **rfdi** (**se_rfi**, **se_rfci**, or **se_rfdi** in VLE mode) instruction to restore state information from the respective machine state save/restore register pair.

5.1 e200 Interrupts

As specified by the Power Architecture Book E specification, 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.

The types of interrupts handled are shown in Table 5-1.

Table 5-1. Interrupt Classifications

Interrupt Types	Synchronous/Asynchronous	Precise/Imprecise	Critical/Non-Critical/Debug
System Reset	Asynchronous, non-maskable	Imprecise	—
Machine Check	—	—	Critical
Critical Input Interrupt Watchdog Timer Interrupt	Asynchronous, maskable	Imprecise	Critical
External Input Interrupt Fixed-Interval Timer Interrupt Decrementer Interrupt	Asynchronous, maskable	Imprecise	Non-critical
Instruction-based Debug Interrupts	Synchronous	Precise	Critical / Debug
Debug Interrupt (UDE) Debug Imprecise Interrupt	Asynchronous	Imprecise	Critical / Debug
Data Storage / Alignment/TLB Interrupts Instruction Storage/TLB Interrupts	Synchronous	Precise	Non-critical

These classifications are discussed in greater detail in Section 5.7, "Interrupt Definitions." Interrupts implemented in e200 and the exception conditions that cause them are listed in Table 5-2.

Table 5-2. Exceptions and Conditions

Interrupt Type	Corresponding Interrupt Vector Offset	Causing Conditions
System reset	none, vector to [*p_rstbase*[0:29]] || 2'b00	Reset by assertion of *p_reset_b* Watchdog Timer Reset Control Debug Reset Control
Critical Input	IVOR 0[1]	*p_critint_b* is asserted and MSR[CE]=1
Machine check	IVOR 1	*p_mcp_b* is asserted and MSR[ME] =1 ISI, ITLB Error on first instruction fetch for an exception handler and current MSR[ME]=1 Bus error (XTE) with MSR[EE]=0 and current MSR[ME]=1

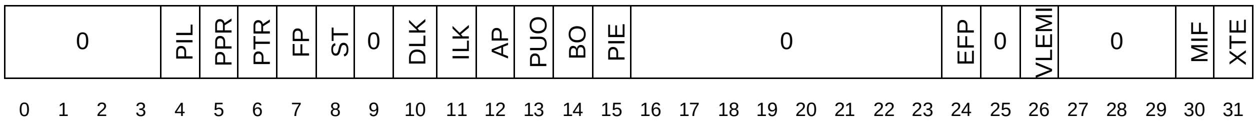
Table 5-2. Exceptions and Conditions (continued)

Interrupt Type	Corresponding Interrupt Vector Offset	Causing Conditions
Data Storage	IVOR 2	Access control. Byte ordering due to misaligned access across page boundary to pages with mismatched E bits. Precise external termination error (*p_d_tea_b* assertion and precise recognition) and MSR[EE]=1
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 Precise external termination error (*p_i_tea_b* assertion and precise recognition) and MSR[EE]=1. See Section 7.2, "Internal Interface Signals," for a definition of internal signals.
External Input	IVOR 4[1]	*p_extint_b* is asserted and MSR[EE]=1.
Alignment	IVOR 5	**lmw**, **stmw** not word aligned. **dcbz** with disabled cache or no cache present, or to W or I storage. **lwarx** or **stwcx.** not word aligned.
Program	IVOR 6	Illegal, Privileged, Trap, FP enabled, AP enabled, Unimplemented Operation.
Floating-point unavailable	IVOR 7	MSR[FP]=0 and attempt to execute a Book E floating point operation.
System call	IVOR 8	Execution of the System Call (**sc**, **se_sc**) instruction
AP unavailable	IVOR 9	Unused
Decrementer	IVOR 10	As specified in Book E v.0.99 *Book E: Enhanced PowerPC™ Architecture*
Fixed Interval Timer	IVOR 11	As specified in Book E v.0.99*Book E: Enhanced PowerPC™ Architecture*
Watchdog Timer	IVOR 12	As specified in Book E v.0.99*Book E: Enhanced PowerPC™ Architecture*
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	—

[1] Autovectored External and Critical Input interrupts use this IVOR. Vectored interrupts supply an interrupt vector offset directly.

5.2 Exception Syndrome Register

The Exception Syndrome Register (ESR) provides a *syndrome* to differentiate between exceptions that can generate the same interrupt type. e200 adds some implementation specific bits to this register, as seen in Figure 5-1.



SPR—62; Read/Write; Reset—0x0

Figure 5-1. Exception Syndrome Register (ESR)

The ESR bits are defined in Table 5-3.

Table 5-3. ESR Bit Settings

Bit(s)	Name	Description	Associated Interrupt Type
0:3 (32:35)	—	Allocated[1]	—
4 (36)	PIL	Illegal Instruction exception	Program
5 (37)	PPR	Privileged Instruction exception	Program
6 (38)	PTR	Trap exception	Program
7 (39)	FP	Floating-point operation	Alignment (not on e200) Data Storage (not on e200) Data TLB (not on e200) Program
8 (40)	ST	Store operation	Alignment Data Storage Data TLB
9 (41)	—	Reserved[2]	—
10 (42)	DLK[3]	Data Cache Locking	Data Storage
11 (43)	ILK[2]	Instruction Cache Locking	Data Storage
12 (44)	AP	Auxiliary Processor operation (Not used by e200)	Alignment (not on e200) Data Storage (not on e200) Data TLB (not on e200) Program (not on e200)
13 (45)	PUO	Unimplemented Operation exception	Program

Table 5-3. ESR Bit Settings (continued)

Bit(s)	Name	Description	Associated Interrupt Type
14 (46)	BO	Byte Ordering exception Mismatched Instruction Storage exception	Data Storage Instruction Storage
15 (47)	PIE	Program Imprecise exception (Reserved)	Currently unused by e200
16:23 (48:55)	—	Reserved[2]	—
24 (56)	EFP	Embedded Floating-point APU Operation	Allocated, is not set by hardware
25 (57)	—	Allocated[1]	—
26 (58)	VLEMI	VLE Mode Instruction	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)	XTE	External Termination Error (Precise)	Data Storage Instruction Storage

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

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

[3] Unused on e200z1.

5.3 Machine State Register

The Machine State Register defines the state of the processor. The e200 MSR is shown in Figure 5-2.

e200z1	0					UCLE	Allocated	0						WE	CE	0	EE	PR	FP	ME	FE0	0	DE	FE1	0		IS	DS	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 5-2. Machine State Register (MSR)

The MSR bits are defined in Table 5-4.

Table 5-4. MSR Bit Settings

Bit(s)	Name	Description
0:4 (32:36)	—	Reserved[1]
5 (37)	UCLE[2]	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)	—	Reserved[1]
7:12 (39:44)	—	Reserved[1]
13 (45)	WE	Wait State (Power management) enable. This bit is defined as optional in the Power Architecture Book E 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)	—	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 (for example, GPRs, SPRs, MSR, etc.). 1 The processor is in user mode, cannot execute any privileged instruction, and cannot access any privileged resource.
19 (51)	ME	Machine Check Enable 0 Machine Check interrupts are disabled. Checkstop mode is entered when the *p_mcp_b* input is recognized asserted or an ISI or ITLB exception occurs on a fetch of the first instruction of an exception handler. 1 Machine Check interrupts are enabled.
20 (52)	FE0	Floating-point exception mode 0 (not used by e200)
21 (53)	—	Reserved[1]
22 (54)	DE	Debug Interrupt Enable 0 Debug interrupts are disabled. 1 Debug interrupts are enabled.

Table 5-4. MSR Bit Settings (continued)

Bit(s)	Name	Description
23 (55)	FE1	Floating-point exception mode 1 (not used by e200)
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).
28:29 (60:61)	—	Reserved[1]
30:31 (62:63)	—	Preserved[1]

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

[2] This bit is implemented, but ignored

5.4 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 5-3.

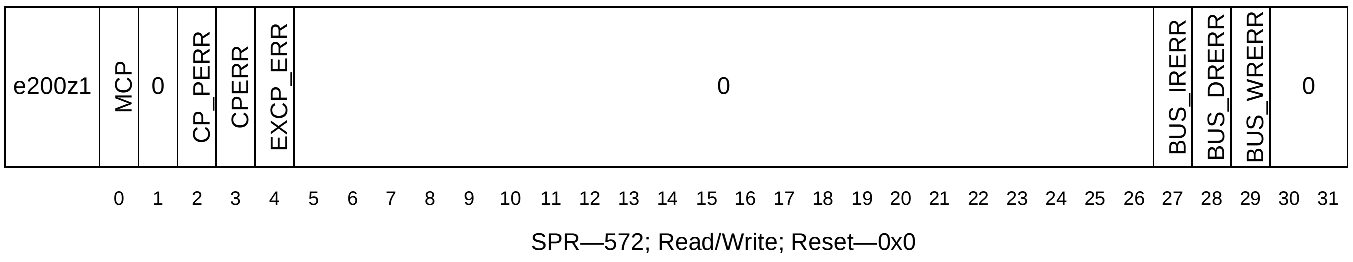


Figure 5-3. Machine Check Syndrome Register (MCSR)

Table 5-5 describes MCSR fields. The MCSR indicates the source of a machine check condition is recoverable. When a syndrome bit in the MCSR is set, the core complex asserts *p_mcp_out* for system information.

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

Bit	Name	Description	Recoverable
0 (32)	MCP	Machine check input pin	Maybe
1 (33)	—	Reserved, should be cleared.	—
2 (34)	CP_PERR[1]	Cache push parity error	Unlikely
3 (35)	CPERR[1]	Cache parity error	Precise
4 (36)	EXCP_ERR	ISI, ITLB, or Bus Error on first instruction fetch for an exception handler	Precise
26 (58)	—	Reserved, should be cleared.	—
27 (59)	BUS_IRERR	Read bus error on Instruction fetch	Unlikely
28 (60)	BUS_DRERR	Read bus error on data load	Unlikely
29 (61)	BUS_WRERR	Write bus error on data store	Unlikely
30:31 (62:63)	—	Reserved, should be cleared.	—

[1] This bit is implemented, but is never set by hardware.

5.5 Interrupt Vector Prefix Register (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 hardwired Interrupt Vector Offset value 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. Note that for e200z1 the IVPR has been extended from 16 to 20 bits, allowing the vector table to reside on any 4-Kbyte boundary. The format of IVPR is shown in Figure 5-4.

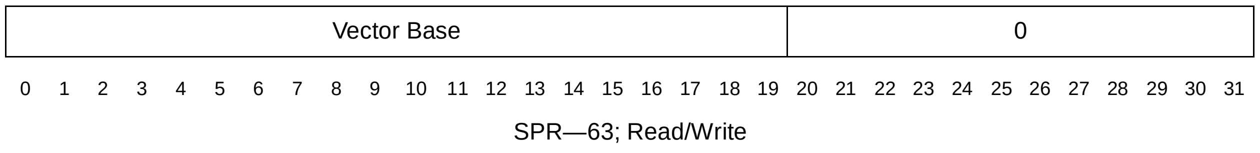


Figure 5-4. e200 Interrupt Vector Prefix Register (IVPR)

The IVPR fields are defined in Table 5-6.

Table 5-6. IVPR Register Fields

Bit(s)	Name	Description
0:19 (32:51)	Vec Base	Vector Base This field is used to define the base location of the vector table, aligned to a 64Kbyte boundary. This field provides the high-order 20 bits of the location of all interrupt handlers. The contents of the IVORxx register value appropriate for the type of exception being processed are concatenated with the IVPR Vector Base to form the address of the handler in memory.
20:31 (52:63)	—	Reserved[1]

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

5.6 Interrupt Vector Offset Values (IVORxx)

Interrupt Vector Offset Registers are not implemented in e200z1. Instead, hardwired values are used for interrupt offsets to the IVPR contents during interrupt processing for determining the starting address of a software handler used to handle an interrupt. The associated hardwired value of the Interrupt Vector Offset 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 as shown in Figure 5-5.

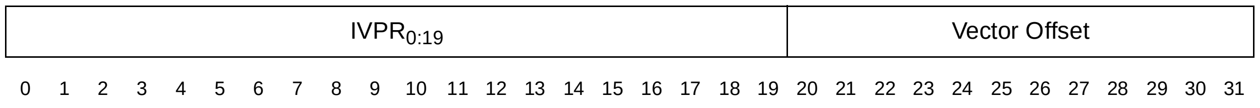


Figure 5-5. e200 Interrupt Vector Addresses

The hardwired Vector Offsets are listed in Table 5-7.

Table 5-7. Hardwired Vector Offset Values

Interrupt Type	Corresponding Interrupt Vector Offset Register	12-Bit Hex Offset
Critical Input	IVOR 0[1]	0x000
Machine check	IVOR 1	0x010
Data Storage	IVOR 2	0x020
Instruction Storage	IVOR 3	0x030

Table 5-7. Hardwired Vector Offset Values (continued)

Interrupt Type	Corresponding Interrupt Vector Offset Register	12-Bit Hex Offset
External Input	IVOR 4[1]	0x040
Alignment	IVOR 5	0x050
Program	IVOR 6	0x060
Floating-point unavailable	IVOR 7	0x070
System call	IVOR 8	0x080
AP unavailable	IVOR 9	Unused
Decrementer	IVOR 10	0x0A0
Fixed Interval Timer	IVOR 11	0x0B0
Watchdog Timer	IVOR 12	0x0C0
Data TLB Error	IVOR 13	0x0D0
Instruction TLB Error	IVOR 14	0x0E0
Debug	IVOR 15	0x0F0

[1] Autovectored External and Critical Input interrupts use this IVOR. Vectored interrupts supply an interrupt vector offset directly.

5.7 Interrupt Definitions

5.7.1 Critical Input Interrupt (IVOR0)

A Critical Input exception is signalled to the processor by the assertion of the critical interrupt pin (*p_critint_b*). When e200 detects the exception, if the exception is enabled by MSR[CE], e200 takes the Critical Input interrupt. The *p_critint_b* input is a level-sensitive signal expected to remain asserted until e200 acknowledges the interrupt. If *p_critint_b* is negated early, recognition of the interrupt request is not guaranteed. After e200 begins execution of the critical interrupt handler, the system can safely negate *p_critint_b*.

A Critical Input interrupt may be delayed by other higher priority exceptions or if MSR[CE] is cleared when the exception occurs.

Table 5-8 lists register settings when a Critical Input interrupt is taken.

Table 5-8. Critical Input Interrupt—Register Settings

Register	Setting Description
CSRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present.
CSRR1	Set to the contents of the MSR at the time of the interrupt

Table 5-8. Critical Input Interrupt—Register Settings (continued)

Register	Setting Description		
MSR	UCLE 0 WE 0 CE 0 EE 0 PR 0	FP 0 ME — FE0 0 DE —/0[1]	FE1 0 IS 0 DS
ESR	Unchanged		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	$IVPR_{0:19}$ || 12'h000 (autovectored) $IVPR_{0:19}$ || *p_voffset*[0:9] || 2'b00 (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 used by autovectored Critical Input interrupts to determine the interrupt handler location. e200 also provides the capability to directly vector Critical Input interrupts to multiple handlers by allowing a Critical Input interrupt request to be accompanied by a vector offset. The *p_voffset*[0:9] input signals are appended with 2'b00 and used in place of the IVOR0 value to form the interrupt vector when a Critical Input interrupt request is not autovectored (*p_avec_b* negated when *p_critint_b* asserted).

5.7.2 Machine Check Interrupt (IVOR1)

e200 implements the Machine Check exception as defined in Power Architecture Book E except for automatic clearing of the MSR[DE] bit (see later paragraph). e200 initiates a Machine Check interrupt if MSR[ME]=1 and any of the machine check sources listed in Table 5-2 is detected. As defined in Power Architecture Book E, the interrupt is not taken if MSR[ME] is cleared, in which case the processor generates an internal checkstop condition and enters the checkstop state. When a processor is in the checkstop state, instruction processing is suspended and generally cannot continue without restarting the processor. Note that other conditions may lead to the checkstop condition; the disabled machine check exception is only one of these.

e200 implements the Machine Check Syndrome register (MCSR) to record the source(s) of machine checks.

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)."

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

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

Table 5-9. Machine Check Interrupt—Register Settings

Register	Setting Description		
CSRR0	On a best-effort basis e200 sets this to the address of some instruction that was executing or about to be executing when the machine check condition occurred.		
CSRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 WE 0 CE 0 EE 0 PR 0	FP 0 ME 0 FE0 0 DE 0/—[1]	FE1 0 IS 0 DS 0
ESR	Unchanged		
MCSR	Updated to reflect the source(s) of a machine check		
DEAR	Unchanged		
Vector	IVPR$_{0:19}$ || 12’h010		

[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 input pin *p_mcp_b* can be masked by HID0[EMCP].

In general, most Machine Check exceptions are unrecoverable in the sense that execution cannot resume in the context that existed before the interrupt; however, system software can use the machine check interrupt handler to try to identify and recover from the machine check condition.

The Machine Check Syndrome register is provided to identify the source(s) of a machine check and may be used to identify recoverable events.

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

5.7.2.2 Checkstop State

A checkstop condition can occur for several reasons. The exception related conditions are:

- MSR[ME]=0 and a machine check occurs.
- First instruction in an interrupt handler can not be executed due to a translation miss (ITLB), page marked No Execute (ISI), or a bus error termination, and MSR[ME]=0.
- Bus error termination for a buffered store and MSR[ME]=0.
- Precise external termination error and MSR[EE]=0 and MSR[ME]=0

Non-exception related checkstop conditions are:

- TCR[WRC] - Watchdog Reset Control bits set to checkstop on second Watchdog Timer overflow event.

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

When a debug request is presented to the e200 core while in the checkstop state, the *p_wakeup* signal is asserted, and when *m_clk* is provided to the CPU, it temporarily exits the checkstop state and enters Debug mode. The *p_chkstop* output is negated for the duration of the time the CPU remains in a debug session (*p_debug_b* asserted). When the debug session is exited, the CPU re-enters the checkstop state. Note that the external system logic may be in an undefined state following a checkstop condition, such as having an outstanding bus transaction, or other inconsistency, thus no guarantee can be made in general about activities performed in debug mode while a checkstop is still outstanding. Debug logic does have the capability of generating assertion of the *p_resetout_b* signal via the DBCR0 register though.

5.7.3 Data Storage Interrupt (IVOR2)

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

- Read or Write Access Control exception condition
- Byte Ordering exception condition
- External Termination Error (precise) and MSR[EE]=1

Access control is defined as in Power Architecture Book E. A Byte Ordering exception condition occurs for any misaligned access across a page boundary to pages with mismatched E bits. Precise external termination errors occur when a load or store is terminated by assertion of *p_d_tea_b* (ERROR termination response). See Section 7.2, “Internal Interface Signals,” for a definition of internal signals.

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

Table 5-10. Data Storage Interrupt—Register Settings

Register	Setting Description		
SRR0	Set to the effective address of the excepting load/store instruction.		
SRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 WE 0 CE — EE 0 PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0
ESR	Access: Byte ordering: External Termination Error (Precise):	[ST], [VLEMI]. All other bits cleared. [ST], [VLEMI], BO. All other bits cleared. [ST], [VLEMI], XTE. All other bits cleared.	
MCSR	Unchanged		
DEAR	For Access and Byte ordering exceptions, set to the effective address of a byte within the page whose access caused the violation.		
Vector	$IVPR_{0:19}$ || 12’h020		

5.7.4 Instruction Storage Interrupt (IVOR3)

An Instruction Storage interrupt (ISI) occurs when no higher priority exception exists and an Execute Access Control exception occurs. This interrupt is implemented as defined by Power Architecture Book E., with the addition of precise external termination errors, which occur when an instruction fetch is terminated by assertion of *p_i_tea_b* (ERROR termination response) and MSR[EE]=1, Misaligned Instruction Fetch exceptions, and the extension of the Byte Ordering exception status to also cover Mismatched Instruction Storage exceptions. See Section 7.2, "Internal Interface Signals," for a definition of internal signals.

Exception extensions implemented in e200 for Power Architecture 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 5-11 and Table 5-12.

Table 5-11. ISI Exceptions and Conditions

Interrupt Type	Interrupt Vector Offset Register	Causing Conditions
Instruction Storage	IVOR 3	1. Access control. 2. Precise external termination error (*p_i_tea_b* assertion and precise recognition) and MSR[EE]=1. See Section 7.2, "Internal Interface Signals," for a definition of internal signals. 3. 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. 4. Misaligned Instruction fetch due to a change of flow to an odd halfword instruction boundary on a BookE (non-VLE) instruction page

Table 5-12 lists register settings when an ISI is taken.

Table 5-12. 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 WE 0 CE — EE 0 PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0
ESR	[XTE, BO, MIF, VLEMI]. All other bits cleared.		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	$\text{IVPR}_{0:19}$ || 12'h030		

5.7.5 External Input Interrupt (IVOR4)

An External Input exception is signalled to the processor by the assertion of the external interrupt pin (*p_extint_b*). The *p_extint_b* input is a level-sensitive signal expected to remain asserted until e200 acknowledges the external interrupt. If *p_extint_b* is negated early, recognition of the interrupt request is not guaranteed. When e200 detects the exception, if the exception is enabled by MSR[EE], e200 takes the External Input interrupt.

An External Input interrupt may be delayed by other higher priority exceptions or if MSR[EE] is cleared when the exception occurs.

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

Table 5-13. 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 WE 0 CE — EE 0 PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0
ESR	Unchanged		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	IVPR$_{0:19}$ || 12’h040 (autovectored) IVPR$_{0:19}$ || *p_voffset*[0:9] || 2’b00 (non-autovectored)		

IVOR4 is the vector offset value used by autovectored External Input interrupts to determine the interrupt handler location. e200 also provides the capability to directly vector External Input interrupts to multiple handlers by allowing a External Input interrupt request to be accompanied by a vector offset. The *p_voffset*[0:9] input signals are appended with 2’b00 and used in place of the IVOR4 value when an External Input interrupt request is not autovectored (*p_avec_b* negated when *p_extint_b* asserted).

5.7.6 Alignment Interrupt (IVOR5)

e200 implements the Alignment Interrupt as defined by Power Architecture Book E. 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
- Execution of a **dcbz** instruction is attempted

Table 5-14 lists register settings when an alignment interrupt is taken.

Table 5-14. 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 WE 0 CE — EE 0 PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0
ESR	[ST], [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	$IVPR_{0:19}$ || 12’h050		

5.7.7 Program Interrupt (IVOR6)

e200 implements the Program Interrupt as defined by Power Architecture Book E. A program interrupt occurs when no higher priority exception exists and one or more of the following exception conditions defined in Power Architecture Book E occur:

- Illegal Instruction exception
- Privileged Instruction exception
- Trap exception
- Unimplemented Operation exception

e200 invokes an Illegal Instruction program exception on attempted execution of the following instructions:

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

e200 invokes a Privileged Instruction program exception on attempted execution of the following instructions when MSR[PR]=1 (user mode):

- A privileged instruction
- **mtspr** and **mfspr** instructions which specify a SPRN value with $SPRN_5$=1 (even if the SPR is undefined).

e200 invokes an Trap exception on execution of the **tw** and **twi** instructions if the trap conditions are met and the exception is not also enabled as a Debug interrupt.

e200 will invoke an Unimplemented Operation program exception on attempted execution of the instructions **lswi**, **lswx**, **stswi**, **stswx**, **mfapidi**, **mfdcrx**, **mtdcrx**, or on any Power Architecture Book E floating point instruction when MSR[FP]=1. All other defined or allocated instructions that are not implemented by e200 cause a illegal instruction program exception.

Table 5-15 lists register settings when a Program interrupt is taken.

Table 5-15. 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 WE 0 CE — EE 0 PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0
ESR	Illegal: Privileged: Trap: Unimplemented:	PIL, [VLEMI]. All other bits cleared. PPR, [VLEMI]. All other bits cleared. PTR, [VLEMI]. All other bits cleared. PUO, [FP], [VLEMI]. All other bits cleared.	
MCSR	Unchanged		
DEAR	Unchanged		
Vector	$IVPR_{0:19}$ || 12’h060		

5.7.8 Floating-Point Unavailable Interrupt (IVOR7)

The Floating-point Unavailable exception is implemented as defined in Power Architecture Book E. A Floating-point Unavailable interrupt occurs when no higher priority exception exists, an attempt is made to execute a floating-point instruction (including floating-point load, store, or move instructions), and the floating-point available bit in the MSR is disabled (MSR[FP]=0).

Table 5-16 lists register settings when a Floating-Point Unavailable interrupt is taken.

Table 5-16. Floating-Point Unavailable Interrupt—Register Settings

Register	Setting Description		
SRR0	Set to the effective address of the excepting instruction.		
SRR1	Set to the contents of the MSR at the time of the interrupt		
MSR	UCLE 0 WE 0 CE — EE 0 PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0
ESR	Unchanged		
MCSR	Unchanged		

Table 5-16. Floating-Point Unavailable Interrupt—Register Settings (continued)

DEAR	Unchanged
Vector	$IVPR_{0:19}$ || 3’h070

5.7.9 System Call Interrupt (IVOR8)

A System Call interrupt occurs when a System Call (**sc**, **se_sc**) instruction is executed and no higher priority exception exists.

Exception extensions implemented in e200 for Power Architecture VLE include modification of the System Call Interrupt definition to include updating the ESR.

Table 5-17 lists register settings when a System Call interrupt is taken.

Table 5-17. 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 WE 0 CE — EE 0 PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0
ESR	[VLEMI] All other bits cleared.		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	$IVPR_{0:19}$ || 12’h080		

5.7.10 Auxiliary Processor Unavailable Interrupt (IVOR9)

An Auxiliary Processor Unavailable exception is defined by Power Architecture Book E to occur when an attempt is made to execute an APU instruction which is implemented but configured as unavailable, and no higher priority exception condition exists.

e200 does not utilize this interrupt.

5.7.11 Decrementer Interrupt (IVOR10)

e200 implements the Decrementer exception. A Decrementer interrupt occurs when no higher priority exception exists, a Decrementer exception condition exists (TSR[DIS]=1), and the interrupt is enabled (both TCR[DIE] and MSR[EE]=1).

The Timer Status Register (TSR) holds the Decrementer interrupt bit set by the Timer facility when an exception is detected. Software must clear this bit in the interrupt handler to avoid repeated Decrementer interrupts.

Table 5-18 lists register settings when a Decrementer interrupt is taken.

Table 5-18. 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 WE 0 CE Unchanged EE 0 PR 0	FP 0 ME Unchanged FE0 0 DE Unchanged	FE1 0 IS 0 DS 0
ESR	Unchanged		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	$IVPR_{0:19}$ || 3'h0A0		

5.7.12 Fixed-Interval Timer Interrupt (IVOR11)

e200 implements the Fixed-Interval Timer (FIT). The triggering of the exception is caused by selected bits in the Time Base register changing from 0 to 1.

A Fixed-Interval Timer interrupt occurs when no higher priority exception exists, a FIT exception exists (TSR[FIS]=1), and the interrupt is enabled (both TCR[FIE] and MSR[EE]=1).

The Timer Status Register (TSR) holds the FIT interrupt bit set by the Timer facility when an exception is detected. Software must clear this bit in the interrupt handler to avoid repeated FIT interrupts.

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

Table 5-19. 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 WE 0 CE — EE 0 PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0
ESR	Unchanged		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	$IVPR_{0:19}$ || 3'h0B0		

5.7.13 Watchdog Timer Interrupt (IVOR12)

e200 implements the Watchdog Timer (WDT) exception. The triggering of the exception is caused by the first enabled watchdog time-out.

A Watchdog Timer interrupt occurs when no higher priority exception exists, a Watchdog Timer exception exists (TSR[WIS]=1), and the interrupt is enabled (both TCR[WIE] and MSR[CE]=1).

The Timer Status Register (TSR) holds the Watchdog interrupt bit set by the Timer facility when an exception is detected. Software must clear this bit in the interrupt handler to avoid repeated Watchdog interrupts.

Table 5-20 lists register settings when a Watchdog Timer interrupt is taken.

Table 5-20. 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 WE 0 CE 0 EE 0 PR 0	FP 0 ME — FE0 0 DE 0/—[1]	FE1 0 IS 0 DS 0
ESR	Unchanged		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	IVPR$_{0:19}$ || 3’h0C0		

[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).

5.7.14 Data TLB Error Interrupt (IVOR13)

A Data TLB Error interrupt occurs when no higher priority exception exists and a Data TLB Error exception exists due to a data translation lookup miss in the TLB.

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

Table 5-21. 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 WE 0 CE — EE 0 PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0
ESR	[ST], [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$_{0:19}$ || 3'h0D0		

5.7.15 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 e200 for Power Architecture VLE involve extending the definition of the Instruction TLB Error Interrupt to include updating the ESR.

Table 5-22 lists register settings when an ITLB interrupt is taken.

Table 5-22. 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 WE 0 CE — EE 0 PR 0	FP 0 ME — FE0 0 DE —	FE1 0 IS 0 DS 0
ESR	[MIF] All other bits cleared.		
MCSR	Unchanged		
DEAR	Unchanged		
Vector	IVPR$_{0:19}$ || 3'h0E0		

5.7.16 Debug Interrupt (IVOR15)

e200 implements the Debug Interrupt as defined in Power Architecture Book E 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, 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 9, "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.

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 signalled when a branch instruction is considered taken by the branch unit and branch taken events are enabled. The Debug interrupt is taken when no higher priority exception is pending.

A Data Address Compare (DAC) exception is signalled when there is a data access address match as defined by the debug control registers and Data Address Compare events are enabled. This could either be a direct data address match or a selected set of data addresses. The Debug interrupt is taken when no higher priority exception is pending.

e200 does not implement the Data Value Compare debug mode, specified in Power Architecture Book E.

The e200 implementation provides IAC linked with DAC exceptions. This results in a DAC exception only if one or more IAC conditions are also met. See Chapter 9, "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 is taken instead of a Trap type Program Interrupt. The Debug interrupt is taken when no higher priority exception is pending. If MSR[DE] is cleared when a trap debug exception occurs, a Trap exception type Program interrupt occurs 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, se_rfci, rfdi,** or **se_rfdi** instructions. If MSR[DE]=1 at the time of the execution of the **rfi** or **se_rfi**, a Debug interrupt occurs provided there exists no higher priority exception which is enabled to cause an interrupt. CSRR0 (Debug APU disabled) or DSRR0 (Debug APU enabled) is set to the address of the **rfi** or **se_rfi** instruction. If MSR[DE]=0 at the time of the execution of the **rfi** or **se_rfi**, a Debug interrupt does not occur immediately, but the event is recorded by setting the DBSR[RET] and DBSR[IDE] status bits.

A Critical Return (CRET) debug exception occurs when executing an **rfci** or **se_rfci** instruction and critical return debug events are enabled. Critical return debug exceptions are only generated for **rfci** or **se_rfci** instructions. If MSR[DE]=1 at the time of the execution of the **rfci** or **se_rfci**, a Debug interrupt occurs provided there exists no higher priority exception which is enabled to cause an interrupt. CSRR0 (Debug APU disabled) or DSRR0 (Debug APU enabled) is set to the address of the **rfci** or **se_rfci** instruction. If MSR[DE]=0 at the time of the execution of the **rfci** or **se_rfci**, a Debug interrupt does not occur immediately, but the event is recorded by setting the DBSR[CRET] and DBSR[IDE] status bits. Note that critical return debug events should not normally be enabled unless the Debug APU is enabled to avoid corruption of CSRR0/1.

An Instruction Complete (ICMP) debug exception is signalled following execution and completion of an instruction while this event is enabled.

A **mtmsr** or **mtdbcr0** which 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 only occurs when detecting a non-critical interrupt on e200. The value saved in CSRR0/DSRR0 is 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 only occurs when detecting a critical interrupt on e200. The value saved in CSRR0/DSRR0 is the address of the critical interrupt handler. Note that Critical Interrupt Taken debug events should not normally be enabled unless the Debug 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 9, “Debug Support.”

Table 5-23 lists register settings when a Debug interrupt is taken.

Table 5-23. Debug Interrupt—Register Settings

Register	Setting Description		
CSRR0/ 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 WE 0 CE —/0[2] EE —/0[2] PR 0	FP 0 ME — FE0 0 DE 0	FE1 0 IS 0 DS 0
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	$\text{IVPR}_{0:19}$ || 12’h0F0		

[1] assumes that the Debug interrupt is precise

[2] conditional based on control bits in HID0. If HID0[DAPUEN] = 1, RI is unaffected because DSRR0/1 are used, otherwise it is cleared because CSRR0/1 are updated.

[3] Note that multiple DBSR bits may be set

5.7.17 System Reset Interrupt

e200 implements the System Reset interrupt as defined in Power Architecture Book E. The System Reset exception is a non-maskable, asynchronous exception signalled to the processor through the assertion of system-defined signals.

A System reset may be initiated by either asserting the *p_reset_b* input signal, during power-on reset by asserting *m_por*, by Watchdog Timer Reset Control, or by Debug Reset Control. 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. e200 internal state after a reset is defined in Section 2.5.4, "Reset Settings."

For reset initiated by Watchdog Timer or Debug Reset Control, e200 implements TSR[WRS] and DBSR[MRR] respectively, to aid software in determining the cause. Watchdog Timer and Debug Reset Control provides the capability to assert the *p_resetout_b* signal. External logic may factor this signal into the *p_reset_b* input signal to cause a e200 reset to occur.

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

Table 5-24. TSR Watchdog Timer Reset Status

Bit(s)	Name	Function
2:3 (34:35)	WRS	00 No action performed by Watchdog Timer 01 Watchdog Timer second time-out caused checkstop 10 Watchdog Timer second time-out caused *p_resetout_b* to be asserted 11 Reserved

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

Table 5-25. DBSR Most Recent Reset

Bit(s)	Name	Function
2:3 (34:35)	MRR	00 No reset occurred because these bits were last cleared by software 01 A reset occurred because these bits were last cleared by software 10 Reserved 11 Reserved

Table 5-26 lists register settings when a System Reset interrupt is taken.

Table 5-26. System Reset Interrupt—Register Settings

Register	Setting Description		
CSRR0	Undefined.		
CSRR1	Undefined.		
MSR	UCLE 0 WE 0 CE 0 EE 0 PR 0	FP 0 ME 0 FE0 0 DE 0	FE1 0 IS 0 DS 0
ESR	Cleared		
DEAR	Undefined		
Vector	[*p_rstbase*[0:1929]] || 2'b00		

5.8 Exception Recognition and Priorities

The following list of exception categories describes how e200 handles exceptions up to the point of signaling the appropriate interrupt to occur. Also, instruction completion is defined as updating all architectural registers associated with that instruction as necessary, and then removing the instruction from the pipeline.

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

    System reset by assertion of *p_reset_b*

    Has highest priority and is taken immediately regardless of other pending exceptions or recoverability. (Includes Watchdog Timer Reset Control and Debug Reset Control)
  - Asynchronous, maskable, possibly 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. Typically unrecoverable.
  - Asynchronous, maskable, recoverable:

    External Input, Fixed-Interval Timer, Decrementer, Critical Input, Unconditional Debug, External Debug Event, Debug Counter Event, and Watchdog Timer interrupts

    Before handling this type of exception, the processor needs to reach a recoverable state. A maskable recoverable exception remains pending until taken or cancelled by software.
- Synchronous, non instruction-based interrupts. The only exception 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 is in a recoverable state due to the state of the machine at the context switch triggering this event.
- Instruction-based interrupts. These interrupts are further organized by the point in instruction processing in which they generate an exception.
  - Instruction Fetch:

    Instruction Storage, Instruction TLB, and Instruction Address Compare debug exceptions.

    Once these types of exceptions are detected, the excepting instruction is tagged. When the excepting instruction is next to begin execution and a recoverable state has been reached, the interrupt is taken. If an event prior to the excepting instruction causes a redirection of execution, the instruction fetch exception is discarded (but may be encountered again).
  - Instruction Dispatch/Execution:

    Program, System Call, Data Storage, Alignment, Floating-point Unavailable, Data TLB, Debug (Trap, Branch Taken, Ret) interrupts.

    These types of exceptions are determined during decode or execution of an instruction. The exception remains pending until all instructions before the exception causing instruction in program order complete. The interrupt is then taken without completing the exception-causing instruction. If completing previous instructions causes an exception, that exception takes priority over the pending instruction dispatch/execution exception, which is discarded (but may be encountered again when instruction processing resumes).

— Post-Instruction Execution

Debug (Data Address Compare, Instruction Complete) interrupt.

These Debug exceptions are generated following execution and completion of an instruction while the event is enabled. If executing the instruction produces conditions for another type of exception with higher priority, that exception is taken and the post-instruction exception is discarded for the instruction (but may be encountered again when instruction processing resumes)

5.8.1 Exception Priorities

Exceptions are prioritized as described in Table 5-27. Some exceptions may be masked or imprecise, which 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 most machine checks are non-recoverable.

Table 5-27. e200 Exception Priorities

Priority	Exception	Cause	IVOR
Asynchronous Exceptions			
0	System reset	Assertion of *p_reset_b*, Watchdog Timer Reset Control, or Debug Reset Control	none
1	Machine check	Assertion of *p_mcp_b*, exception on fetch of first instruction of an interrupt handler, bus error on buffered store, Bus error (XTE) with MSR[EE]=0 and current MSR[ME]=1	1
2	—	—	—
3[1]	Debug: 1. UDE 2. DEVT1 3. DEVT2 4. DCNT1 5. DCNT2 6. IDE	1. Assertion of *p_ude* (Unconditional Debug Event) 2. Assertion of *p_devt1* and event enabled (External Debug Event 1) 3. Assertion of *p_devt2* and event enabled (External Debug Event 2) 4. Debug Counter 1 exception 5. Debug Counter 2 exception 6. Imprecise Debug Event (event imprecise due to previous higher priority interrupt	15
4[1]	Critical Input	Assertion of *p_critint_b*	0
5[1,]	Watchdog Timer	Watchdog Timer first enabled time-out	12
6[1]	External Input	Assertion of *p_extint_b*	4
7[1,]	Fixed-Interval Timer	Posting of a FIT exception in TSR due to programmer-specified bit transition in the Time Base register	11
8[1,]	Decrementer	Posting of a Decrementer exception in TSR due to programmer-specified Decrementer condition	10

Table 5-27. e200 Exception Priorities (continued)

Priority	Exception	Cause	IVOR
Instruction Fetch Exceptions			
9	Debug: IAC (unlinked)	Instruction address compare match for enabled IAC debug event and DBCR0[IDM] asserted	15
10	ITLB Error	Instruction translation lookup miss in the TLB	14
11	Instruction Storage	1. Access control. 2. Byte ordering due to misaligned instruction across page boundary to pages with mismatched VLE bits, or access to page with VLE set, and E indicating little-endian. 3. Misaligned Instruction fetch due to a change of flow to an odd halfword instruction boundary on a BookE (non-VLE) instruction page, due to value in LR, CTR, or xSRR0 4. Precise external termination error (*p_tea_b* assertion and precise recognition) and MSR[EE]=1	3
Instruction Dispatch/Execution Interrupts			
12	Program: Illegal	Attempted execution of an illegal instruction.	6
13	Program: Privileged	Attempted execution of a privileged instruction in user-mode	6
14	Floating-point Unavailable	Any floating-point unavailable exception condition.	7
15	Program: Unimplemented	Attempted execution of an unimplemented instruction.	6
16	Debug: 1. BRT 2. Trap 3. RET 4. CRET	1. Attempted execution of a taken branch instruction 2. Condition specified in **tw** or **twi** instruction met. 3. Attempted execution of a **rfi** or **se_rfi** instruction. 4. Attempted execution of an **rfci** or **se_rfci** instruction. **Note**: Exceptions requires corresponding debug event enabled, MSR[DE]=1, and DBCR0[IDM]=1.	15
17	Program: Trap	Condition specified in **tw** or **twi** instruction met and not trap debug.	15
	System Call	Execution of the System Call (**sc, se_sc**) instruction.	8
18	Alignment	**lmw**, **stmw, lwarx,** or **stwcx.** not word aligned. **dcbz** with cache disabled or not present	5
19	Debug with concurrent DTLB or DSI exception: 1. DAC/IAC linked[2] 2. DAC unlinked[2]	Debug with concurrent DTLB or DSI exception. DBSR[IDE] also set. 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] is set. Saved PC points to the load or store instruction causing the DAC event. MMU MAS registers are updated due to the DTLB event	15
20	Data TLB Error	Data translation lookup miss in the TLB.	13

Table 5-27. e200 Exception Priorities (continued)

Priority	Exception	Cause	IVOR
21	Data Storage	1. Access control. 2. Byte ordering due to misaligned access across page boundary to pages with mismatched E bits. 3. Precise external termination error (*p_tea_b* assertion and precise recognition) and MSR[EE]=1	2
22[3]	Alignment	**dcbz** to W=1 or I=1 storage with cache enabled	5
23	Debug: 1. IRPT 2. CIRPT	1. Interrupt taken (non-critical) 2. Critical Interrupt taken (critical only) **Note**: Exceptions requires corresponding debug event enabled, MSR[DE]=1, and DBCR0[IDM]=1.	15
Post-Instruction Execution Exceptions			
24	Debug: 1. DAC/IAC linked[2] 2. DAC unlinked[2]	1. Data Address Compare linked with Instruction Address Compare 2. Data Address Compare unlinked **Notes**: Exceptions requires corresponding debug event enabled, MSR[DE]=1, and DBCR0[IDM]=1. Saved PC points to the instruction following the load or store instruction causing the DAC event.	15
25	Debug: 1. ICMP	1. Completion of an instruction. **Note**: Exceptions requires corresponding debug event enabled, MSR[DE]=1, and DBCR0[IDM]=1.	15

[1] These exceptions are sampled at instruction boundaries, thus may actually occur after exceptions which are due to a currently executing instruction. If one of these exceptions occurs during execution of an instruction in the pipeline, it is not processed until the pipeline has been flushed, and the exception associated with the excepting instruction may occur first.

[2] When no Data Storage Interrupt or Data TLB Error occurs, e200 implements the data address compare debug exceptions as post-instruction exceptions which differs from the Power Architecture Book E definition. When a TEA (either a DTLB error or DSI, or a Machine Check (if MSR[EE]=0)) occurs in conjunction with an enabled DAC or linked DAC/IAC on a load or store class instruction, the Debug Interrupt takes priority, and the saved PC value points to the load or store class instruction, rather than to the next instruction. In addition, the MMU MAS registers are updated due to the DTLB event.

[3] Unused on cacheless cores

5.9 Interrupt Processing

When an interrupt is taken, the processor uses SRR0/SRR1 for non-critical interrupts, CSRR0/CSRR1 for critical and 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 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** or **se_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** or **se_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** or **se_rfdi** instruction is executed.

The Exception Syndrome register is loaded with information specific to the exception type. Some interrupt types can only be caused by a single exception type, and thus do not use an ESR setting to indicate the interrupt cause.

The Machine State register is updated to preclude unrecoverable interrupts from occurring during the initial portion of the interrupt handler. Specific settings are described in Table 5-28.

For Alignment, Data TLB, or Data Storage interrupts, the Data Exception Address Register (DEAR) is loaded with the address which caused the interrupt to occur.

For Machine Check interrupts, the Machine Check Syndrome register is loaded with information specific to the exception type.

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) value specific for each type of interrupt (see Table 5-2).

Table 5-28 shows the MSR settings for different interrupt categories.

Table 5-28. MSR Setting Due to Interrupt

Bit(s)	MSR Definition	Reset Setting	Non-Critical Interrupt	Critical Interrupt	Debug Interrupt
5 (37)	UCLE	0	0	0	0
13 (45)	WE	0	0	0	0
14 (46)	CE	0	—	0	—/0[1]
16 (48)	EE	0	0	0	—/0[1]
17 (49)	PR	0	0	0	0
18 (50)	FP	0	0	0	0
19 (51)	ME	0	—	—	—
20 (52)	FE0	0	0	0	0
22 (54)	DE	0	—	—/0[1]	0
23 (55)	FE1	0	0	0	0
26 (58)	IS	0	0	0	0
27 (59)	DS	0	0	0	0

Reserved and preserved bits are unimplemented and read as 0.

[1] Conditionally cleared based on control bits in HID0

5.9.1 Enabling and Disabling Exceptions

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

- System reset exceptions cannot be masked.
- A machine check exception can occur only if the machine check enable bit (MSR[ME]) is set. If MSR[ME] is cleared, the processor goes directly into checkstop state when a machine check exception condition occurs. Individual machine check exceptions can be enabled and disabled through 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 9, “Debug Support,” for more details on individual control of debug exceptions.
- The Floating Point unavailable exception can be prevented by setting MSR[FP] (although an unimplemented instruction exception will be generated by e200 instead).

5.9.2 Returning from an Interrupt Handler

The return from interrupt (**rfi, se_rfi**), return from critical interrupt (**rfci, se_rfci**) and return from debug interrupt (**rfdi, se_rfdi**) instructions perform context synchronization by allowing previously-issued instructions to complete before returning to the interrupted process. In general, execution of a return instruction 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** or **se_rfi** instruction copies SRR1 bits back into the MSR.
- The **rfci** or **se_rfci** instruction copies CSRR1 bits back into the MSR.
- The **rfdi** or **se_rfdi** instruction copies DSRR1 bits back into the MSR.
- Instructions fetched after this instruction execute in the context established by this instruction.
- Program execution resumes at the instruction indicated by SRR0 for **rfi** or **se_rfi**, CSRR0 for **rfci** or **se_rfci** and DSRR0 for **rfdi** or **se_rfdi**.

Note that the return instruction **rfi** or **se_rfi** may be subject to a Return type debug exception, and that the return from critical interrupt instruction **rfci** or **se_rfci** may be subject to a Critical Return type debug exception. For a complete description of context synchronization, refer to Power Architecture Book E Specification.

5.10 Process Switching

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

- The **msync** instruction orders the effects of data memory instruction execution. All instructions previously initiated appear to have completed before the **msync** instruction completes, and no subsequent instructions appear to be initiated until the **msync** instruction completes.
- The **isync** instruction waits for all previous instructions to complete and then discards any fetched instructions, causing subsequent instructions to be fetched (or refetched) from memory and to execute in the context (privilege, translation, and protection) established by the previous instructions.
- The **stwcx.** instructions clears any outstanding reservations, ensuring that a load and reserve instruction in an old process is not paired with a store conditional instruction in a new one.

Chapter 6
Memory Management Unit

6.1 Overview

The e200z1 Memory Management Unit is a 32-bit Power Architecture Book E compliant implementation, with the following feature set:

- MMU architecture compliant with Freescale EIS as documented in the *EREF: A Programmer's Reference Manual for Freescale Book E Processors*
- Translates from 32-bit effective to 32-bit real addresses
- 8-entry fully associative TLB with support for eleven page sizes (4 Kbytes, 16 Kbytes, 64 Kbytes, 256 Kbytes, 1 Mbyte, 4 Mbytes, 16 Mbytes, 64 Mbytes, 256 Mbytes, 1 Gbyte, 4 Gbytes)
- Hardware assist for TLB miss exceptions
- Software managed by **tlbre**, **tlbwe**, **tlbsx**, **tlbsync**, and **tlbivax** instructions

6.2 Effective to Real Address Translation

6.2.1 Effective Addresses

Instruction accesses are generated by sequential instruction fetches or due to a change in program flow (branches and interrupts). Data accesses are generated by load, store, and cache management instructions. The e200 instruction fetch, branch, and load/store units generate 32-bit effective addresses. The MMU translates this effective address to a 32-bit real address which is then used for memory accesses.

The Power Architecture Book E architecture divides the effective (virtual) and real (physical) address space into pages. The page represents the granularity of effective address translation, permission control, and memory/cache attributes. The MMU supports eleven page sizes (4 Kbytes, 16 Kbytes, 64 Kbytes, 256 Kbytes, 1 Mbyte, 4 Mbytes, 16 Mbytes, 64 Mbytes, 256 Mbytes, 1 Gbyte, 4 Gbytes). 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.

6.2.2 Address Spaces

Instruction accesses are generated by sequential instruction fetches or due to a change in program flow (branches and interrupts). Data accesses are generated by load, store, and cache management instructions.

The Power Architecture Book E architecture defines two effective address spaces for instruction accesses and two effective address spaces for data accesses. The current effective address space for instruction or data accesses is determined by the value of MSR[IS] 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_[d,i]_tc[1]** interface signals indicates the appropriate address space. See Section 7.3, “Signal Descriptions,” for a definition of internal interface signals.

6.2.3 Process ID

The Power Architecture Book E 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 e200 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.

6.2.4 Translation Flow

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

Figure 6-1 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.

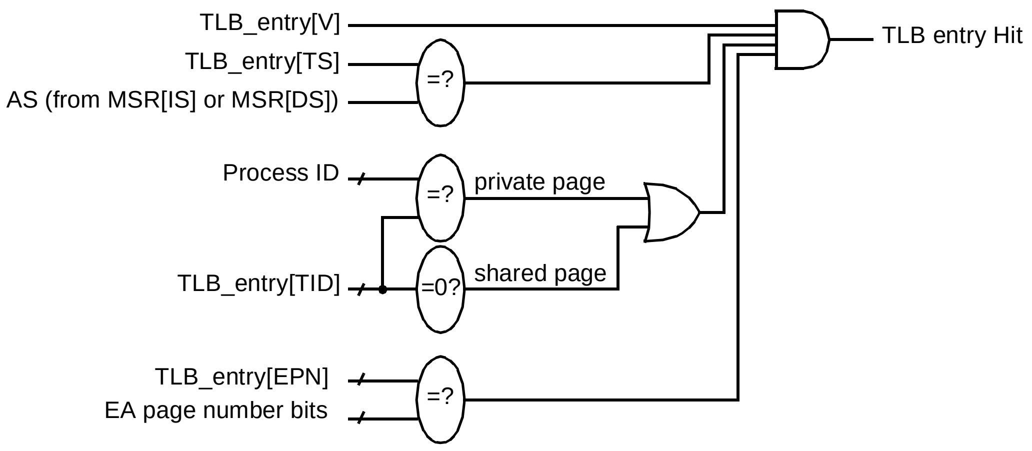


Figure 6-1. Virtual Address and TLB-Entry Compare Process

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 6-1. On a TLB hit, the corresponding bits of the Real Page Number (RPN) field are used to form the real address.

Table 6-1. Page Size and EPN Field Comparison

SIZE Field	Page Size (4^{SIZE}Kbytes)	EA to EPN Comparison
0b0001	4 Kbytes	EA[0:19] =? EPN[0:19]
0b0010	16 Kbytes	EA[0:17] =? EPN[0:17]
0b0011	64 Kbytes	EA[0:15] =? EPN[0:15]
0b0100	256 Kbytes	EA[0:13] =? EPN[0:13]
0b0101	1 Mbyte	EA[0:11] =? EPN[0:11]
0b0110	4 Mbytes	EA[0:9] =? EPN[0:9]
0b0111	16 Mbytes	EA[0:7] =? EPN[0:7]
0b1000	64 Mbytes	EA[0:5] =? EPN[0:5]
0b1001	256 Mbytes	EA[0:3] =? EPN[0:3]
0b1010	1 Gbytes	EA[0:1] =? EPN[0:1]
0b1011	4 Gbytes	—

On a TLB hit, the generation of the physical address occurs as shown in Figure 6-2.

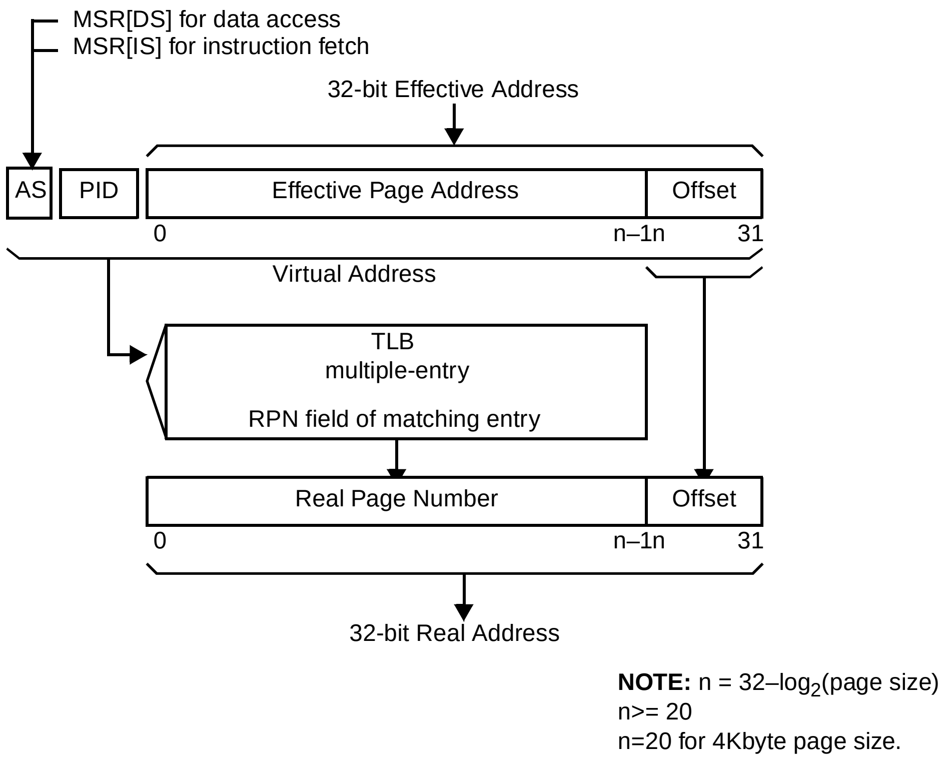


Figure 6-2. Effective to Real Address Translation Flow

Address mapping may be disabled via the $\text{MMUCSR0}_{\text{BYPASS}}$ control bit, which is sampled from the **p_addr_bypass** core input at system reset. When bypassing is enabled, the translation flow is still followed, and translation misses may still occur, but the real address is driven directly from the effective address. Protection and attribute information is not affected by address bypassing.

6.2.5 Permissions

An operating system may restrict access to virtual pages by selectively granting permissions for user mode read, write, and execute, and supervisor mode read, write, and execute on a per page basis. These permissions can be set up for a particular system (for example, program code 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 load instructions to access the page while in supervisor mode (MSR[PR=0]).
- SW—Supervisor write permission. Allows store 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 load instructions to access the page while in user mode (MSR[PR=1]).
- UW—User write permission. Allows store instructions to access the page while in user mode (MSR[PR=1]).
- UX—User execute permission. Allows instruction fetches to access the page and instructions to be executed from the page while in user mode (MSR[PR=1]).

If the translation match was successful, the permission bits are checked as shown in Figure 6-3. If the access is not allowed by the access permission mechanism, the processor generates an Instruction or Data Storage interrupt (ISI or DSI). The current privilege level of an access is signaled to the MMU with the CPU's **p_[d,i]tc[0]** output signals.

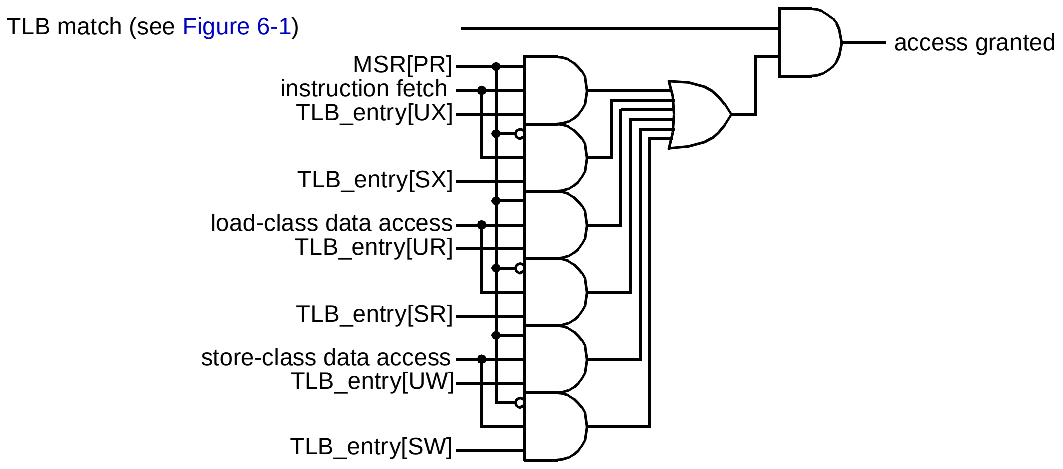


Figure 6-3. Granting of Access Permission

6.3 Translation Lookaside Buffer

The Freescale embedded implementation standards (EIS) architecture, documented in *EREF: A Programmer's Reference Manual for Freescale Book E Processors*, 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 e200z1 MMU supports a single TLB which is fully associative and supports variable page sizes, thus it corresponds to TLB1. For the rest of this document, TLB, TLBCAM, and TLB1 are used interchangeably.

The TLB consists of an 8-entry, fully associative CAM array with support for eleven 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 and TLB entries may be corrupted (an exception will not be reported).

Table 6-2. TLB Entry Bit Definitions

Field	Comments
V	Valid bit for entry
TS	Translation address space (compared against AS bit)
TID[0:7]	Translation ID (compared against PID0 or '0')

Table 6-2. TLB Entry Bit Definitions (continued)

Field	Comments
EPN[0:19]	Effective page number (compared against effective address)
RPN[0:19]	Real page number (translated address)
SIZE[0-3]	Page size (4-Kbyte/16-Kbyte/64-Kbyte/256-Kbyte /1-Mbyte/4-Mbyte/16-Mbyte/64-Mbyte/256-Mbyte/1-Gbyte/4-Gbyte)
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

6.4 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.

6.4.1 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 e200 MMU design. The MMUCFG register is shown in Figure 6-4.

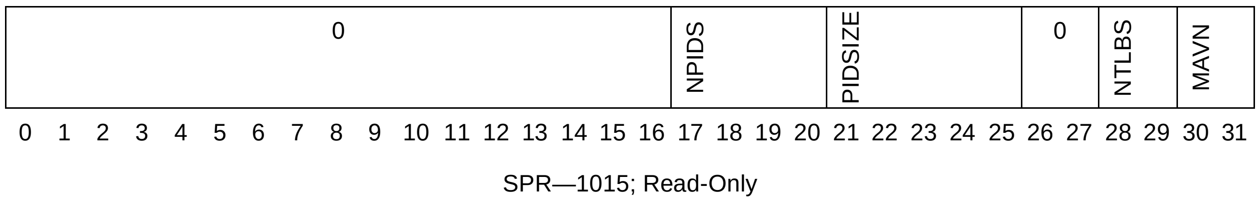


Figure 6-4. MMU Configuration Register (MMUCFG)

The MMUCFG bits are described in Table 6-3.

Table 6-3. MMUCFG Field Descriptions

Bits	Name	Function
0:16 [32:48]	—	Reserved[1]
17:20 [49:52]	NPIDS	Number of PID Registers 0001 - This version of the e200 MMU implements one PID register (PID0)

Table 6-3. MMUCFG Field Descriptions (continued)

Bits	Name	Function
21:25 [53:57]	PIDSIZE	PID Register Size 00111 - PID registers contain 8 bits in this version of the e200 MMU
26:27 [58:59]	—	Reserved[1]
28:29 [60:61]	NTLBS	Number of TLBs 01 - This version of the e200 MMU implements two TLB structures: a null TLB0 and a TLBCAM for TLB1
30:31 [62:63]	MAVN	MMU Architecture Version Number 00 - This version of the e200 MMU implements Version 1.0 of the EIS MMU Architecture. The EIS specifications can be found in *EREF: A Programmer's Reference Manual for Freescale Book E Processors*

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

6.4.2 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 e200 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 the EIS Book E specification, as found in *EREF: A Programmer's Reference Manual for Freescale Book E Processors*. The TLB0CFG register is shown in Figure 6-5.

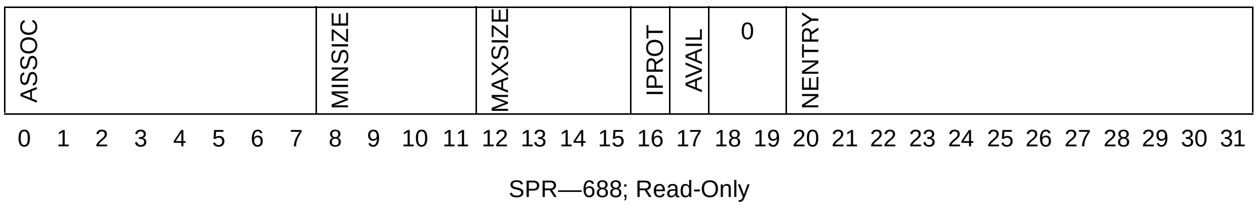


Figure 6-5. TLB0 Configuration Register (TLB0CFG)

The TLB0CFG bits are described in Table 6-4.

Table 6-4. 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

Table 6-4. TLB0CFG Field Descriptions (continued)

Bits	Name	Function
17 [49]	AVAIL	Page Size Availability 0
18:19 [50:51]	—	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.

6.4.3 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 e200 MMU. The TLB1CFG register is shown in Figure 6-6.

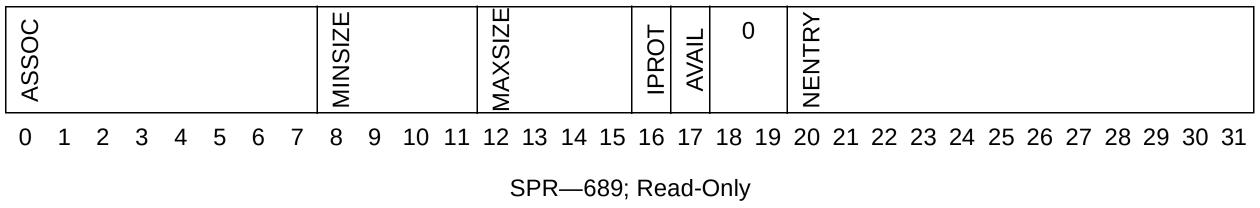


Figure 6-6. TLB1 Configuration Register (TLB1CFG)

The TLB1CFG bits are described in Table 6-5.

Table 6-5. TLB1CFG Field Descriptions

Bits	Name	Function
0:7 [32:39]	ASSOC	Associativity 0x8 - Indicates that TLB1 associativity is 8
8:11 [40:43]	MINSIZE	Minimum Page Size 0x1 - Smallest page size is 4K
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
18:19 [50:51]	—	Reserved[1]
20:31 [52:63]	NENTRY	Number of Entries 0x8 - TLB1 contains 8 entries

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

6.5 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, Data TLB Error, DSI, or ISI 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, the hardware updates only the search PID (SPID) and search address space (SAS) fields in the MAS registers using PID0, and appropriate MSR[IS] or MSR[DS] values which were used when the DSI or ISI exception was recognized. During the interrupt handler, software can issue a TLB search instruction (**tlbsx**), which uses the SPID field along with the SAS field, to determine the entry related to the DSI or ISI exception. (It is possible that the entry which caused the DSI or ISI interrupt no longer exists in the TLB by the time the search occurs if a TLB invalidate or replacement removes the entry between the time the exception is recognized and when the **tlbsx** is executed.)

The **tlbre**, **tlbwe**, **tlbsx**, **tlbivax**, and **tlbsync** instructions are privileged.

6.5.1 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 ESELCAM fields of the MAS0 register. The entry contents are placed in the MAS1, MAS2, and MAS3 registers. See Table 6-15 for details on how MAS register fields are updated.

tlbre **tlbre**

tlb read entry

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

```
tlb_entry_id = MAS0(TLBSEL, ESELCAM)
result = MMU(tlb_entry_id)
MAS1, MAS2, MAS3 = result
```

6.5.2 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 ESELCAM fields of the MAS0 register.

tlbwe **tlbwe**

tlb write entry

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

```
tlb_entry_id = MAS0(TLBSEL, ESELCAM)
MMU(tlb_entry_id) = MAS1, MAS2, MAS3
```

6.5.3 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 6-15 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 6-15, column labelled "tlbsx hit"
else result = see Table 6-15, column labelled "tlbsx miss"
MAS0, MAS1, MAS2, MAS3 = result
```

6.5.4 TLB Invalidate (tlbivax) Instruction

The TLB invalidate operation is performed whenever a TLB Invalidate Virtual Address Indexed (**tlbivax**) instruction is executed. This instruction invalidates TLB entries which correspond to the virtual address calculated by this instruction. The address is detailed in Table 6-6. No other information except for that shown in Table 6-6 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 the TLBCAM 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 the TLBCAM which are not marked as invalidation protected (IPROT bit of entry set to '1').

The bits of EA used to perform the **tlbivax** invalidation of the TLBCAM are bits 0:19.

Table 6-6. tlbivax EA Bit Definitions

Bits	Field
0:19	EA[0:19]
20:27	Reserved[1]
28	TLBSEL(1=TLBCAM) Should be set to '1' for future compatibility.
29	INV_ALL
30:31	Reserved[1]

[1] These bits should be zero for future compatibility. They are ignored.

tlbivax

TLB Invalidate Virtual Address Indexed

tlbivax RA,RB Form X

0–5	6–10	11–15	16–20	21–30	31
31	0	RA	RB	1 1 0 0 0 1 0 0 1 0	0

```
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
```

6.5.5 TLB Synchronize Instruction (tlbsync)

The TLB Synchronize instruction is treated as a privileged no-op by the e200z1.

tlbsync tlbsync

TLB Synchronize

tlbsync

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

6.6 TLB Operations

6.6.1 Translation Reload

The TLB reload function is performed in software with some hardware assist. This hardware assist consists of the following:

- Five 32-bit MMU assist registers (MAS0–4,MAS6) for support of the **tlbre**, **tlbwe**, and **tlbsx** TLB management instructions.
- Loading of MAS0–2 based upon defaults in MAS4 for TLB miss exceptions. This automatically generates most of the TLB entry.
- Loading of the data exception address register (DEAR) with the effective address of the load, store, or cache management instruction that caused an Alignment, Data TLB Miss, or Data Storage Interrupt.
- The **tlbwe** instruction. When **tlbwe** is executed, the new TLB entry contained in MAS0-MAS2 is written into the TLB.

6.6.2 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 ESELCAM 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.

6.6.3 Writing the TLB

The TLBCAM 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 ESELCAM 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.

6.6.4 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[0:19] from the GPR selected by the instruction, SAS (search

AS bit) in MAS6, and SPID in MAS6. If the search is successful, the given TLB entry information 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.

6.6.5 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[0:19] of the access that caused the exception. In addition, the ESELCAM bits are updated with the replacement entry value.

This sets up all the TLB entry data necessary for a TLB write except for the RPN[0:19], the U0-U3 user bits, and the UX/SX/UW/SW/UR/SR permission bits, all of which are stored in MAS3. Thus, if the defaults stored in MAS4 are applicable to the TLB entry to be loaded, the TLB miss exception handler 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.

6.6.6 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[TLBCAM_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.

6.6.7 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 6-7. 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[0:19]	value of **p_rstbase[0:19]**	Page address present on **p_rstbase[0:19]**. See Section 7.3.2.4, “Reset Base (p_rstbase[0:29]).”
RPN[0:19]	value of **p_rstbase[0:19]**	Page address present on **p_rstbase[0:19]**. See Section 7.3.2.4, “Reset Base (p_rstbase[0:29]).”
SIZE[0-3]	0001	4-Kbyte page size
SX/SW/SR	111	Full supervisor mode access allowed
UX/UW/UR	111	Full user mode access allowed
WIMG	0100	Cache inhibited, non-coherent

Table 6-7. TLB Entry 0 Values after Reset (continued)

Field	Reset Value	Comments
E	value of **p_rst_endmode**	Value present on **p_rst_endmode**. See Section 7.3.2.5, "Reset Endian Mode (p_rst_endmode)."
U0-U3	0000	User bits
IPROT	1	Page is protected from invalidation
VLE	the value of **p_rst_vlemode**	Value present on **p_rst_vlemode signal.** See Section 7.3.2.6, "Reset VLE Mode (p_rst_vlemode)."

6.6.8 The G Bit

The G bit provides protection from bus accesses which could be cancelled due to an exception on a prior uncompleted instruction.

If G = 1 (guarded), these types of accesses must stall until the exception status of the instruction(s) in progress is known. If G = 0 (unguarded), then these accesses may be issued to the bus regardless of the completion status of other instructions. Since the e200z1 does not make requests for load or store instructions until it is known that prior instructions will complete without exceptions, the G bit is essentially ignored. Proper operation will always occur to guarded storage.

6.7 MMU Control Registers

6.7.1 DEAR Register

The Data Exception Address register is loaded with the effective address of the data access which results in an Alignment, Data TLB Miss, or DSI exception.

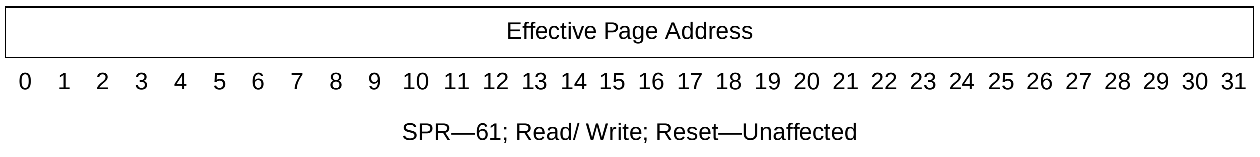
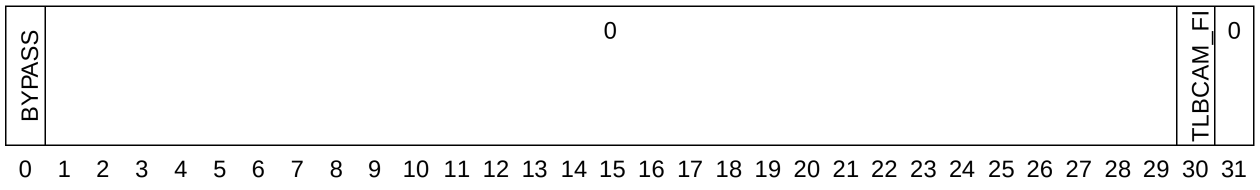


Figure 6-7. DEAR Register

The DEAR register can be read or written using the **mfspr** and **mtspr** instructions.

6.7.2 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 6-8.



SPR—1012; Read/ Write; Reset—p_notrans_tlb || $^{31}0$

Figure 6-8. MMU Control and Status Register 0 (MMUCSR0)

The MMUCSR0 bits are described in Table 6-8.

Table 6-8. MMUCSR0—MMU Control and Status Register 0

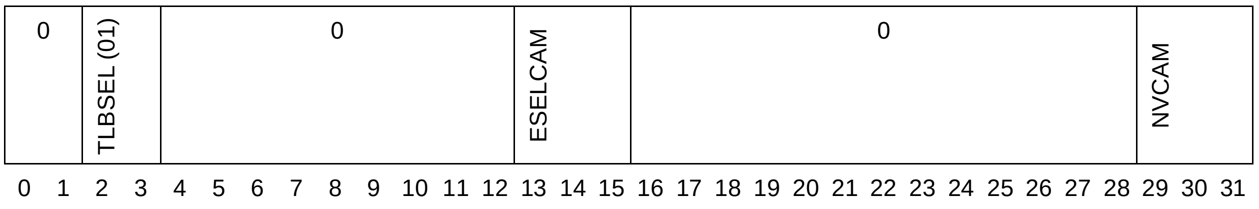
Bits	Name	Description
0 [32]	BYPASS	Bypass translation 0 - TLB translates effective to real addresses normally 1 - TLB bypasses the effective address directly to the real address When set to ‘1’, the TLB performs no address translation; instead a 1:1 address mapping is used. TLB lookups are still performed for page attribute and protection information, and TLB misses can still occur. ISI and DSI can still occur due to lack of access permissions. On reset negation, this bit is set to the value of the **p_addr_bypass** input.
1:29 [33:61]	—	Reserved[1]
30 [62]	TLBCAM_FI	TLBCAM flash invalidate 0 - No flash invalidate 1 - TLBCAM invalidation operation When written to a ‘1’, a TLBCAM 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. TLBCAM 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.

6.7.3 MMU Assist Registers (MAS)

e200 uses six special purpose registers (MAS0, MAS1, MAS2, MAS3, MAS4 and MAS6) to facilitate reading, writing, and searching the TLB. The MAS registers can be read or written using the **mfspr** and **mtspr** instructions. e200 does not implement the MAS5 register, present in other EIS Book E designs, because the **tlbsx** instruction only searches based on a single SPID value. The Book E specification can be found in as found in *EREF: A Programmer’s Reference Manual for Freescale Book E Processors*.

The MAS0 register is shown in Figure 6-9. Fields are defined in Table 6-9.



SPR—624; Read/ Write; Reset—Unaffected

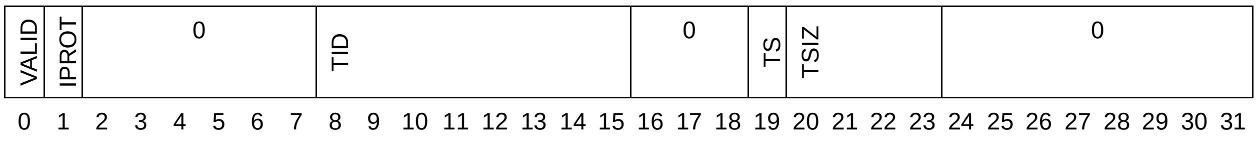
Figure 6-9. MMU Assist Register 0 (MAS0)

Table 6-9. MAS0—MMU Read/Write and Replacement Control

Bit	Name	Comments, or Function when Set
0:1 [32:33]	—	Reserved[1]
2:3 [34:35]	TLBSEL	selects TLB for access: 01=TLBCAM (ignored by Zen, should be written to 01 for future compatibility)
4:11 [36:42]	—	Reserved[1]
13:15 [44:47]	ESELCAM	Entry select for TLBCAM
16:27 [48:59]	—	Reserved[1]
29:31 [61:63]	NVCAM	Next replacement victim for TLBCAM (software managed) Software updates this field; it is copied to the ESELCAM field on a TLB Error (see Table 6-15)

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

The MAS1 register is shown in Figure 6-10. Fields are defined in Table 6-10.



SPR—625; Read/ Write; Reset—Unaffected

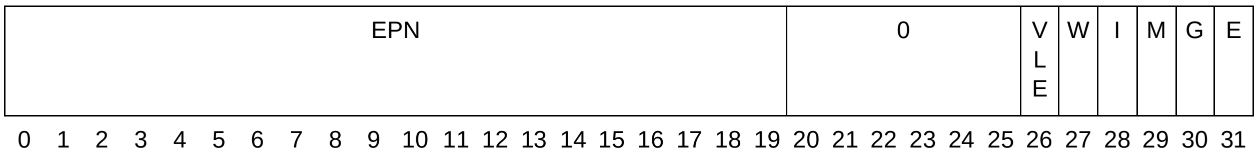
Figure 6-10. MMU Assist Register 1 (MAS1)

Table 6-10. MAS1—Descriptor Context and Configuration Control

Bit	Name	Comments, or Function when Set
0 [32]	VALID	TLB Entry Valid 0 - This TLB entry is invalid 1 - This TLB entry is valid
1 [33]	IPROT	Invalidation Protect 0 - Entry is not protected from invalidation 1 - Entry is protected from invalidation as described in Section 6.6.6, “IPROT Invalidation Protection.” Protects TLB entry from invalidation by **tlbivax** (TLBCAM only), or flash invalidates through MMUSCR0[TLBCAM_FI].
2:7 [34:39]	—	Reserved[1]
8:15 [40:47]	TID	Translation ID bits This field is compared with the current process IDs of the effective address to be translated. A TID value of 0 defines an entry as global and matches with all process IDs.
16:18 [48:50]	—	Reserved[1]
19 [51]	TS	Translation address space This bit is compared with the IS or DS fields of the MSR (depending on the type of access) to determine if this TLB entry may be used for translation.
20:23 [52:55]	TSIZE	Entry’s page size Supported page sizes are: TSIZE(0:3) = 0b0001 4KB TSIZE(0:3) = 0b0010 16KB TSIZE(0:3) = 0b0011 64k TSIZE(0:3) = 0b0100 256k TSIZE(0:3) = 0b0101 1MB TSIZE(0:3) = 0b0110 4MB TSIZE(0:3) = 0b0111 16MB TSIZE(0:3) = 0b1000 64MB TSIZE(0:3) = 0b1001 256MB TSIZE(0:3) = 0b1010 1 GB TSIZE(0:3) = 0b1011 4 GB All other values are undefined
24:31 [56:63]	—	Reserved[1]

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

The MAS2 register is shown in Figure 6-11. Fields are defined in Table 6-11.



SPR—626; Read/ Write; Reset—Unaffected

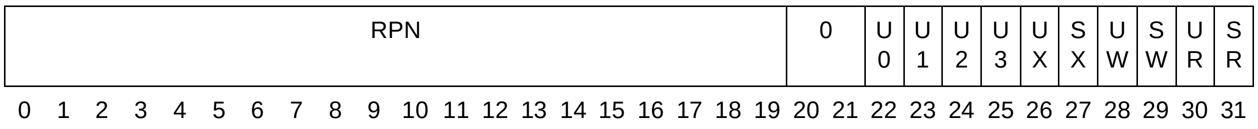
Figure 6-11. MMU Assist Register 2 (MAS2)

Table 6-11. MAS2—EPN and Page Attributes

Bit	Name	Comments, or Function when Set
0:19 [32:51]	EPN	Effective page number [0:19]
20:25 [52:57]	—	Reserved[1]
26 [58]	VLE	Power Architecture VLE 0 - This page is a standard BookE page 1 - This page is a Power Architecture 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 (such as, they are known to be required).
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 7.2.4, “Byte Lane Specification for more information

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

The MAS3 register is shown in Figure 6-12. Fields are defined in Table 6-12.



SPR—627; Read/ Write; Reset—Unaffected

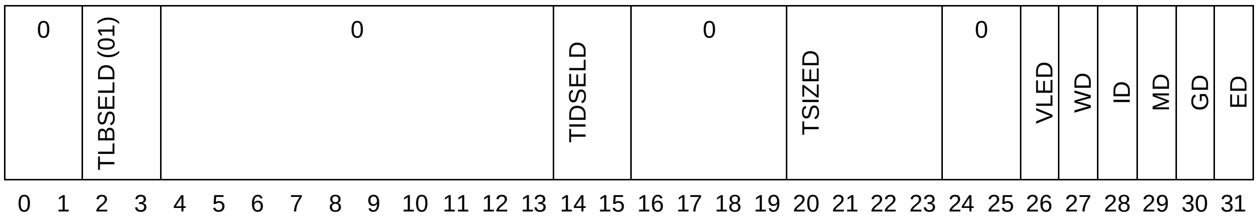
Figure 6-12. MMU Assist Register 3 (MAS3)

Table 6-12. MAS3—RPN and Access Control

Bit	Name	Comments, or Function when Set
0:19 [32:51]	RPN	Real page number [0:19] Only bits that correspond to a page number are valid. Bits that represent offsets within a page are ignored and should be zero.
20:21 [52:53]	—	Reserved[1]
22:25 [54:57]	U0-U3	User bits [0-3]
26:31 [58:63]	PERMIS	Permission bits (UX, SX, UW, SW, UR, SR)

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

The MAS4 register is shown in Figure 6-13. Fields are defined in Table 6-13.



SPR—628; Read/ Write; Reset—Unaffected

Figure 6-13. MMU Assist Register 4 (MAS4)

Table 6-13. MAS4—Hardware Replacement Assist Configuration Register

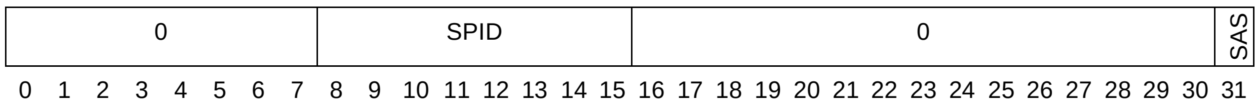
Bit	Name	Comments, or Function when Set
0:1 [32:33]	—	Reserved[1]
2:3 [34:35]	TLBSELD	Default TLB selected: 01=TLBCAM (ignored by Zen, should be written to 01 for future compatibility)
4:13 [36:45]	—	Reserved[1]

Table 6-13. MAS4—Hardware Replacement Assist Configuration Register

Bit	Name	Comments, or Function when Set
14:15 [46:47]	TIDSELD	Default PID# to load TID from 00 - PID0 01 - Reserved, do not use 10 - Reserved, do not use 11 = TIDZ (8'h00)) (Use all zeros, the globally shared value)
16:19 [48:51]	—	Reserved[1]
20:23 [52:55]	TSIZED	Default TSIZE value
24:25 [56:57]	—	Reserved[1]
26 [58]	VLED	Default VLE value This bit will always read as zero and writes will be ignored if **p_vle_present** is negated.
27:31 [59:63]	DWIMGE	Default WIMGE values

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

The MAS6 register is shown in Figure 6-14. Fields are defined in Table 6-14.



SPR—630; Read/ Write; Reset—Unaffected

Figure 6-14. MMU Assist Register 6 (MAS6)

Table 6-14. MAS6—TLB Search Context Register 0

Bit	Name	Comments, or Function when Set
0:7 [32:39]	—	Reserved[1]
8:15 [40:47]	SPID	PID value for searches
16:30 [48:62]	—	Reserved[1]
31 [63]	SAS	AS value for searches

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

6.7.4 MAS Registers Summary

The MAS registers are summarized in Figure 6-15.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
MAS0	0		TLBSEL (01)		0									ESELCAM			0													NVCAM		
MAS1	VALID	IPROT	0						TID								0			TS	TSIZ				0							
MAS2	EPN																				0						VLE	W	I	M	G	E
MAS3	RPN																				0		U0	U1	U2	U3	UX	SX	UW	SW	UR	SR
MAS4	0		TLBSELD (01)		0										TIDSELD		0				TSIZED				0		VLED	WD	ID	MD	GD	ED
MAS6	0								SPID								0															SAS
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Figure 6-15. MMU Assist Registers Summary

6.7.5 MAS Register Updates

Table 6-15 details the updates to each MAS register field for each update type.

Table 6-15. MMU Assist Register Field Updates

Bit/Field	MAS Affected	Instr/Data TLB Error	tlbsx hit	tlbsx miss	tlbre	tlbwe	ISI/DSI
TLBSEL	0	TLBSELD	‘01’	TLBSELD	NC	NC	NC
ESELCAM	0	NVCAM	matched entry	NVCAM	NC	NC	NC
NVCAM	0	NC	NC	NC	NC	NC	NC
VALID	1	1	1	0	V(array)	NC	NC
IPROT	1	0	Matched IPROT	0	IPROT(array)	NC	NC
TID[0:7]	1	TIDSELD (pid0,TIDZ)	TID(array)	SPID	TID(array)	NC	NC
TS	1	MSR(IS/DS)	SAS	SAS	TS(array)	NC	NC
TSIZE[0:3]	1	TSIZED	TSIZE(array)	TSIZED	TSIZE(array)	NC	NC
EPN[0:19]	2	I/D EPN	EPN(array)	**tlbsx** EPN	EPN(Array)	NC	NC
VWIMGE	2	Default values	VWIMGE(array)	Default values	VWIMGE(array)	NC	NC
RPN[0:19]	3	Zeroed	RPN(Array)	Zeroed	RPN(Array)	NC	NC
ACCESS (PERMISS + U0:U3)	3	Zeroed	Access(Array)	Zeroed	Access(Array)	NC	NC
TLBSELD	4	NC	NC	NC	NC	NC	NC
TIDSELD[0:1]	4	NC	NC	NC	NC	NC	NC
TSIZED[0:3]	4	NC	NC	NC	NC	NC	NC
Default 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

6.8 TLB Coherency Control

The e200 core provides the ability to invalidate a TLB entry as described in the Book E PowerPC architecture. The **tlbivax** instruction invalidates local TLB entries only. No broadcast is performed, as no hardware-based coherency support is provided.

The **tlbivax** instruction invalidates by effective address only. This means that only the TLB entry’s EPN bits are used to determine if the TLB entry should be invalidated. It is therefore possible for a single **tlbivax** instruction to invalidate multiple TLB entries, since the AS and TID fields of the entries are ignored.

6.9 Core Interface Operation for MMU Control Instructions

MMU control instructions will utilize the internal core data interface to perform MMU control instructions. The data 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 which is driven.

6.9.1 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 6-16 shows the definitions of the **p_ttype[0:3]** encodings.

Table 6-16. Transfer Type Encoding

p_ttype[0:3]	Transfer Type	Instruction
0000	Normal	normal loads/stores
0001	Atomic	**lwarx, stwcx.**
0110-0111	Reserved	—
0010-0111	Reserved	—
1000	TLB Invalidate	**tlbivax**
1001	TLB Search	**tlbsx**
1010	TLB Read entry	**tlbre**
1011	TLB Write entry	**tlbwe**
1100-1111	Reserved	—

6.10 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 9.4.4.3, "e200 OnCE Debug Output (jd_debug_b)," for more detail on controlling MMU operation during debug sessions.

Chapter 7 Core Complex Interfaces

This chapter describes the external interfaces to the e200 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/OnCE/JTAG interface signals, and a Test interface.

The memory portion of the e200 core interface is comprised of a pair of 32-bit wide system buses, one for instructions and the other for data in the e200z1. The data memory interface supports read and write transfers of 8, 16, 24, and 32 bits, supports misaligned transfers, supports true big- and little-Endian operating modes, and operates in a pipelined fashion. In the e200z1 the instruction memory interface supports read transfers of 16 and 32 bits, supports misaligned transfers, supports true big- and little-endian operating modes, and operates in a pipelined fashion.

Single-beat and misaligned transfers are supported for read and write cycles. Incrementing burst transfers are supported for instruction prefetch operations.

Misaligned accesses are supported with one or more transfers to a bus interface. If an access is misaligned, but is contained within an aligned 32-bit word, the core performs a single transfer, and the memory interface is responsible for delivering (reads) or accepting (writes) the data corresponding to the size and byte enable signals aligned according to the low order two address bits. If an access is misaligned and crosses a 32-bit boundary, the bus interface unite (BIU) performs a pair of transfers beginning at the effective address for the first transfer, along with appropriate byte enables, and for the second transfer the address is incremented to the next 32-bit boundary, and the size and byte enable signals are driven to correspond to the number of remaining bytes to be transferred.

7.1 Signal Index

This section contains an index of the e200 signals.

The following prefixes are used for e200 signal mnemonics:

- *m* denotes master clock and reset signals
- *p* denotes processor or core-related signals
- *j* denotes JTAG mode signals
- *jd* denotes JTAG and Debug mode signals
- *ipt* denotes Scan and Test Mode signals
- *nex* denotes Nexus2 signals. Nexus signals support an optional Nexus2 and Nexus3 block on the e200z1 core.

NOTE

The "_b" suffix denotes an active low signal. Signals without the active-low suffix are active high.

Figure 7-1 groups core bus and control signals by function.

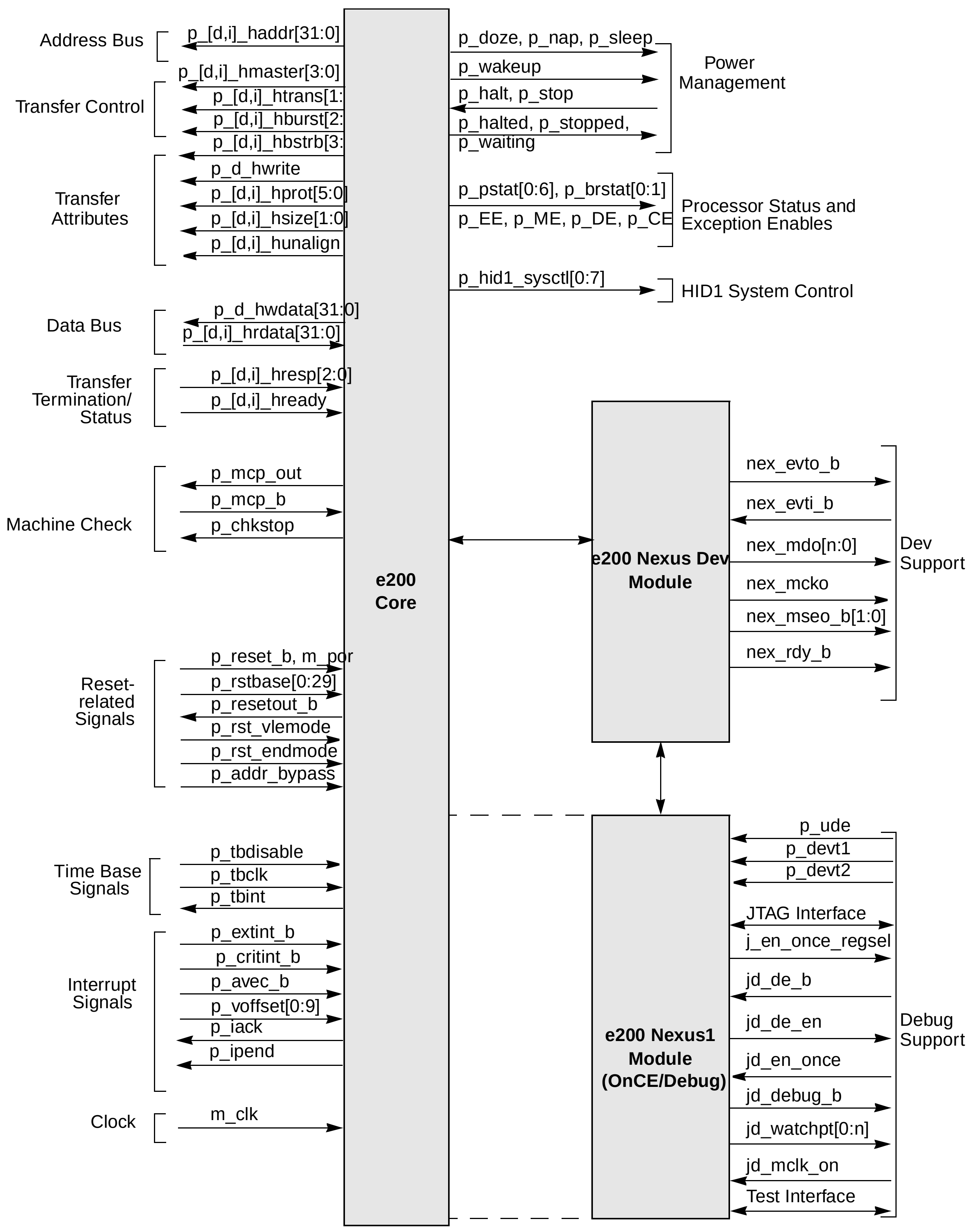


Figure 7-1. e200z1 Signal Groups

Table 7-1 shows e200 external signal function and type, signal definition, and reset value. Signals are presented in functional groups.

Table 7-1. External 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_resetout_b	O	—	Processor reset output
p_rstbase[0:29]	I	—	Reset exception handler base address, value to be loaded into TLB entry 0 on reset.
p_rst_endmode	I	—	Reset endian mode select, value to be loaded into TLB entry 0 on reset.
p_rst_pemode	I	—	Reset VLE mode select, value to be loaded into TLB entry 0 on reset.
p_addr_bypass	I	—	Address bypass mode select, value to be loaded into $\text{MMUCSR0}_{\text{BYPASS}}$ 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	O	0	Write signal
p_d_hprot[5:0], *p_i_hprot*[5:0]	O	—	Protection Codes
p_d_htrans[1:0], *p_i_htrans*[1:0]	O	—	Transfer Type
p_d_hburst[2:0], *p_i_hburst*[2:0]	O	—	Burst Type
p_d_hsize[1:0], *p_i_hsize*[1:0]	O	—	Transfer Size
p_d_hunalign, p_i_hunalign	O	—	Indicates the current access is a misaligned access.
p_d_hbstrb[3:0], *p_i_hbstrb*[3:0]	O	0	Byte strobes
p_d_hrdata[31:0], *p_i_hrdata*[31:0]	I	—	Read data buses
p_d_hwdata[31:0]	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
Interrupt Interface Signals			
p_extint_b	I	—	External Input interrupt request
p_critint_b	I	—	Critical Input interrupt request
p_avec_b	I	—	Autovector request Use internal interrupt vector offset

Table 7-1. External Interface Signal Definitions (continued)

Signal Name	Type	Reset Value	Definition
p_voffset[0:9]	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
Time Base Signals			
p_tbint	O	0	Time Base Interrupt
p_tbdisable	I	—	Time Base Disable input
p_tbclk	I	—	Time Base Clock input
Misc. CPU Signals			
p_pid0[0:7]	O	0	*PID0*[24:31] outputs
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_pstat[0:6]	O	0	Indicates processor status
p_EE, p_DE, p_CE, p_ME	O	0	Reflect the values of these MSR bits
p_brstat[0:1]	O	0	Indicates Branch prediction status
p_mcp_out	O	0	Indicates a machine check has occurred
p_chkstop	O	0	Indicates a checkstop has occurred
p_doze	O	0	Indicates low-power doze mode of operation
p_nap	O	0	Indicates low-power nap mode of operation
p_sleep	O	0	Indicates low-power sleep mode of operation
p_wakeup	O	0	Indicates to external clock control module to enable clocks and exit from low-power mode
p_halt	I	—	CPU halt request
p_halted	O	0	CPU halted
p_stop	I	—	CPU stop request
p_stopped	O	0	CPU stopped
p_waiting	O	0	CPU waiting
CPU Debug Event Signals			
p_ude	I	—	Unconditional Debug Event
p_devt1	I	—	Debug Event 1 input

Table 7-1. External Interface Signal Definitions (continued)

Signal Name	Type	Reset Value	Definition
p_devt2	I	—	Debug Event 2 input
Debug/Emulation Support Signals (Nexus 1/OnCE)			
jd_en_once	I	—	Enable full OnCE operation
jd_debug_b	O	1	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[0:7]	O	0	Indicate an address watchpoint has occurred
Development Support Signals (Nexus /3) (Optional)			
nex_mcko	O	—	Nexus 2/3 Clock Output
nex_rdy_b	O	—	Nexus 2/3 Ready Output
nex_evto_b	O	—	Nexus 2/3 Event-Out Output
nex_evti_b	I	—	Nexus 2/3 Event-In Input
nex_mdo[n:0]	O	—	Nexus 2/3 Message Data Output
nex_mseo_b[1:0]	O	—	Nexus 2/3 Message Start/End Output
JTAG-Related Signals			
j_trst_b	I	—	JTAG test reset from pad
j_tclk	I	—	JTAG test clock from pad
j_tms	I	—	JTAG test mode select from pad
j_tdi	I	—	JTAG test data input from pad
j_tdo	O	0	JTAG test data out to master controller or pad
j_tdo_en	O	0	Enables TDO output buffer
j_tst_log_rst	O	0	Indicates Test-Logic-Reset state of JTAG controller
j_capture_ir	O	0	Indicates Capture_IR state of JTAG controller
j_update_ir	O	0	Indicates Update_IR state of JTAG controller
j_shift_ir	O	0	Indicates Shift_IR state of JTAG controller
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

Table 7-1. External Interface Signal Definitions (continued)

Signal Name	Type	Reset Value	Definition
j_lsrl_regsel	O	0	External LSRL register select
j_gp_regsel[0:11]	O	0	General-purpose external JTAG register select
j_id_sequence[0:1]	I	—	JTAG ID Register (2 MSBs of sequence field)
j_id_version[0:3]	I	—	JTAG ID Register Version Field
j_serial_data	I	—	Serial data from external JTAG registers
Test Primary Input/Output Signals			
Test Control Interface[1]	—	—	Test Mode determination
Scan Test Interface[1]	—		Scan Configuration and Testing
Memory BIST Interface[1]	—	—	Memory BIST Configuration and Testing

[1] Please refer to the e200 Test Guide for information on the Test signals

7.2 Internal Interface Signals

Table 7-2 shows e200 internal signal function and type, signal definition, and reset value. Signals are presented in functional groups. Note that these signals are for reference purposes and their functionality is beyond the scope of this document.

Table 7-2. Internal Interface Signal Definitions

Signal Name	Type	Reset Value	Definition
Data Memory Interface Signals			
p_d_addr[0:31]	O	—	Address bus
p_d_rw_b	O	1	Read/$\overline{\text{write}}$
p_d_tc[0:1]	O	—	Transfer Code
p_d_ttype[0:3]	O	—	Transfer Type
p_d_tsiz[0:2]	O	—	Transfer Size
p_d_seq_b	O	1	Indicates the current access is in sequential address order from the last access.
p_d_misal_b	O	1	Indicates the current data access is the first portion of a misaligned access.
p_d_err_kill	O	1	Indicates the current access will cause an abort if terminated with error.
p_d_treq_b	O	1	Transfer Request Indicates a request for a bus cycle.
p_d_tbusy_b	O	1	Transfer Busy Indicates a bus cycle is in progress.

Table 7-2. Internal Interface Signal Definitions (continued)

Signal Name	Type	Reset Value	Definition
p_d_abort_b	O	1	Aborts a requested access.
p_d_data_in[0:31]	I	—	Input data bus
p_d_data_out[0:31]	O	—	Output data bus
p_d_ta_b	I	—	Transfer Acknowledge
p_d_tea_b	I	—	Transfer Error
p_d_bus_wrerr	I	—	Buffered Write Bus Error
p_d_tmiss_b	I	—	Translation Miss
p_d_boerr_b	I	—	Byte Ordering Error
p_d_xte_b	I	—	Precise External Termination Error
p_d_xfail_b	I	—	Store Exclusive Failure
p_d_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.
Instruction Memory Interface Signals			
p_i_addr[0:31]	O	—	Address bus
p_i_tc[0:4]	O	—	Transfer Code
p_i_tsiz[0:2]	O	—	Transfer Size
p_i_seq_b	O	1	Indicates the current access is in sequential address order from the last access. For sequential instruction fetches.
p_i_err_kill	O	1	Indicates the current access will cause an abort if terminated with error.
p_i_treq_b	O	1	Transfer Request Indicates a request for a bus cycle.
p_i_tbusy_b	O	1	Transfer Busy Indicates a bus cycle is in progress.
p_i_abort_b	O	1	Aborts a requested access.
p_i_data_in[0:31]	I	—	Input data bus
p_i_ta_b	I	—	Transfer Acknowledge
p_i_tea_b	I	—	Transfer Error
p_i_tmiss_b	I		Translation Miss
p_i_xte_b	I	—	Precise External Termination Error
p_ifsiz	I	—	Instruction Fetch Size Indicates the port size of the device being accessed.
p_i_rdbigend_b	I	—	Selects Little or Big Endian mode for read accesses.
p_rd_vle	I	—	Indicates VLE or BookE mode for inst accesses.
SPR Interface Signals			

Table 7-2. Internal Interface Signal Definitions (continued)

Signal Name	Type	Reset Value	Definition
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
p_vle_present	I	—	VLE apu status
MMU Signals			
p_d_cmbusy, p_i_cmbusy	I	—	MMU busy
p_d_dmdis, p_i_dmdis	O	—	Debug Mode MMU disable
p_d_dbg_w	O	—	Debug Mode ‘W’ attribute
p_d_dbg_i, p_i_dbg_i	O	—	Debug Mode ‘I’ attribute
p_d_dbg_m, p_i_dbg_m	O	—	Debug Mode ‘M’ attribute
p_d_dbg_g	O	—	Debug Mode ‘G’ attribute
p_d_dbg_e, p_i_dbg_e	O	—	Debug Mode ‘E’ attribute
Test Primary Input/Output Signals			
Test Control Interface	—	—	Test Mode determination
Scan Test Interface	—	—	Scan Configuration and Testing
Memory BIST Interface	—	—	Memory BIST Configuration and Testing

7.3 Signal Descriptions

The following paragraphs provide descriptions of the external signals.

7.3.1 e200 Processor Clock (*m_clk*)

The *m_clk* input is the synchronous clock source for the e200 processor core.

Because e200 is designed for static operation, *m_clk* can be gated off to lower power dissipation (for example, during low-power stopped states).

7.3.2 Reset-Related Signals

e200 supports several reset input signals for the CPU and JTAG/OnCE control logic: *m_por*, *p_reset_b*, *p_resetout_b* and *j_trst_b*. The reset domains have been partitioned such that the CPU *p_reset_b* signal does not affect JTAG/OnCE logic and *j_trst_b* does not affect processor logic. It is possible and desirable to access OnCE registers while the processor is running or in reset. Alternatively, it is also possible and desirable to assert *j_trst_b* and clear the JTAG/OnCE logic without affecting the state of the processor.

The synchronization logic between the processor and debug module requires an assertion of either *j_trst_b* or *m_por* during initial processor power-up reset in order to ensure proper operation. If the pin associated with the *j_trst_b* input is designed with a pull-up resistor and left floating, then assertion of *m_por* is required during the initial power-on processor reset. Similarly, for those systems which do not have a power-on reset circuit and choose to tie *m_por* low, it is required to assert *j_trst_b* during processor power-up reset. Once a power-up reset has been achieved, the two resets can be asserted independently.

A reset output signal *p_resetout_b* is also provided.

A set of input signals (*p_rstbase*[0:29], *p_rst_endmode, p_rst_pemode*) 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.

7.3.2.1 Power-On Reset (*m_por*)

The *m_por* signal is the power-on reset input for the e200 processor. This signal serves the following purposes:

- *m_por* is "ORed" with the *j_trst_b* function and the resulting signal clears the JTAG TAP controller and associated registers as well as the OnCE state machine. This is an asynchronous clear with a short assertion time requirement.
- *m_por* is "ORed" with the *p_reset_b* function and the resulting signal clears certain CPU registers. This is an asynchronous clear with a short assertion time requirement.

7.3.2.2 Reset (*p_reset_b*)

The *p_reset_b* input is the active-low reset input for the e200 processor. *p_reset_b* is treated as an asynchronous input and is sampled by the clock control logic in the e200 debug module.

7.3.2.3 Reset Out (*p_resetout_b*)

The *p_resetout_b* output is an active-low reset output control signal from the e200 core. *p_resetout_b* is conditionally asserted by either the Watchdog Timer (Section 2.4.8, "Timer Control Register (TCR)") or Debug control logic (Section 9.3.3.1, "Debug Control Register 0 (DBCR0)"). *p_resetout_b* is *not* asserted by *p_reset_b*.

7.3.2.4 Reset Base (*p_rstbase*[0:29])

The *p_rstbase*[0:29] inputs are provided to allow system integrators to be able to specify/relocate the base address of the reset exception handler. These inputs are used to form the upper 30 bits of the instruction access following negation of reset which is used to fetch the initial instruction of the reset exception handler. These bits should be driven to a value corresponding to the desired boot memory device in the system. These inputs must remain stable in a window beginning two clocks prior to the negation of reset and extending into the cycle in which the reset vector fetch is initiated. These inputs are also used by the MMU during reset to form a default TLB entry 0 for translation of the reset vector fetch.

The initial instruction fetch occurs to the location [*p_rstbase*[0:29]] || 2'b00.

7.3.2.5 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.

7.3.2.6 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 Book E page. A high logic level on this signal will cause the resultant entry 'VLE' bit to set to '1', indicating a VLE page.

7.3.2.7 Reset Addr Bypass (*p_addr_bypass*)

The *p_addr_bypass* input is used by the MMU during reset to update the 'BYPASS' bit of the MMUCSR0. See Section 6.7.2, "MMU Control and Status Register 0 (MMUCSR0)."

7.3.2.8 JTAG/OnCE Reset (*j_trst_b*)

The *j_trst_b* signal (referred to in the *IEEE 1149.1 JTAG Specification* as the TRST* signal) is an asynchronous reset with a short assertion time requirement. It is "ORed" with the *m_por* function and the resulting signal clears the OnCE TAP controller and associated registers as well as the OnCE state machine.

7.3.3 Address and Data Buses

Dual instruction and data interfaces are provided by the CPU. They are described together, with appropriate differences denoted.

7.3.3.1 Address Bus (*p_d_haddr*[31:0], *p_i_haddr*[31:0])

These outputs provide the address for a bus transfer. Per the AHB definition, *p_[d,i]_haddr*[31] is the MSB and *p_[d,i]_haddr*[0] is the LSB.

7.3.3.2 Read Data Bus (*p_d_hrdata*[31:0], *p_i_hrdata*[31:0])

These inputs provide data to the CPU on read transfers. The data read data bus can transfer 8, 16, 24, or 32 bits of data per bus transfer. The instruction read data bus can transfer 16 or 32 bits of data per bus transfer. Instruction transfers do not use the 8-bit and 24-bit capability. Per AHB definition, *p_[d,i]_hrdata*[31] is the MSB and *p_hrdata*[0] is the LSB. Table 7-3 shows the relationship of byte addresses to read data bus signals.

Table 7-3. *p_hrdata*[31:0] Byte Address Mappings

Memory Byte Address	Wired to *p_d_hrdata* Bits
00	7:0
01	15:8
10	23:16
11	31:24

7.3.3.3 Write Data Bus (*p_d_hwdata*[31:0])

These outputs transfer data from the CPU on write transfers. The write data bus can transfer 8, 16, 24, or 32 bits of data per bus transfer. Per AHB definition, *p_d_hwdata*[31] is the MSB and *p_d_hwdata*[0] is the LSB. Table 7-4 shows the relationship of byte addresses to write data bus signals.

Table 7-4. *p_d_hwdata*[31:0] Byte Address Mappings

Memory Byte Address	Wired to *p_d_hwdata* Bits
00	7:0
01	15:8
10	23:16
11	31:24

7.3.4 Transfer Attribute Signals

The following paragraphs describe the transfer attribute signals, which provide additional information about the bus transfer cycle. Transfer attributes are driven with address at the beginning of a bus transfer.

7.3.4.1 Transfer Type (*p_d_htrans*[1:0], *p_i_htrans*[1:0])

The processor drives these signals to indicate the current transfer type. Table 7-5 shows *p_[d,i]_htrans*[1:0] encoding.

Table 7-5. *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 e200)
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. e200 does not utilize the BUSY encoding, and does not present this type of transfer to a bus slave. Slaves must terminate IDLE transfers with a zero wait-state OKAY response and ignore the (non-existent) transfer.

7.3.4.2 Write *(p_d_hwrite*, *p_i_hwrite*)

This output signal defines the data transfer direction for the current bus cycle. A high (logic one) level indicates a write cycle, and a low (logic zero) level indicates a read cycle. For *p_i_hwrite*, the signal is internally driven low for all IAHB transfers.

7.3.4.3 Transfer Size (*p_d_hsize*[1:0], *p_i_hsize*[1:0])

The *p_[d,i]_hsize*[1:0] signals indicate the data size for a bus transfer. Table 7-6 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 7-11 and Table 7-12 for *p_[d,i]_hsize*[1:0] encodings used for aligned and misaligned transfers.

Table 7-6. *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	Reserved

7.3.4.4 Burst Type (*p_d_hburst*[2:0], *p_i_hburst*[2:0])

The *p_[d,i]_hburst*[2:0] signals indicate the burst type for a bus transfer. Table 7-7 shows the definitions of the *p_[d,i]_hburst*[2:0] encodings.

Table 7-7. *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
010	WRAP4—4-beat wrapping burst—Unused
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

e200 only utilizes SINGLE and INCR (for instruction) burst types. In addition, all INCR bursts are of word size aligned to word boundaries.

7.3.4.5 Protection Control (*p_d_hprot*[5:0], *p_i_hprot*[5:0])

e200 drives the *p_[d,i]_hprot*[5:0] signals to indicate the type of access for the current bus cycle. *p_[d,i]_hprot*[0] indicates instruction/data, *p_[d,i]_hprot*[1] indicates user/supervisor. *p_[d,i]_hprot*[5] indicates whether the access is Exclusive (such as for a **lwarx** or **stwcx.**). *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 7-8 shows the definitions of the *p_[d,i]_hprot*[5:0] signals.

Table 7-8. *p_[d,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	Instruction Access
—	—	—	—	—	1	Data Access
—	—	—	—	0	—	User mode access
—	—	—	—	1	—	Supervisor mode access
—	—	—	—	—	—	Cache-Inhibited
—	—	—	—	—	—	Guarded, not Cache-Inhibited

Table 7-8. *p_[d,i]_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
—	—	—	—	—	—	Reserved
—	—	—	—	—	—	Reserved
—	—	—	—	—	—	Reserved
—	—	—	—	—	—	Reserved
—	—	—	—	—	—	Cacheable, Writethrough
—	—	—	—	—	—	Cacheable, Writeback
0	—	—	—	—	—	Not Exclusive
1	—	—	—	—	—	Exclusive Access

Note that all signals are provided on both I and D ports, although they do not all change state (for example, *p_d_hprot*0 is always high, etc.).

e200 maps the Power Architecture Book E storage attributes to the AHB *hprot* signals in the manner described in Table 7-9.

Table 7-9. Mapping of Access Attributes to *p_hprot*[4:2] Protection Control

[I]	[G]	[W]	*p_hprot*[4]	*p_hprot*[3]	*p_hprot*[2]	Transfer Type
0	0	0	1	1	1	Cacheable, Writeback
0	0	1	1	1	0	Cacheable, Writethrough
0	1	—	0	0	1	Guarded, not Cache-Inhibited
1	—	—	0	0	0	Cache-Inhibited
—	—	—	0	0	1	Buffered Store, page marked Guarded
—	—	—	1	1	0	Buffered Store and page marked Writethrough, and non-Guarded
—	—	—	1	1	1	Buffered Store and page marked copyback, and non-Guarded

7.3.5 Byte Lane Specification

Read transactions transfer from 1 to 4 bytes of data on the *p_[d,i]_hrdata*[31:0] bus. The byte lanes involved in the transfer are determined by the starting byte number specified by the lower address bits in conjunction with the transfer size and byte strobes. Addressing of the byte lanes is shown big-endian (left to right) regardless of the Endian mode of the e200 core. The byte of memory corresponding to address 0 is connected to B0 (*p_[d,i]_h{r,w}data*[7:0]) and the byte of memory corresponding to address 3 is connected to B3 (*p_[d,i]_h{r,w}data*[31:24]). 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.

7.3.5.1 Unaligned Access (*p_d_hunalign*, *p_i_hunalign*)

The *p_[d,i]_hunalign* output signal indicates that the current access is a misaligned access. This signal is asserted for misaligned data accesses, and for misaligned instruction accesses from VLE pages. Normal Book E 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*[3:0] byte strobe signals indicate the selected bytes involved in the current portion of the misaligned access, which may not include all bytes defined by the size and low-order address signals. Aligned transfers also assert the byte strobes, but in a manner corresponding to size and low order address bits.

7.3.5.2 Byte Strobes (*p_d_hbstrb*[3:0], *p_i_hbstrb*[3:0])

The *p_[d,i]_hbstrb*[3:0] 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 correspond to the bytes defined by the size and low-order address signals. Table 7-10 shows the relationship of byte addresses to the byte strobe signals.

Table 7-10. *p_hbstrb*[3:0] to Byte Address Mappings

Memory Byte Address	Wired to p_h{r,w}data Bits	Corresponding Byte Strobe Signal
00	7:0	*p_[d,i]_hbstrb*[0]
01	15:8	*p_[d,i]_hbstrb*[1]
10	23:16	*p_[d,i]_hbstrb*[2]
11	31:24	*p_[d,i]_hbstrb*[3]

Table 7-11 lists all of the data transfer permutations. Note that misaligned data requests which cross a 32-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. e200 performs the proper byte routing internally based on endianness.

Table 7-11. Byte Strobe Assertion for Transfers

Program Size and Byte Offset	A(1:0)	HSIZE [1:0]	Data Bus Byte Strobes				HUNALIGN
			B0	B1	B2	B3	
Byte @00	0 0	0 0	X	—	—	—	0
Byte @01	0 1	0 0	—	X	—	—	0
Byte @10	1 0	0 0	—	—	X	—	0
Byte @11	1 1	0 0	—	—	—	X	0
Half @00	0 0	0 1	X	X	—	—	0
Half @01	0 1	1 0#	—	X	X	—	1
Half @10	1 0	0 1	—	—	X	X	0
Half @11 (2 bus transfers)	1 1 0 0	0 1* 0 0	— X	— —	—- —	X —	1 0
Word @00	0 0	1 0	X	X	X	X	0
Word @01 (2 bus transfers)	0 1 0 0	1 0* 0 0	— X	X —	X —	X —	1 0
Word @10 (2 bus transfers)	1 0 0 0	1 0* 0 1	— X	— X	X —	X —	1 0
Word @11 (2 bus transfers)	1 1 0 0	10* 1 0#	— X	— X	— X	X —	1 1

Note:

"X" indicates byte lanes involved in the transfer; Other lanes 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 7-12 shows the final layout in memory for data transferred from a 32-bit GPR containing the bytes ‘E F G H’ to memory. Misaligned accesses which cross a word boundary are broken into a pair of accesses by the CPU.

Table 7-12. Big- and Little-Endian Memory Storage

Program Size and Byte Offset	A(2:0)	HSIZE (1:0)	Even Word—0				0dd Word—1			
			B0	B1	B2	B3	B0	B1	B2	B3
Byte @000	0 0 0	0 0	H	—	—	—	—	—	—	—
Byte @001	0 0 1	0 0	—	H	—	—	—	—	—	—
Byte @010	0 1 0	0 0	—	—	H	—	—	—	—	—
Byte @011	0 1 1	0 0	—	—	—	H	—	—	—	—
Byte @100	1 0 0	0 0	—	—	—	—	H	—	—	—
Byte @101	1 0 1	0 0	—	—	—	—	—	H	—	—
Byte @110	1 1 0	0 0	—	—	—	—	—	—	H	—
Byte @111	1 1 1	0 0	—	—	—	—	—	—	—	H
B. E. Half @000	0 0 0	0 1	G	H	—	—	—	—	—	—
B. E. Half @001	0 0 1	1 0#	—	G	H	—	—	—	—	—
B. E. Half @010	0 1 0	0 1	—	—	G	H	—	—	—	—
B. E. Half @011	0 1 1 1 0 0	0 1 0 0	—	—	—	G	H	—	—	—
B. E. Half @100	1 0 0	0 1	—	—	—	—	G	H	—	—
B. E. Half @101	1 0 1	1 0#	—	—	—	—	—	G	H	—
B. E. Half @110	1 1 0	0 1	—	—	—	—	—	—	G	H
B.E. Half @111	1 1 1	0 1	—	—	—	—	—	—	—	G
	+ 0 0 0 (next word)	0 0	H	—	—	—	—	—	—	—
L. E. Half @000	0 0 0	0 1	H	G	—	—	—	—	—	—
L. E. Half @001	0 0 1	1 0#	—	H	G	—	—	—	—	—
L. E. Half @010	0 1 0	0 1	—	—	H	G	—	—	—	—
L. E. Half @011	0 1 1 1 0 0	0 1 0 0	—	—	—	H	—	—	—	—
L. E. Half @100	1 0 0	0 1	—	—	—	—	—	—	—	—
L. E. Half @101	1 0 1	1 0#	—	—	—	—	—	—	—	—
L. E. Half @110	1 1 0	0 1	—	—	—	—	—	—	—	—
L.E. Half @111	1 1 1	0 1	—	—	—	—	—	—	—	—
	+ 0 0 0 (next word)	0 0	G	—	—	—	—	—	—	—
B. E. Word @000	0 0 0	1 0	E	F	G	H	—	—	—	—

Table 7-12. Big- and Little-Endian Memory Storage (continued)

Program Size and Byte Offset	A(2:0)	HSIZE (1:0)	Even Word—0				0dd Word—1			
			B0	B1	B2	B3	B0	B1	B2	B3
B. E. Word @001	0 0 1 1 0 0	1 0* 0 0	—	E	F	G	H	—	—	—
B. E. Word @010	0 1 0 1 0 0	1 0* 0 1	—	—	E	F	G	H	—	—
B. E. Word @011	0 1 1 1 0 0	1 0* 1 0#	—	—	—	E	F	G	H	—
B. E. Word @100	1 0 0	1 0	—	—	—	—	E	F	G	H
B. E. Word @101	1 0 1	1 0*	—	—	—	—	—	E	F	G
	+ 0 0 0 (next word)	0 0	H	—	—	—	—	—	—	—
B. E. Word @110	1 1 0	1 0*	—	—	—	—	—	—	E	F
	+ 0 0 0 (next word)	0 1	G	H	—	—	—	—	—	—
B. E. Word @111	1 1 1	1 0*	—	—	—	—	—	—	—	E
	+ 0 0 0 (next word)	1 0#	F	G	H	—	—	—	—	—
L. E. Word @000	0 0 0	1 0	H	G	F	E	—	—	—	—
L. E. Word @001	0 0 1 1 0 0	1 0# 0 0	—	H	G	F	E	—	—	—
L. E. Word @010	0 1 0 1 0 0	1 0# 0 1	—	—	H	G	F	E	—	—
L. E. Word @011	0 1 1 1 0 0	1 0* 1 0#	—	—	—	H	G	F	E	—
L. E. Word @100	1 0 0	1 0	—	—	—	—	H	G	F	E
L. E. Word @101	1 0 1	1 0*	—	—	—	—	—	H	G	F
	+ 0 0 0 (next word)	0 0	E	—	—	—	—	—	—	—
L. E. Word @110	1 1 0	1 0*	—	—	—	—	—	—	H	G
	+ 0 0 0 (next word)	0 1	F	E	—	—	—	—	—	—
L. E. Word @111	1 1 1	1 0*	—	—	—	—	—	—	—	H
	+ 0 0 0 (next word)	1 0#	G	F	E	—	—	—	—	—

Notes:

Assumes a 32-bit GPR contains ‘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.

***** These misaligned cases drive request size according to the size specified by the load or store instruction.

7.3.6 Transfer Control Signals

The following paragraphs describe the transfer control signals.

7.3.6.1 Transfer Ready (*p_d_hready, p_i_hready*)

The *p_[d,i]_hready* input signal indicates completion of a requested transfer operation. An external device asserts *p_[d,i]_hready* to terminate the transfer. The *p_[d,i]_hresp*[2:0] signals indicate status of the transfer.

7.3.6.2 Transfer Response (*p_d_hresp*[2:0], *p_i_hresp*[1:0])

The *p_d_hresp*[2:0] and *p_i_hresp*[1:0] signals indicate status of a terminating transfer on the respective interfaces. Table 7-13 shows the definitions of the *p_d_hresp*[2:0] and *p_i_hresp*[1:0] encodings.

Table 7-13. *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 7-14. *p_i_hresp*[1:0] Transfer Response Encoding

***p_i_hresp*[1:0]**	**Response Type**
00	OKAY—transfer terminated normally
01	ERROR—transfer terminated abnormally
10	Reserved (RETRY not supported in AHB-Lite protocol)
11	Reserved (SPLIT not supported in AHB-Lite protocol)

The ERROR and XFAIL responses are required to be two cycle responses. In this case, the ERROR or XFAIL responses must be signaled one cycle prior to assertion of *p_[d,i]_hready*, and must remain unchanged during the cycle *p_[d,i]_hready* is asserted.

The XFAIL response is signaled to the CPU.

7.3.7 Interrupt Signals

The following paragraphs describe the signals which 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.

On the other hand, 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 input *p_mcp_b* is transition-sensitive and must be held asserted until acknowledged in order to be guaranteed to be recognized, although there is no guarantee the CPU will not recognize the interrupt request even if the request is later removed.

7.3.7.1 External Input Interrupt Request (*p_extint_b*)

This active-low signal provides the External Input interrupt request to the e200 core. *p_extint_b* is masked by the MSR[EE] bit. This signal is not internally synchronized by the e200 core, thus it must meet setup and hold time constraints relative to *m_clk* when the e200 core clock is running. This signal is level sensitive and must remain asserted to be guaranteed to be recognized.

7.3.7.2 Critical Input Interrupt Request (*p_critint_b*)

This active-low signal provides the Critical Input interrupt request to the e200 core. *p_critint_b* is masked by the MSR[CE] bit. This signal is not internally synchronized by the e200 core, thus it must meet setup and hold time constraints relative to *m_clk* when the e200 core clock is running. This signal is level sensitive and must remain asserted to be guaranteed to be recognized.

7.3.7.3 Interrupt Pending (*p_ipend*)

This active-high signal indicates that an asserted *p_extint_b* or *p_critint_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. 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.

7.3.7.4 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 values 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*[0:9] 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.

7.3.7.5 Interrupt Vector Offset (*p_voffset*[0:9])

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*[0:9] signals correspond to bits 20:29 of the exception handler address (the low order two bits 30:31 are forced to 00). The *p_voffset*[0:9] 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.

7.3.7.6 Interrupt Vector Acknowledge (*p_iack*)

The *p_iack* output signal provide an interrupt vector acknowledge indicator to allow external interrupt controllers to be informed when a critical input or external input interrupt is being processed. The *p_iack* signal is asserted after the cycle in which the *p_avec_b* and *p_voffset*[0:9] signals are sampled in preparation for exception processing. See Table 7-14 and Figure 7-30 for timing diagrams of operation.

7.3.7.7 Machine Check (*p_mcp_b*)

This active-low signal provides the Machine Check interrupt request to the e200 core. *p_mcp_b* is masked by the HID0[EMCP] bit. This signal is not internally synchronized by the e200 core, thus it must meet setup and hold time constraints to *m_clk* when the e200 core clock is running. The *p_mcp_b* input is sampled on two consecutive *m_clk periods* to detect a transition from the negated to the asserted state. It is internally qualified with this transition, but must remain asserted to be guaranteed to be recognized.

The *p_mcp_b* signal is not sampled while the e200 core is in the halted or stopped power management states, but is sampled while the CPU is in the waiting state. See Section 7.3.12.3, "Processor Halted (p_halted)," and Section 7.3.12.5, "Processor Stopped (p_stopped)." Also, when the core is in the debug state (as reflected on the *cpu_dbgack* internal state signal), the internal *m_clk* is not running, so the *p_mcp_b* input is not recognized until the core is released with a go+noexit or a go+exit OnCE command.

7.3.8 Timer Facility Signals

The following sub-sections describe the processor signals associated with the Timer Facilities (Time Base, Watchdog, Fixed-interval and Decrementer).

7.3.8.1 Timer Disable (*p_tbdisable*)

The active-high *p_tbdisable* input signal is used to disable the internal Time Base and Decrementer counters. When this signal is asserted, Time Base and Decrementer updates are frozen. When this signal is negated, Time Base and Decrementer updates are unaffected. This signal may be used to freeze the state of the Time Base and Decrementer during low power or debug operation. This signal is not internally synchronized by the e200 core, thus it must meet setup and hold time constraints relative to *m_clk* when the e200 core clock is running, as well as to *p_tbclk* when selected as an alternate clock source for the Time Base.

7.3.8.2 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*.

7.3.8.3 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.

7.3.9 Processor Reservation Signals

The following sub-sections describe processor reservation signals associated with the **lwarx** and **stwcx.**. instructions.

7.3.9.1 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 **lwarx** instruction. This signal is set following the successful completion of a **lwarx**. This signal remains set until the reservation has been cleared. (Refer to Section 3.6, "Memory Synchronization and Reservation Instructions"). This signal is provided as a status indicator for specialized system applications only.

7.3.9.2 CPU Reservation Clear (*p_rsrv_clr*)

The active-high *p_rsrv_clr* input signal is used to clear a reservation that has been previously established. External reservation management logic may use this signal to implement reservation management policies which are outside of the scope of the CPU. (Refer to Section 3.6, "Memory Synchronization and Reservation Instructions"). This signal may be asserted independently of any bus transfer.

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

The *p_d_xfail_b* signal is provided to indicate success/failure of a **stwcx.** instruction as part of bus transfer termination using the XFAIL *p_d_hresp*[2:0] encoding. See Section 7.2.3.7, "Store Exclusive Failure (p_d_xfail_b)," for more detail on *p_d_xfail_b*.

7.3.10 Miscellaneous Processor Signals

The following paragraph describes several miscellaneous processor signals.

7.3.10.1 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.

7.3.10.2 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 asserts during the clock cycle the *p_pid0*[0:7] outputs are changing.

7.3.10.3 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)."

7.3.11 Processor State Signals

The following sub-sections describe processor internal state signals.

7.3.11.1 Processor Status (*p_pstat*[0:6])

These signals indicate the internal execution unit status. The timing is synchronous with the *m_clk*, so the indicated status may not apply to a current bus transfer. Table 7-15 shows *p_pstat*[0:6] encoding.

Table 7-15. Processor Status Encoding[1]

p_pstat[0:6]							Internal Processor Status
0	0	0	0	0	x	x	Execution Stalled
0	0	0	0	1	x	x	Execute Exception
0	0	0	1	0	x	x	Instruction Squashed
0	0	0	1	1	x	x	Reserved
0	0	1	0	0	x	x	Reserved
0	0	1	0	1	x	x	Reserved
0	0	1	1	0	x	x	Reserved
0	0	1	1	1	x	x	Processor in Waiting State
0	1	0	0	0	x	x	Processor in Halted state
0	1	0	0	1	x	x	Processor in Stopped state
0	1	0	1	0	x	x	Processor in Debug mode[2]
0	1	0	1	1	x	x	Processor in Checkstop state
0	1	1	0	0	x	x	Reserved

Table 7-15. Processor Status Encoding[1] (continued)

p_pstat[0:6]							Internal Processor Status
0	1	1	0	1	x	x	Reserved
0	1	1	1	0	x	x	Reserved
0	1	1	1	1	x	x	Reserved
1	0	0	0	0	s	m	Complete Instruction[3,4]
1	0	0	0	1	0	0	Complete **lmw**, or **stmw**
1	0	0	0	1	0	1	Complete **e_lmw**, or **e_stmw**
1	0	0	1	0	0	0	Complete **isync**
1	0	0	1	0	1	1	Complete **se_isync**
1	0	0	1	1	0	m	Complete **lwarx** or **stwcx.**[5]
1	0	1	0	0	x	x	Reserved
1	0	1	0	1	x	x	Reserved
1	0	1	1	0	x	x	Reserved
1	0	1	1	1	x	x	Reserved
1	1	0	0	0	0	0	Complete Branch Instruction **bc**, **bcl**, **bca**, **bcla, b, bl, ba, bla** resolved as not taken
1	1	0	0	0	0	1	Complete Branch Instruction **e_bc**, **e_bcl**, **e_b, e_bl** resolved as not taken
1	1	0	0	0	1	1	Complete Branch Instruction **se_bc**, **se_bcl**, **se_b, se_bl** resolved as not taken
1	1	0	0	1	0	0	Complete Branch Instruction **bc**, **bcl**, **bca**, **bcla, b, bl, ba, bla** resolved as taken
1	1	0	0	1	0	1	Complete Branch Instruction **e_bc**, **e_bcl**, **e_b, e_bl** resolved as taken
1	1	0	0	1	1	1	Complete Branch Instruction **se_bc**, **se_bcl**, **se_b, se_bl** resolved as taken
1	1	0	1	0	0	0	Complete **bclr**, **bclrl, bcctr**, **bcctrl** resolved as not taken
1	1	0	1	1	0	0	Complete **bclr**, **bclrl, bcctr**, **bcctrl** resolved as taken
1	1	0	1	1	1	1	Complete **se_blr**, **se_blrl, se_bctr**, **se_bctrl** (always taken)
1	1	1	0	0	0	m	Complete **isel** with condition false
1	1	1	0	1	0	m	Complete **isel** with condition true
1	1	1	1	0	x	x	Reserved
1	1	1	1	1	0	0	Complete **rfi**, **rfci**, or **rfdi**
1	1	1	1	1	1	1	Complete **se_rfi**, **se_rfci**, or **se_rfdi**

1 All encodings which do not appear in the table are reserved
2 As reflected on the **cpu_dbgack** internal state signal
3 Except **rfi, rfci, rfdi, lmw, stmw, lwarx, stwcx., isync, isel, se_rfi, se_rfci, se_rfdi, e_lmw, e_stmw, se_isel,** and Change of Flow Instructions
4 s—instruction size, 0=32-bit, 1=16-bit
m—0 for BookE page, 1 for VLE page
5 m—0 for BookE page, 1 for VLE page

7.3.11.2 Processor Exception Enable MSR Values (*p_EE, p_CE, p_DE, p_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**, **se_rfi**, **se_rfci**, **se_rfdi**, **wrtee**, or **wrteei** instruction, or during exception processing where one or more bits may be cleared during the exception processing sequence.

7.3.11.3 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 7-16 shows **p_brstat[0:1]** encoding.

Table 7-16. Branch Prediction Status Encoding

***p_brstat*[0:1]**		**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

7.3.11.4 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 a syndrome bit to be set in the Machine Check Syndrome register. Refer to Section 2.4.7, "Machine Check Syndrome Register (MCSR)."

7.3.11.5 Processor Checkstop (*p_chkstop*)

The active-high p_chkstop output signal is asserted by the processor when a checkstop condition has occurred and the CPU has entered the checkstop state.

7.3.12 Power Management Control Signals

The following signals are provided for power management or other control functions by external control logic.

7.3.12.1 Processor Waiting (*p_waiting*)

The active-high p_waiting output signal is used to indicate that the processor has entered the Waiting state (Section 8.1.2, "Waiting State").

7.3.12.2 Processor Halt Request (*p_halt)*

The active-high p_halt input signal is used to request the processor to enter the Halted state (Section 8.1.3, “Halted State”).

7.3.12.3 Processor Halted (*p_halted*)

The active-high p_halted output signal is used to indicate that the processor has entered the Halted state (Section 8.1.3, “Halted State”).

7.3.12.4 Processor Stop Request (*p_stop*)

The active-high *p_stop* input signal is used to request the processor to enter the Stopped state (Section 8.1.4, “Stopped State”).

7.3.12.5 Processor Stopped (*p_stopped*)

The active-high *p_stopped* output signal is used to indicate that the processor has entered the Stopped state (Section 8.1.4, “Stopped State”).

7.3.12.6 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 e200 core and peripherals in a low-power consumption state. The *p_wakeup* signal can be monitored to determine when to end the low-power condition.

The e200 core can be placed in a low-power state by forcing the *m_clk* input to a quiescent state, and brought out of low-power state by re-enabling *m_clk*. The Time Base facilities may be separately enabled or disabled using combinations of the Timer Facility control signals described in Section 7.3.8, “Timer Facility Signals.”

7.3.12.7 Wakeup (*p_wakeup*)

The active-high *p_wakeup* output signal should be used by external logic to remove the e200 core and system logic from a low-power state. It also is used to indicate to the system clock controller that the *m_clk* input should be re-enabled for debug purposes. This signal is asynchronous to the system clock and should be synchronized to the system clock domain to avoid hazards.

p_wakeup asserts whenever the following occurs:

- A valid pending interrupt is detected by the core
- A request to enter debug mode is made by setting the DR bit in the OnCE control register (OCR) or via the assertion of the *jd_de_b* or *p_ude* input signals.
- The processor is in a debug session and the *jd_debug_b* output is asserted

- A request to enable the *m_clk* input has been made by setting the WKUP bit in the OnCE control register

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.

7.3.13 Debug Event Signals

The following interface signals are provided to signal debug events to the e200 core.

7.3.13.1 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 9.2.13, "Unconditional Debug Event." This signal is not internally synchronized by the e200 core, thus it must meet setup and hold time constraints relative to *m_clk* when the e200 core clock is running. This signal is level sensitive and must be held asserted until acknowledged by software, or, when external debug mode is enabled, by assertion of the *jd_debug_b* output to be guaranteed to be recognized. In addition, only a transition from the negated state to the asserted state of the *p_ude* signal causes an event to occur. The level on this signal is used however to cause assertion of the *p_wakeup* output.

7.3.13.2 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 9.2.12, "External Debug Event." This signal is not internally synchronized by the e200 core, thus it must meet setup and hold time constraints relative to *m_clk* when the e200 core clock is running. If the e200 core clock is disabled, this signal is not recognized. In addition, only a transition from the negated state to the asserted state of the *p_devt1* signal causes an event to occur. It is intended to signal e200 related events that are generated while the CPU is active.

7.3.13.3 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 9.2.12, "External Debug Event." This signal is not internally synchronized by the e200 core, thus it must meet setup and hold time constraints relative to *m_clk* when the e200 core clock is running. If the e200 core clock is disabled, this signal is not recognized. In addition, only a transition from the negated state to the asserted state of the *p_devt2* signal causes an event to occur. It is intended to signal e200 related events that are generated while the CPU is active.

7.3.14 Debug/Emulation (Nexus 1/OnCE) Support Signals

The following interface signals are provided to assist in implementing an On-Chip Emulation capability with a controller external to the e200 core.

Table 7-17. e200 Debug/Emulation Support Signals

Signal	Type	Description
jd_en_once	I	Enable full OnCE operation
jd_debug_b	O	Debug Session indicator
jd_de_b	I	Debug request
jd_de_en	O	DE_b active high output enable
jd_mclk_on	I	CPU clock is active indicator

7.3.14.1 OnCE Enable (*jd_en_once*)

The OnCE enable signal *jd_en_once* is used to enable the OnCE controller to allow certain instructions and operations to be executed. Assertion of this signal enables the full OnCE command set, as well as operation of control signals and OnCE Control register functions. When this signal is disabled, only the Bypass, ID and Enable_OnCE commands are executed by the Z5 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 7.3.15.15, “Enable Once Register Select (j_en_once_regsel),” for a description of the *j_en_once_regsel* output signal, and to Section 7.3.15.19, “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 takes effect after one additional *j_tclk* cycle of synchronization.

7.3.14.2 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 9.4.4.3, “e200 OnCE Debug Output (jd_debug_b),” for additional information on this signal.

7.3.14.3 Debug Request (*jd_de_b*)

This signal is the debug mode request input. This signal is not internally synchronized by the e200 core, thus it must meet setup and hold time constraints relative to *j_tclk*. To be recognized, it must be held asserted for a minimum of two *j_tclk* periods, and the *jd_en_once* input must be in the asserted state. *jd_de_b* is synchronized to *m_clk* in the debug module before being sent to the processor (two clocks).

This signal is normally the input from the top-level *DE_b* open-drain bidirectional I/O cell. Refer to Section 9.4.4.2, "OnCE Debug Request/Event (jd_de_b, jd_de_en)," for additional information on this signal.

7.3.14.4 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 9.4.4.2, "OnCE Debug Request/Event (jd_de_b, jd_de_en)," for additional information on this signal.

7.3.14.5 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.

7.3.14.6 Watchpoint Events (*jd_watchpoint*[0:7])

The *jd_watchpoint*[0:7] active-high output signals are used to indicate that a watchpoint has occurred. Each debug address compare function (IAC1-4, DAC1-2), and Debug Counter event (DCNT1-2) is capable of triggering a watchpoint output. Refer to Section 9.5, "Watchpoint Support," for the signal assignments of each watchpoint source.

7.3.15 JTAG Support Signals

Table 7-18 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 7-18. 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.*

7.3.15.1 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.

7.3.15.2 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.

7.3.15.3 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 IEEE Std 1149™ compatibility, the system level pin should change state on the falling edge of TCLK.

7.3.15.4 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.

7.3.15.5 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 7-19 details additional signals which may be used to support external JTAG data registers using the e200 TAP controller.

Table 7-19. 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

Table 7-19. JTAG Signals Used to Support External Registers (continued)

Signal Name	Type	Description
j_capture_dr	O	Indicates the TAP controller is in the capture DR state
j_shift_dr	O	Indicates the TAP controller is in shift DR state
j_update_gp_reg	O	Updates JTAG controller general-purpose data register
j_gp_regsel[0:11]	O	General-purpose external JTAG register select
j_en_once_regsel	O	External Enable OnCE register select
j_key_in	I	Serial data from external key logic
j_lsrl_regsel	O	External LSRL register select
j_serial_data	I	Serial data from external JTAG register(s)

7.3.15.6 Test-Logic-Reset (*j_tst_log_rst*)

This signal indicates the TAP controller is in the Test-Logic-Reset state.

7.3.15.7 Run-Test/Idle (*j_rti*)

This signal indicates the TAP controller is in the Run-Test/Idle state.

7.3.15.8 Capture IR (*j_capture_ir*)

This signal indicates the TAP controller is in the Capture_IR state.

7.3.15.9 Shift IR (*j_shift_ir*)

This signal indicates the TAP controller is in the Shift_IR state.

7.3.15.10 Update IR (*j_update_ir*)

This signal indicates the TAP controller is in the Update_IR state.

7.3.15.11 Capture DR (*j_capture_dr*)

This signal indicates the TAP controller is in the Capture_DR state.

7.3.15.12 Shift DR (*j_shift_dr*)

This signal indicates the TAP controller is in the Shift_DR state.

7.3.15.13 Update DR (*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[0:11]* signals should be monitored to see which register, if any, needs to be updated.

7.3.15.14 Register Select (*j_gp_regsel*)

The outputs shown in Table 7-20 are a decode of the REGSEL[0:6] field in the OnCE Command Register (OCMD). They are used to specify which external general purpose JTAG register to access via the e200 TAP controller.

Table 7-20. 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
j_gp_regsel[10]	O	REGSEL[0:6]=7’h7A
j_gp_regsel[11]	O	REGSEL[0:6]=7’h7B

7.3.15.15 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 e200 TAP controller. This control signal may be used by external security logic to assist in controlling the *jd_enable_once* input signal. The external Enable_OnCE register should be muxed onto the *j_serial_data* input (Refer to Section 7.3.15.18, “Serial Data (j_serial_data)”). During the Shift_DR state, *j_serial_data* is supplied to the *j_tdo* output.

7.3.15.16 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 e200 TAP controller.

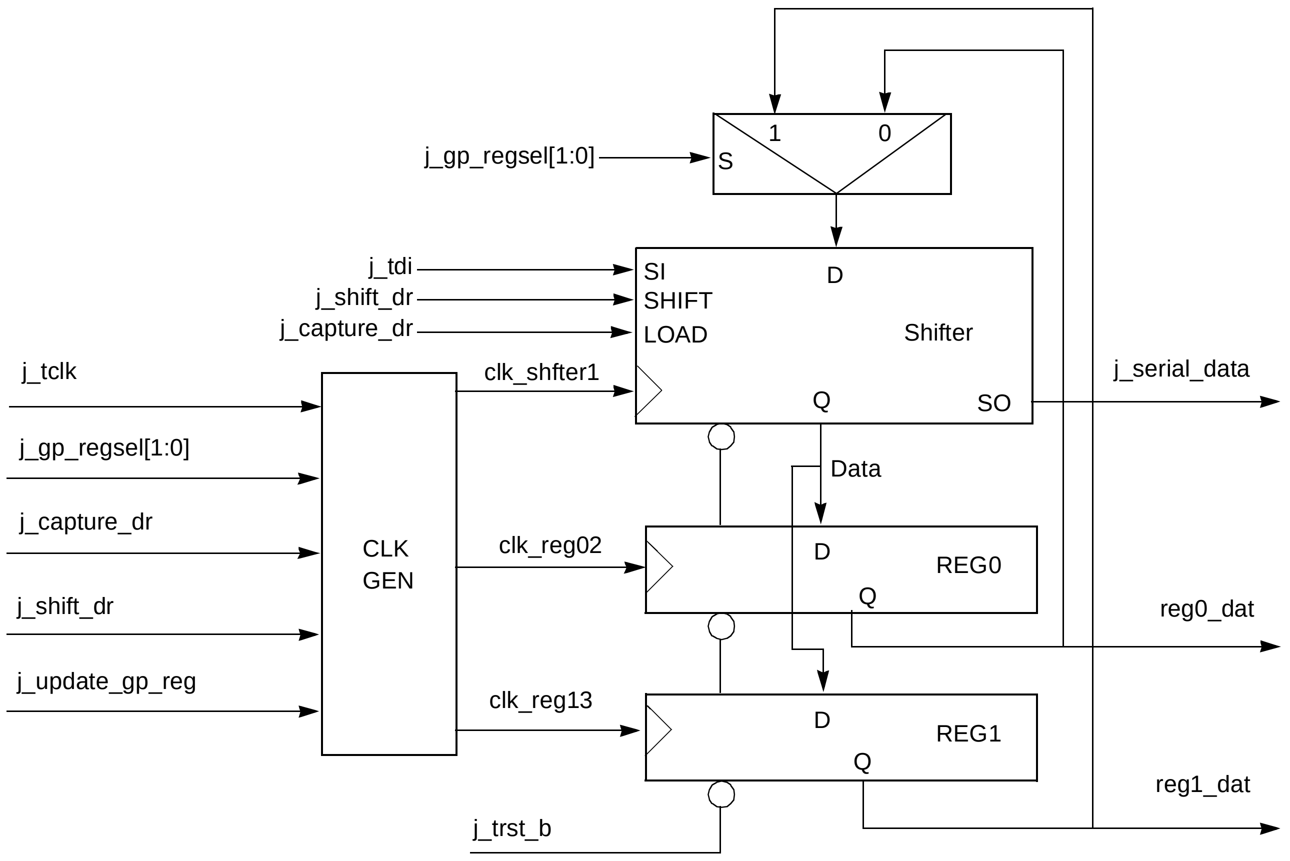
7.3.15.17 External LSRL Register Select (*j_lsrl_regsel*)

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 e200 TAP controller.

7.3.15.18 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*[0:11], *j_lsrl_regsel*, and *j_en_once_regsel* signals. The data is internally routed to *j_tdo*.

Figure 7-2 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 e200 core.The shifter and the registers may be as wide as the application warrants [0:x]. The length determines the number of states the TAP controller is held in Shift_DR (x+1).



NOTES:
1. clk_shfter = j_tclk and (j_shift_dr | j_capture_dr)
2. clk_reg0 = j_tclk and j_update_gp_reg and j_gp_regsel[0]
3. clk_reg1 = j_tclk and j_update_gp_reg and j_gp_regsel[1]

Figure 7-2. Example External JTAG Register Design

7.3.15.19 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 processor which conditionally asserts *jd_en_once*. During the Shift_IR state, when *jd_en_once* is negated, this input is sampled on the rising edge of *j_tclk*, and after a two clock delay the data is internally routed to *j_tdo*. This allows provision of a key value via the *j_tdo* output following a transition from Capture_IR to Shift_IR. The key value is provided via the *j_key_in* input.

7.3.16 JTAG ID Signals

Table 7-21 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 7-21. JTAG Register ID Fields

Bit Field	Type	Description	Value
[31:28]	Variable	Version Number	Variable
[27:22]	Fixed	Design Center Number	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 e200 core shifts out a “1” as the first bit on *j_tdo* if the Shift_DR state is entered directly from the test-logic-reset state. This is per the JTAG specification and informs any JTAG controller that an ID register exists on the part. The e200 JTAG ID register is accessed by writing the OCMR (OnCE Command Register) with the value 7’h02 in the REGSEL[0:6] field.

The JTAG ID bit, manufacturer ID field, and design center number are fixed by the JTAG Consortium and/or Freescale. The version numbers and the two most significant bits (MSBs) of the sequence number are variable and brought out to external ports. The lower eight bits of the sequence number are variable and strapped internally to track variations in processor deliverables.

Table 7-22 shows the inputs to the JTAG ID register that are input ports on the e200 core. These bits are provided for a customer to track revisions of a device using the e200 core.

Table 7-22. 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

7.3.16.1 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.

7.3.16.2 JTAG ID Sequence (*j_id_sequence*[2:9])

The *j_id_sequence*[2:9] field is internally strapped to track variations in processor and module deliverables. Each e200 deliverable has a unique sequence number. Additionally, each revision of these modules can be identified by unique sequence numbers.

7.3.16.3 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.

7.4 Timing Diagrams

7.4.1 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, and word transfers. All bus input and output signals are sampled and driven with respect to the rising edge of the *m_clk* signal. The core moves data on the bus by issuing control signals and using a handshake protocol to ensure correct data movement.

The memory interface operates in a pipelined fashion to allow additional access time for memory and peripherals. AHB transfers consist of an address phase which lasts only a single cycle, followed by the data phase which may last for one or more cycles depending on the state of the *p_hready* signal.

Read transfers consist of a request cycle, where address and attributes are driven along with a transfer request, and one or more memory access cycles to perform accesses and return data to the CPU for alignment, sign or zero extension, and forwarding.

Write transfers consist of a request cycle, where address and attributes are driven along with a transfer request, and one or more data drive cycles where write data is driven and external devices accept write data for the access.

Access requests are generated in an overlapped fashion in order to support sustained single cycle transfers. Up to two access requests may be in progress at any one cycle, one access outstanding and a second in the pending request phase.

Access requests are assumed to be accepted as long as there are no accesses in progress, or if an access in progress is terminated during the same cycle a new request is active (*p_hready* asserted). Once an access has been accepted, the BIU is free to change the current request at any time, even if part of a burst transfer.

The local memory control logic is responsible for proper pipelining and latching of all interface signals to initiate memory accesses.

The system hardware can use the *p_hresp*[2:0] signals to signal that the current bus cycle has an error when a fault is detected, using the ERROR response encoding. ERROR assertion requires a two cycle response. In the first cycle of the response, the *p_hresp*[2:0] signals are driven to indicate ERROR and *p_hready* must be negated. During the following cycle, the ERROR response must continue to be driven, and *p_hready* must be asserted. When the core recognizes a bus error condition for an access at the end of the first cycle of the two cycle error response, a subsequent pending access request may be removed by the BIU driving the *p_htrans*[2:0] signals to the IDLE state in the second cycle of the two cycle error response. Not all pending requests are removed, however.

When a bus cycle is terminated with a bus error, the core can enter storage error exception processing immediately following the bus cycle, or it can defer processing the exception.

The instruction prefetch mechanism requests instruction words from the instruction memory unit before it is ready to execute them. If a bus error occurs on an instruction fetch, the core does not take the exception until it attempts to use the instruction. Should an intervening instruction cause a branch, or should a task switch occur, the storage error exception for the unused access does not occur. A bus error termination for any write access or read access that reference data specifically requested by the execution unit causes the core to begin exception processing.

7.4.1.1 Basic Read Transfer Cycles

During a read transfer, the core receives data from a memory or peripheral device. Figure 7-3 illustrates functional timing for basic read transfers. Clock-by-clock descriptions of activity in Figure 7-3 follows:

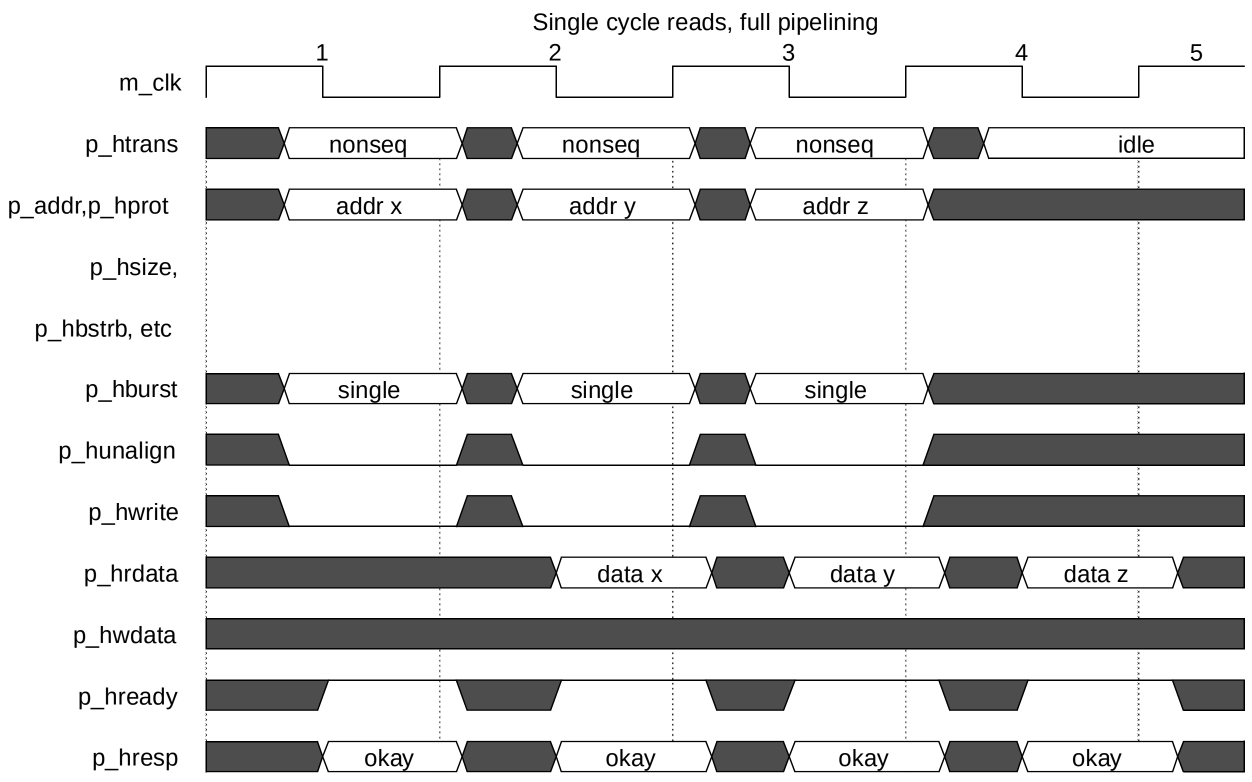


Figure 7-3. Basic Read Transfers

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*[3:0]) 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. Because the bus is currently idle, (0 transfers outstanding), the first read request to addr$_x$ is considered *taken* at the end of C1. The default slave drives an ready/OKAY response for the current idle cycle.

Clock 2 (C2):

During C2, the addr$_x$ memory access takes place using the address and attribute values which 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 because the access to addr$_x$ is completing, it is considered *taken* at the end of C2.

Clock 3 (C3):

During C3, the addr$_y$ memory access takes place using the address and attribute values which were driven during C2 to enable reading of one or more bytes of memory. Read data from the slave device for addr$_y$ is provided on the *p_hrdata* inputs. The slave device responds by asserting *p_hready* to indicate the cycle is completing and drives an OKAY response.

Another read transfer request is made during C3 to addr$_z$ (*p_htrans* = NONSEQ), and because the access to addr$_y$ is completing, it is considered *taken* at the end of C3.

Clock 4 (C4):

During C4, the addr$_z$ memory access takes place using the address and attribute values which were driven during C3 to enable reading of one or more bytes of memory. Read data from the slave device for addr$_z$ is provided on the *p_hrdata* inputs. The slave device responds by asserting *p_hready* to indicate the cycle is completing and drives an OKAY response.

The CPU has no more outstanding requests, so *p_htrans* indicates IDLE. The address and attribute signals are thus undefined.

7.4.1.2 Read Transfer with Wait State

Figure 7-4 shows an example of wait state operation. Signal *p_hready* for the first request (addr$_x$) is not asserted during C2, so a wait state is inserted until *p_hready* is recognized (during C3).

Meanwhile, a subsequent request has been generated by the CPU for addr$_y$ which is not *taken* in C2, because the previous transaction is still outstanding. The address and transfer attributes remain driven in cycle C3 and are taken at the end of C3 because the previous access is completing. Data for addr$_x$ and a ready/OKAY response is driven back by the slave device. In cycle C4, a request for addr$_z$ is made. The request for access to addr$_z$ is taken at the end of C4, and during C5, the data and a ready/OKAY response is provided by the slave device. In cycle C5, no further accesses are requested.

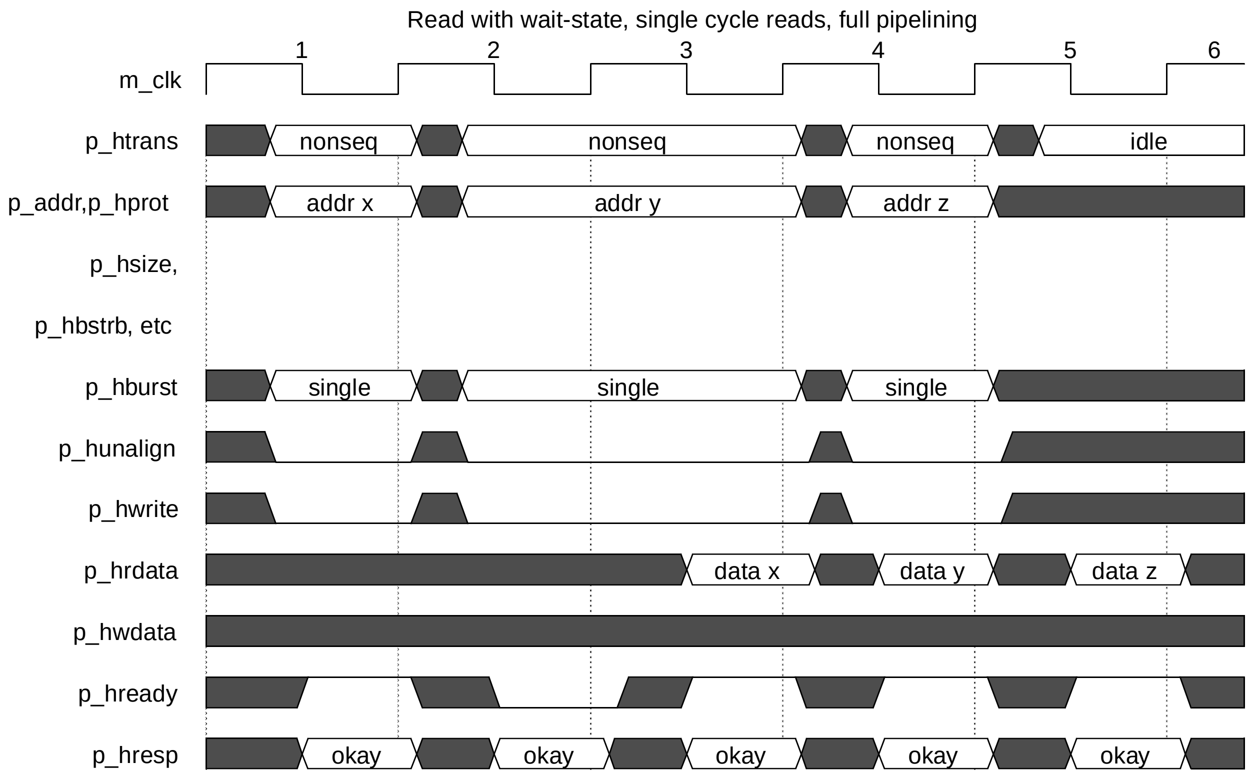


Figure 7-4. Read Transfer with Wait-State

7.4.1.3 Basic Write Transfer Cycles

During a write transfer, the core provides write data to a memory or peripheral device. Figure 7-5 illustrates functional timing for basic write transfers. Clock-by-clock descriptions of activity in Figure 7-5 follows:

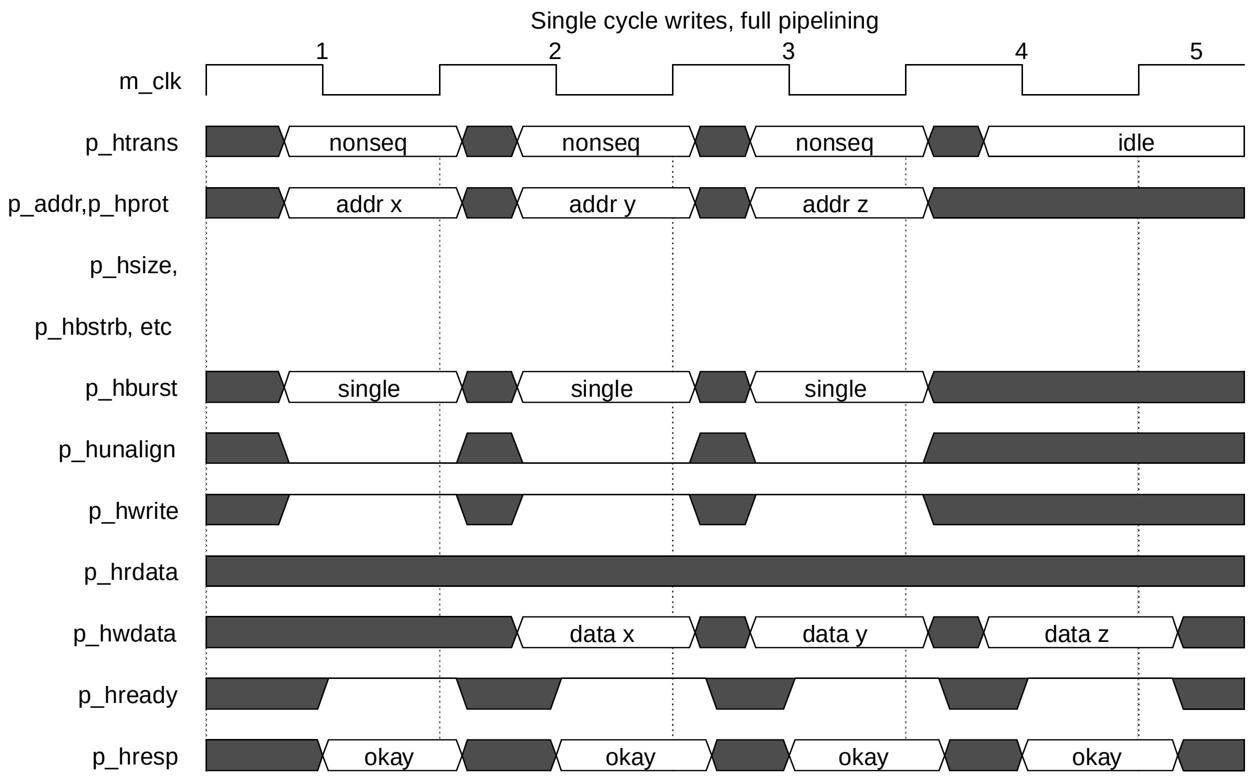


Figure 7-5. Basic Write Transfers

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 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*[3:0]) are driven to indicate active byte lanes. The write (*p_hwrite*) signal is driven high for a write cycle.

The core asserts transfer request (*p_htrans*= NONSEQ) during C1 to indicate that a transfer is being requested. Because the bus is currently idle, (0 transfers outstanding), the first read request to addr$_x$ is considered *taken* at the end of C1. The default slave drives an ready/OKAY response for the current idle cycle.

Clock 2 (C2):

During C2, the write data for the access is driven, and the addr$_x$ memory access takes place using the address and attribute values which were driven during C1 to enable writing of one or more bytes of memory. The slave device responds by asserting *p_hready* to indicate the cycle is completing and drives an OKAY response.

Another write transfer request is made during C2 to addr$_y$ (*p_htrans* = NONSEQ), and because the access to addr$_x$ is completing, it is considered *taken* at the end of C2.

Clock 3 (C3):

During C3, write data for addr$_y$ is driven, and the addr$_y$ memory access takes place using the address and attribute values which were driven during C2 to enable writing of one or more bytes of memory. The slave device responds by asserting *p_hready* to indicate the cycle is completing and drives an OKAY response.

Another write transfer request is made during C3 to addr$_z$ (*p_htrans* = NONSEQ), and because the access to addr$_y$ is completing, it is considered *taken* at the end of C3.

Clock 4 (C4):

During C4, write data for addr$_z$ is driven, and the addr$_z$ memory access takes place using the address and attribute values which were driven during C3 to enable reading of one or more bytes of memory. The slave device responds by asserting *p_hready* to indicate the cycle is completing and drives an OKAY response.

The CPU has no more outstanding requests, so *p_htrans* indicates IDLE. The address and attribute signals are thus undefined.

7.4.1.4 Write Transfer with Wait States

Figure 7-6 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, because the previous transaction is still outstanding. The address, transfer attributes, and write data remain driven in cycle C3 and are taken at the end of C3 because a ready/OKAY response is driven back by the slave device for the previous access. In cycle C4, a request for addr$_z$ is made. The request for access to addr$_z$ is taken at the end of C4, and during C5, a ready/OKAY response is provided by the slave device. In cycle C5, no further accesses are requested.

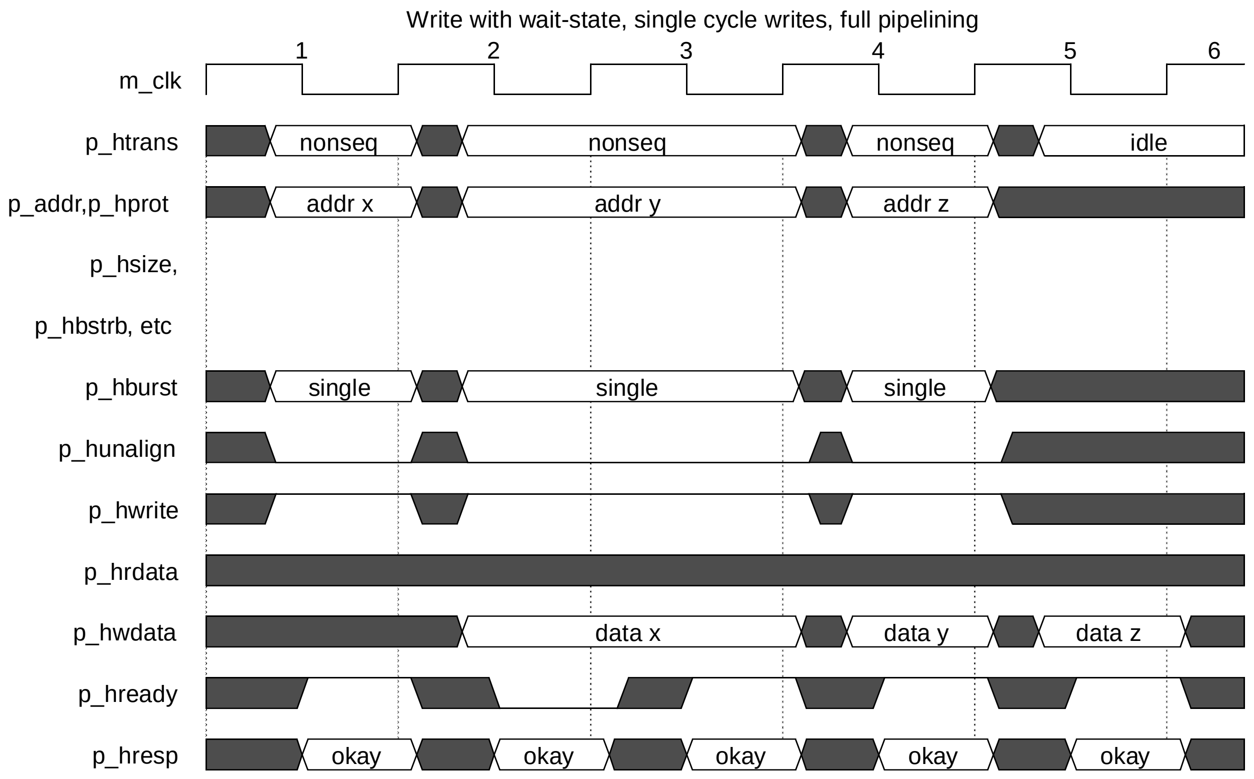


Figure 7-6. Write Transfer with Wait-State

7.4.1.5 Read and Write Transfers

Figure 7-7 shows a sequence of read and write cycles.

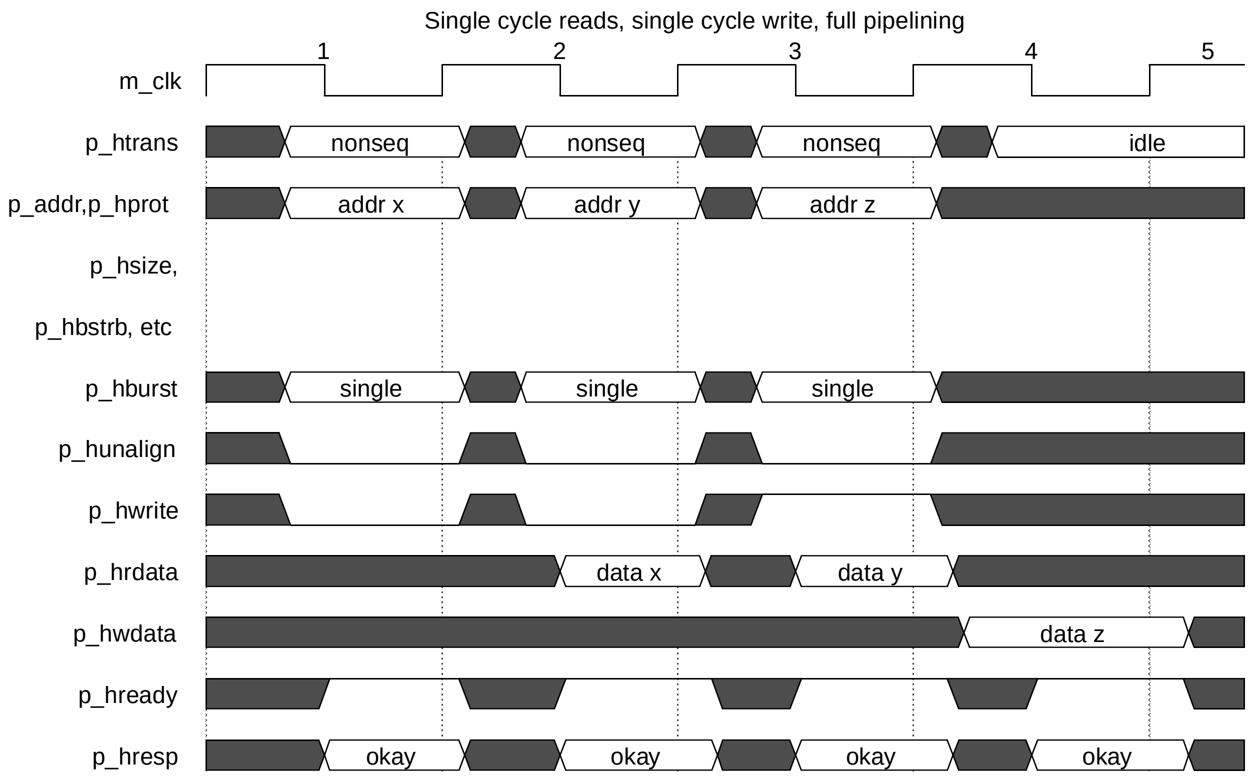


Figure 7-7. Single Cycle Read and Write Transfers

The first read request ($addr_x$) is *taken* at the end of cycle C1 because the bus is idle.

The second read request ($addr_y$) is *taken* at the end of C2 because a ready/OKAY response is asserted during C2 for the first read access ($addr_x$). During C3, a request is generated for a write to $addr_y$, which is taken at the end of C3 because the second access is terminating.

Data for the $addr_z$ write cycle is driven in C4, the cycle after the access is *taken*, and a ready/OKAY response is signaled to complete the write cycle to $addr_z$.

Figure 7-8 shows another sequence of read and write cycles. This example shows an interleaved write access between two reads.

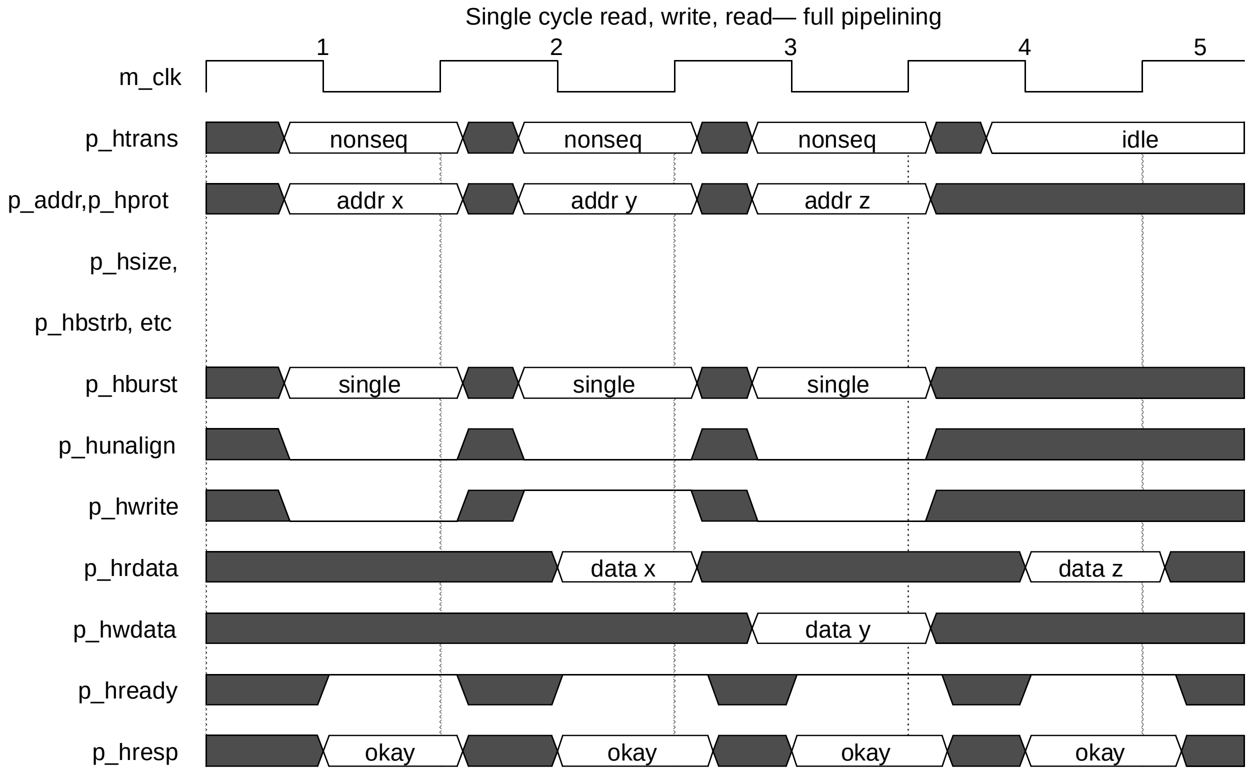


Figure 7-8. Single Cycle Read and Write Transfers—2

The first read request ($addr_x$) is *taken* at the end of cycle C1 because the bus is idle.

The first write request ($addr_y$) is *taken* at the end of C2 because the first access is terminating ($addr_x$).

Data for the $addr_y$ write cycle is driven in C3, the cycle after the access is *taken*. Also during C3, a request is generated for a read to $addr_z$, which is taken at the end of C3 because the write access is terminating.

During C4, the $addr_y$ write access is terminated, and no further access is requested

Figure 7-9 shows another sequence of read and write cycles. In this example, reads incur a single wait state.

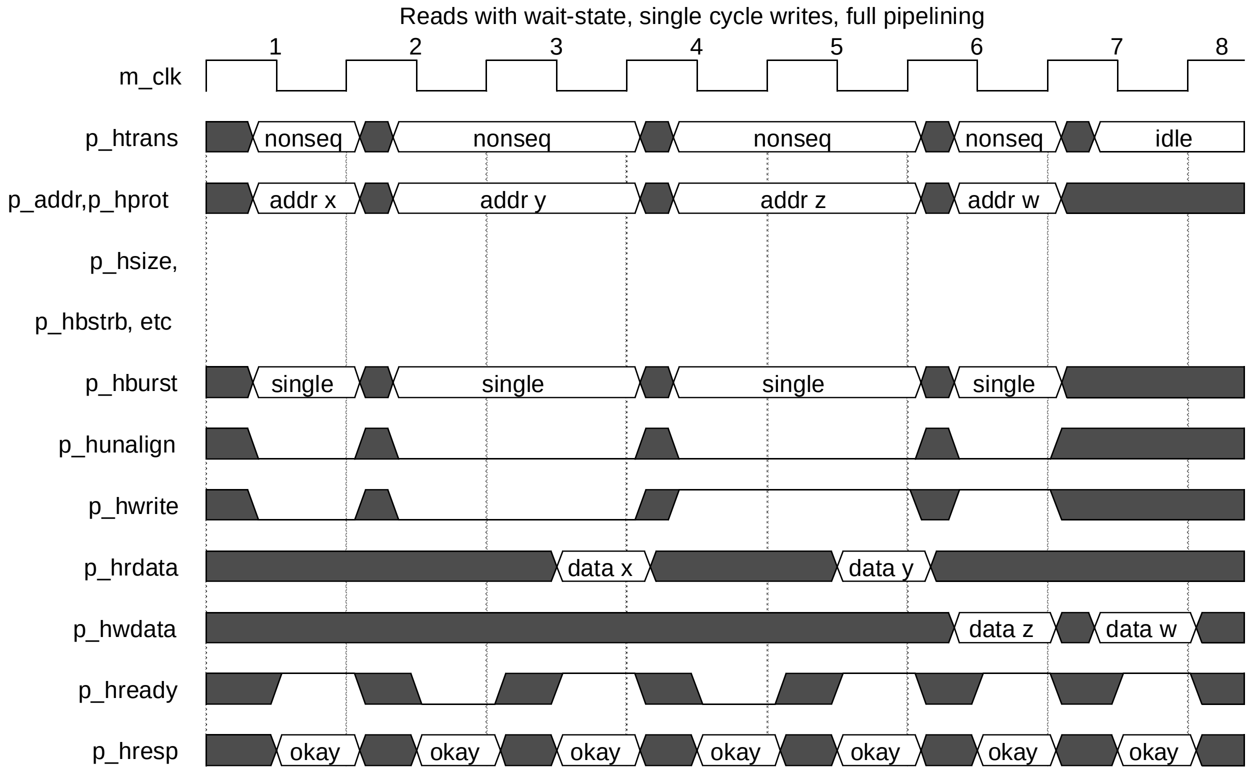


Figure 7-9. Multi-Cycle Read and Write Transfers

The first read request ($addr_x$) is *taken* at the end of cycle C1 because the bus is idle.

The second read request ($addr_y$) is not *taken* at the end of cycle C2 because no ready response is signaled and only one access can be outstanding ($addr_x$). It is taken at the end of C3 once the first read request has signaled a ready/OKAY response.

The first write request ($addr_z$) is not taken during C4 because a ready response is not asserted during C4 for the second read access ($addr_y$). During C5, the request for a write to $addr_z$ is taken because the second access is terminating.

Data for the $addr_z$ write cycle is driven in C6, the cycle after the access is *taken*.

During C6, the $addr_z$ write access is terminated and the $addr_w$ write request is *taken*.

During C7, data for the $addr_w$ write access is driven, and a ready/OKAY response is asserted to complete the write cycle to $addr_w$.

Figure 7-10 shows another sequence of read and write cycles. In this example, reads incur a single wait state.

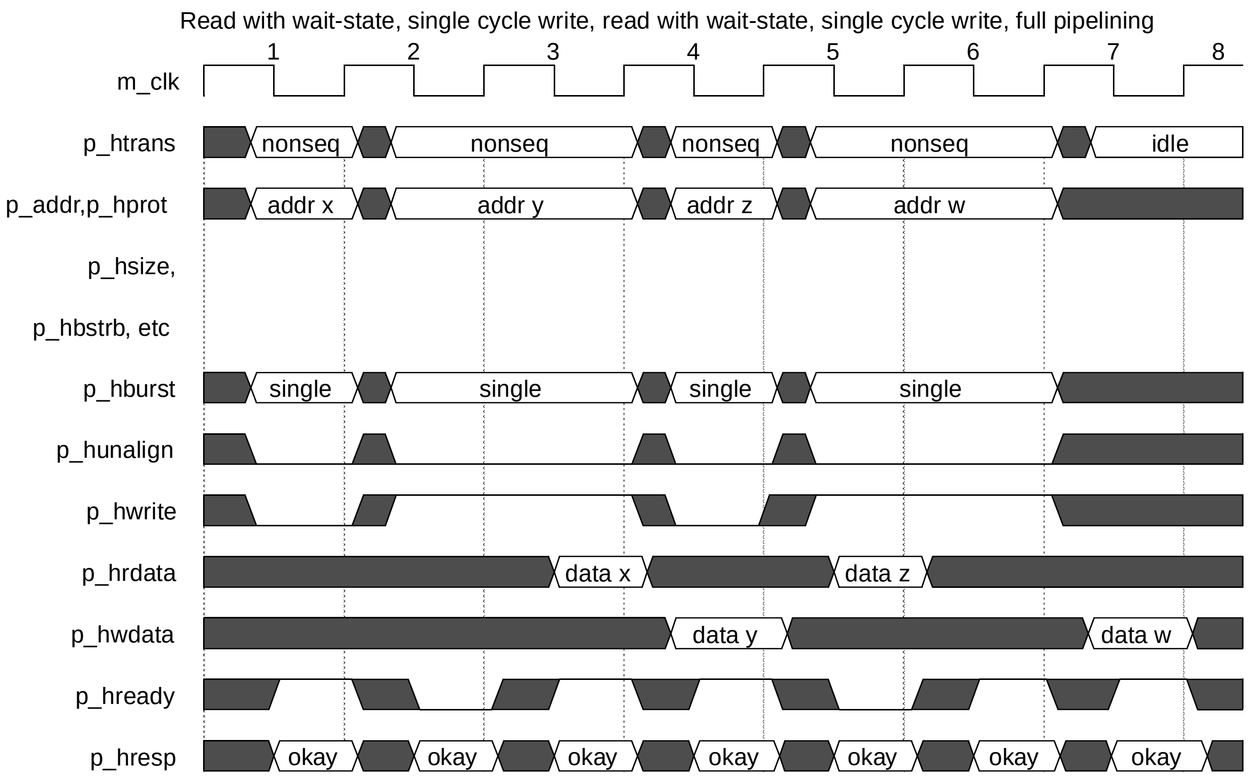


Figure 7-10. Multi-Cycle Read and Write Transfers—2

The first read request (addr$_x$) is *taken* at the end of cycle C1 because the bus is idle.

The first write request (addr$_y$) is not *taken* at the end of cycle C2 because no ready response is signaled and only one access can be outstanding (addr$_x$). It is taken at the end of C3 once the first read request has signaled a ready/OKAY response.

Data for the addr$_y$ write cycle is driven in C4, the cycle after the access is *taken*.

The second read request (addr$_z$) is taken during C4 because the addr$_y$ write is terminating.

A second write request (addr$_w$) is not taken at the end of C5 because the second read access is not terminating, 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.

7.4.1.6 Misaligned Accesses

Figure 7-11 illustrates functional timing for a misaligned read transfer. The read to addr$_x$ is misaligned across a 32-bit boundary.

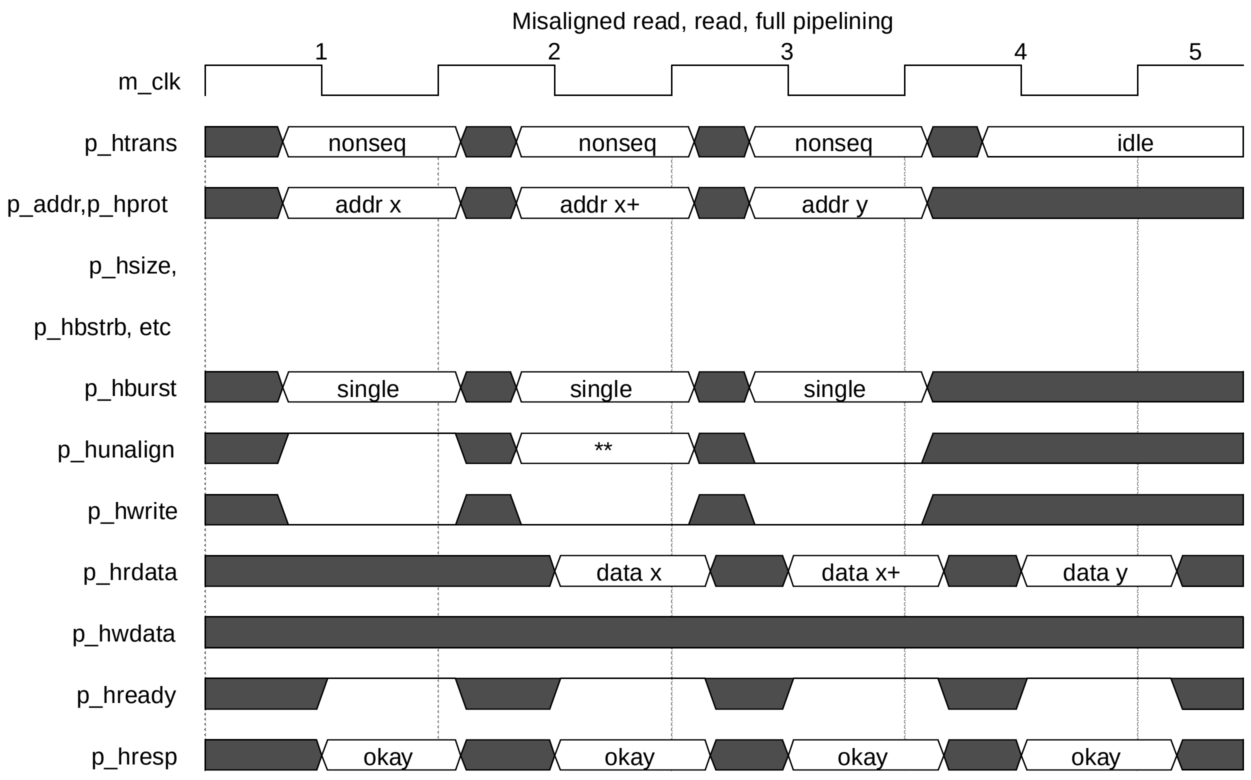


Figure 7-11. Misaligned Read Transfer

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 *p_hwrite* signal is driven low for a read cycle. The transfer size attributes (*p_hsize*) indicate the size of the transfer. Even though the transfer is misaligned, the size value driven corresponds to the size of the entire misaligned data item. *p_hunalign* is driven high to indicate that the access is misaligned. The *p_hbstrb* outputs are asserted to indicate the active byte lanes for the read, which may not correspond to size and low-order address outputs. *p_htrans* is driven to NONSEQ.

During C2, the addr$_x$ memory access takes place using the address and attribute values which were driven during C1 to enable reading of one or more bytes of memory.

The second portion of the misaligned read transfer request is made during C2 to addr$_{x+}$ (which is aligned to the next higher 32-bit boundary), and because the first portion of the misaligned access is completing, it is *taken* at the end of C2. The p_htrans signals indicate NONSEQ. The size value driven is the size of the remaining bytes of data in the misaligned read, rounded up (for the 3-byte case) to the next higher power-of-2. The *p_hbstrb* signals indicate the active byte lanes. For the second portion of a misaligned transfer, the *p_hunalign* signal is driven high for the 3-byte case (low for all others). The next read access is requested in C3 and *p_htrans* indicates NONSEQ. *p_hunalign* is negated, because this access is aligned.

Figure 7-12 illustrates functional timing for a misaligned write transfer. The write to addr$_x$ is misaligned across a 32-bit boundary.

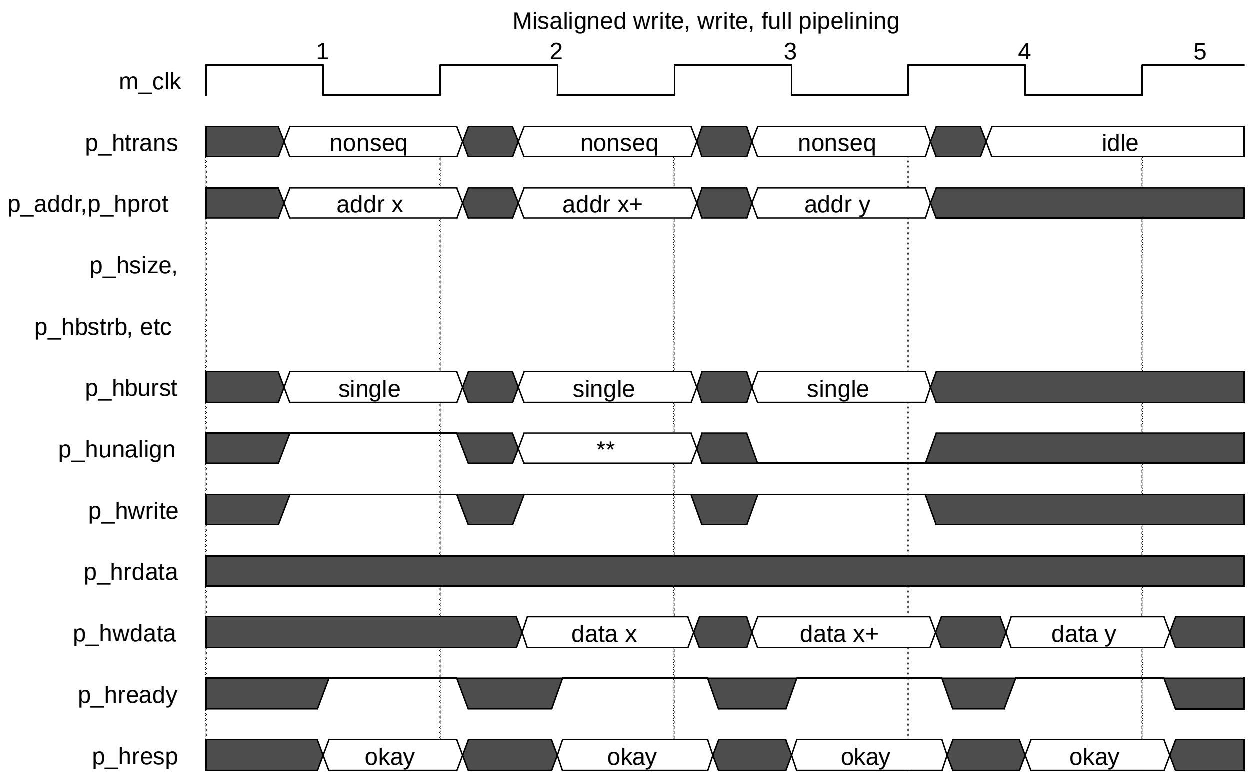


Figure 7-12. Misaligned Write Transfer

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 *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 which were driven during C1 to enable writing of one or more bytes of memory.

The second portion of the misaligned write transfer request is made during C2 to addr$_{x+}$ (which is aligned to the next higher 32-bit boundary), and because the first portion of the misaligned access is completing, it is *taken* at the end of C2. The p_htrans signals indicate NONSEQ. The size value driven is the size of the remaining bytes of data in the misaligned write, rounded up (for the 3-byte case) to the next higher power-of-2. The *p_hbstrb* signals indicate the active byte lanes. For the second portion of a misaligned transfer, the *p_hunalign* signal is driven high for the 3-byte case (low for all others).

The next write access is requested in C3 and *p_htrans* indicates NONSEQ. *p_hunalign* is negated, because this access is aligned.

An example of a misaligned write cycle followed by an aligned read cycle is shown in Figure 7-13. It is similar to the previous example in Figure 7-12.

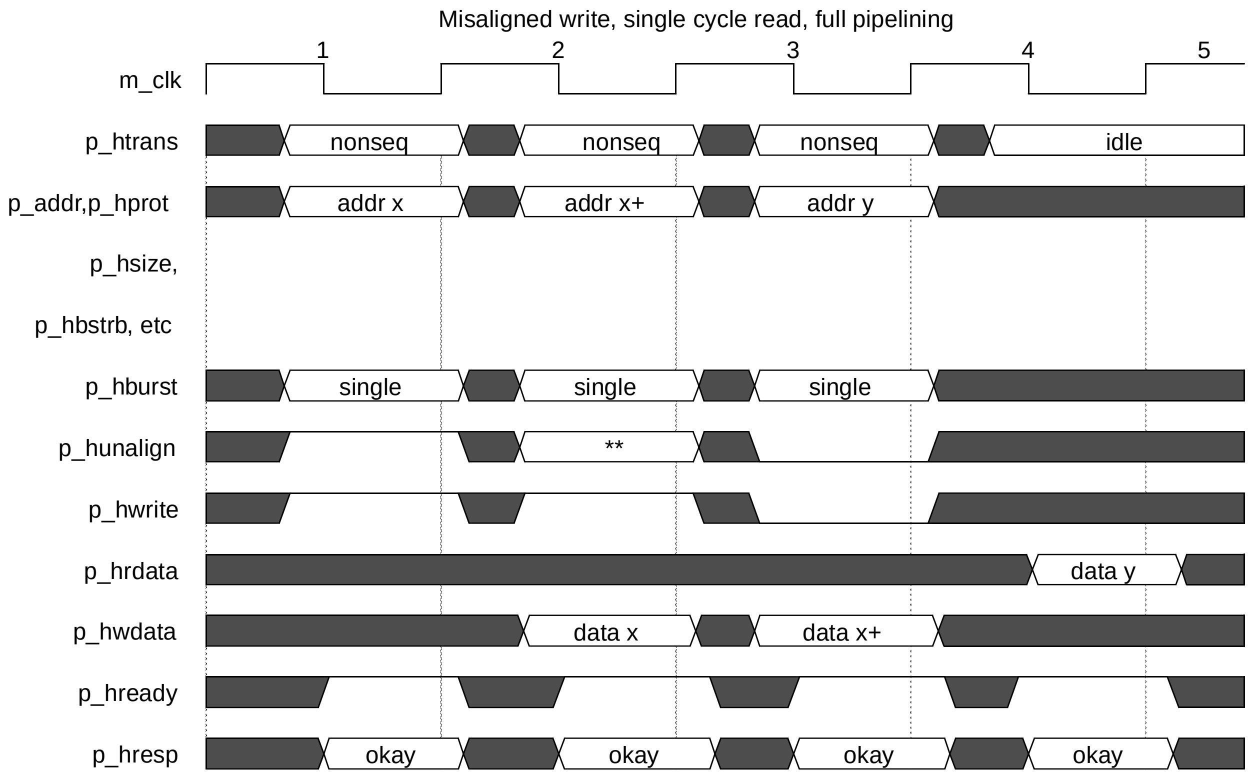


Figure 7-13. Misaligned Write, Single Cycle Read Transfer

7.4.1.7 Burst Accesses

Figure 7-14 illustrates functional timing for a burst read transfer.

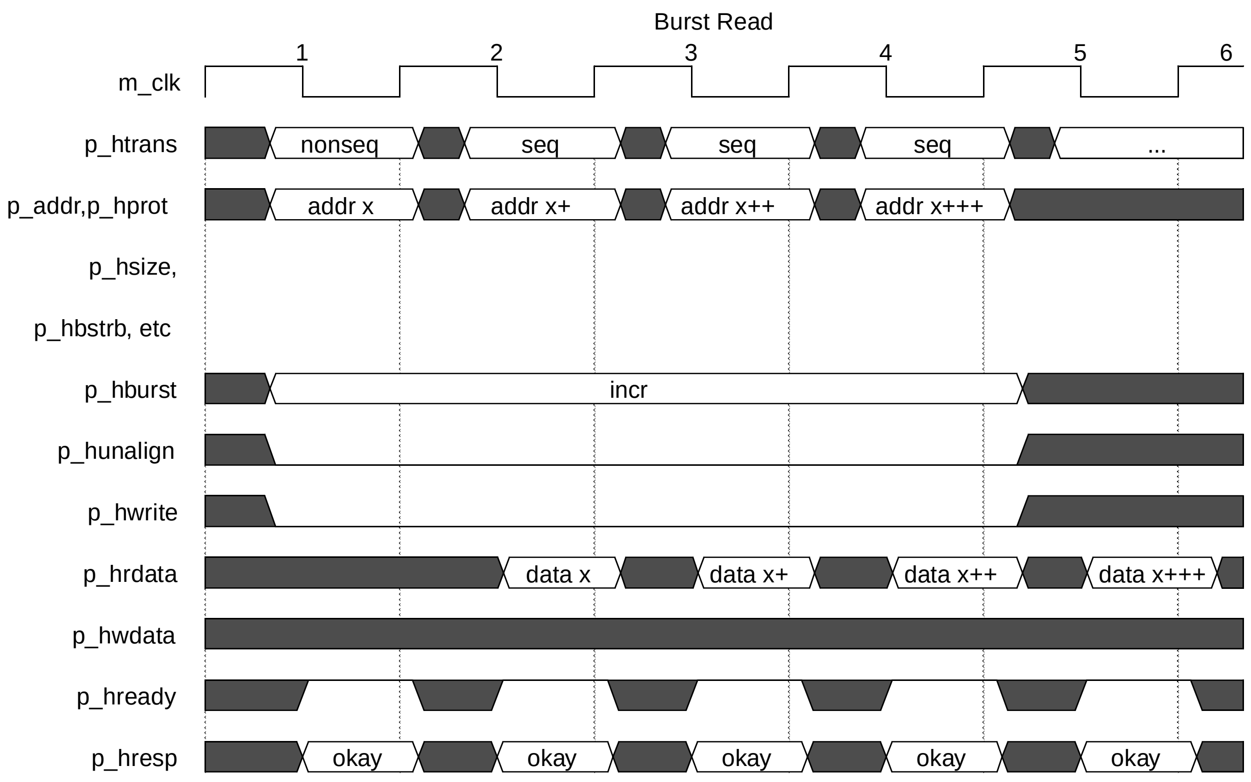


Figure 7-14. Burst Read Transfer

The *p_hburst* signals indicate INCR for all burst transfers. The *p_hunalign* signal is negated. *p_hsize* indicates 32-bits, and all four *p_hbstrb* signals are asserted. The burst address is aligned to a 32-bit boundary and increments by words. Note that in this example four beats are shown, but in operation the burst may be of any length including only a single beat.

NOTE

Bursts may be interrupted immediately at any time, and be followed by any type of cycle. No idle cycle is required.

Figure 7-15 illustrates functional timing for a burst read with wait-state transfer.

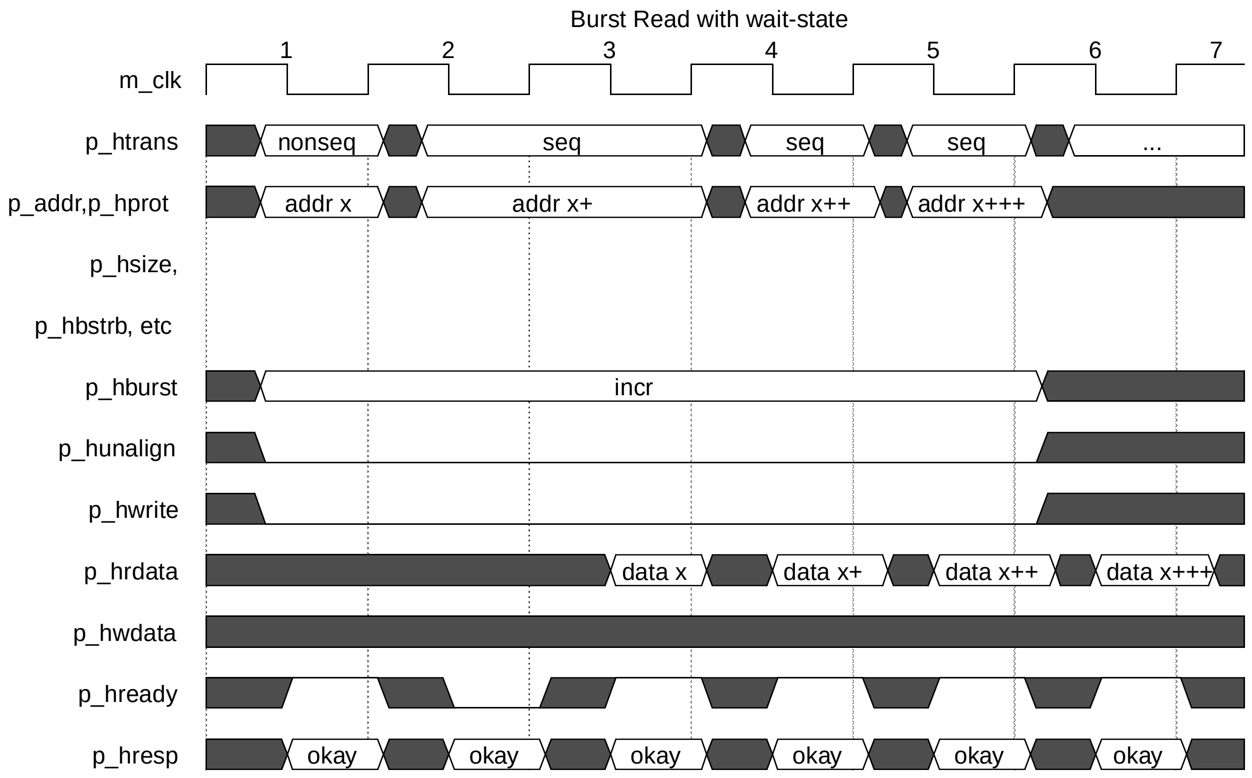


Figure 7-15. Burst Read with Wait-State Transfer

The first cycle of the burst incurs a single wait-state.

Figure 7-16 illustrates functional timing for a burst write transfer.

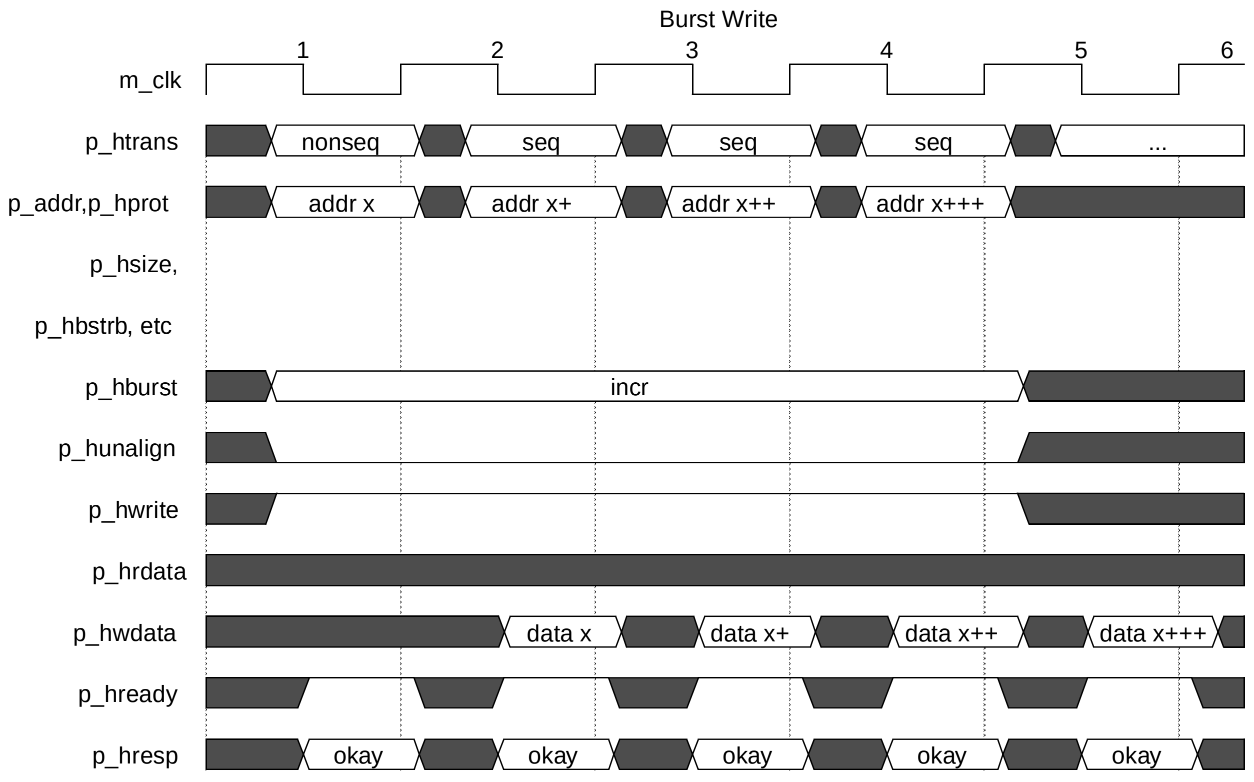


Figure 7-16. Burst Write Transfer

Figure 7-15 illustrates functional timing for a burst write with wait-state transfer.

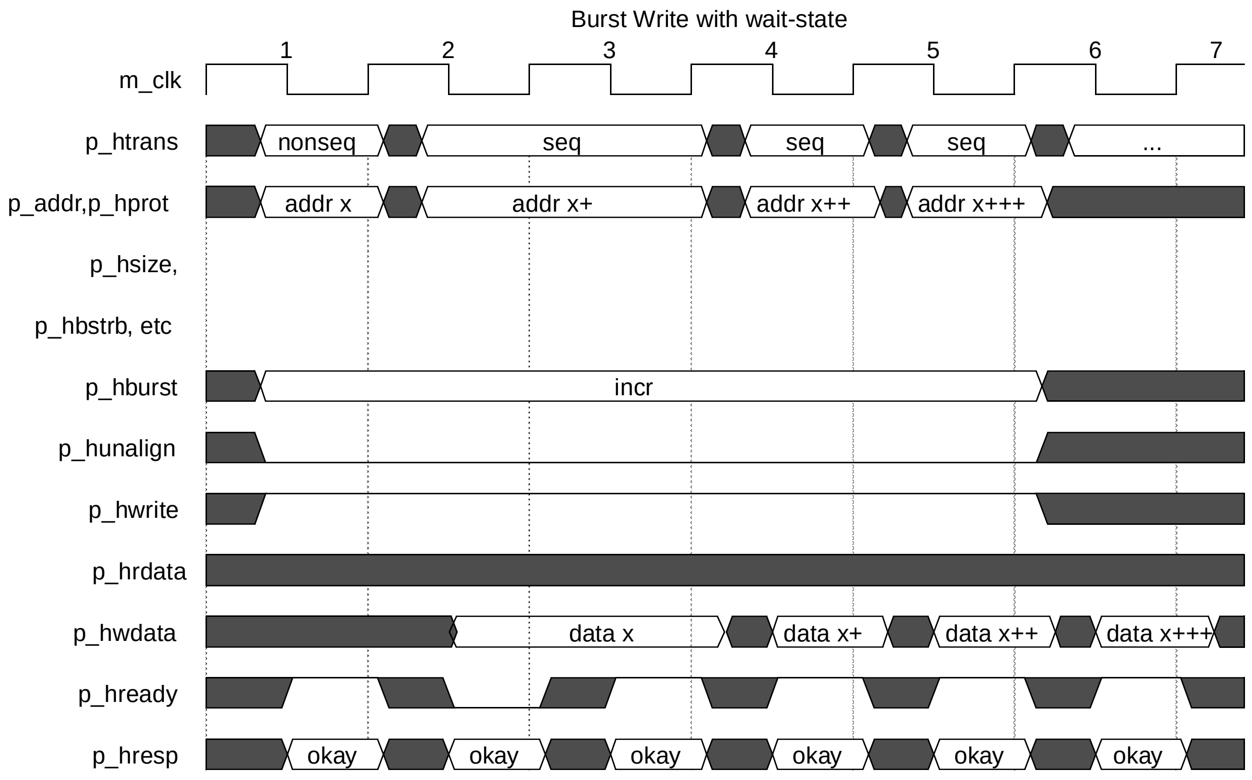


Figure 7-17. Burst Write with Wait-State Transfer

The first cycle of the burst incurs a single wait-state. Data for the second beat of the burst is valid the cycle after the second beat is *taken*.

Figure 7-18 illustrates functional timing for a pair of burst read transfers.

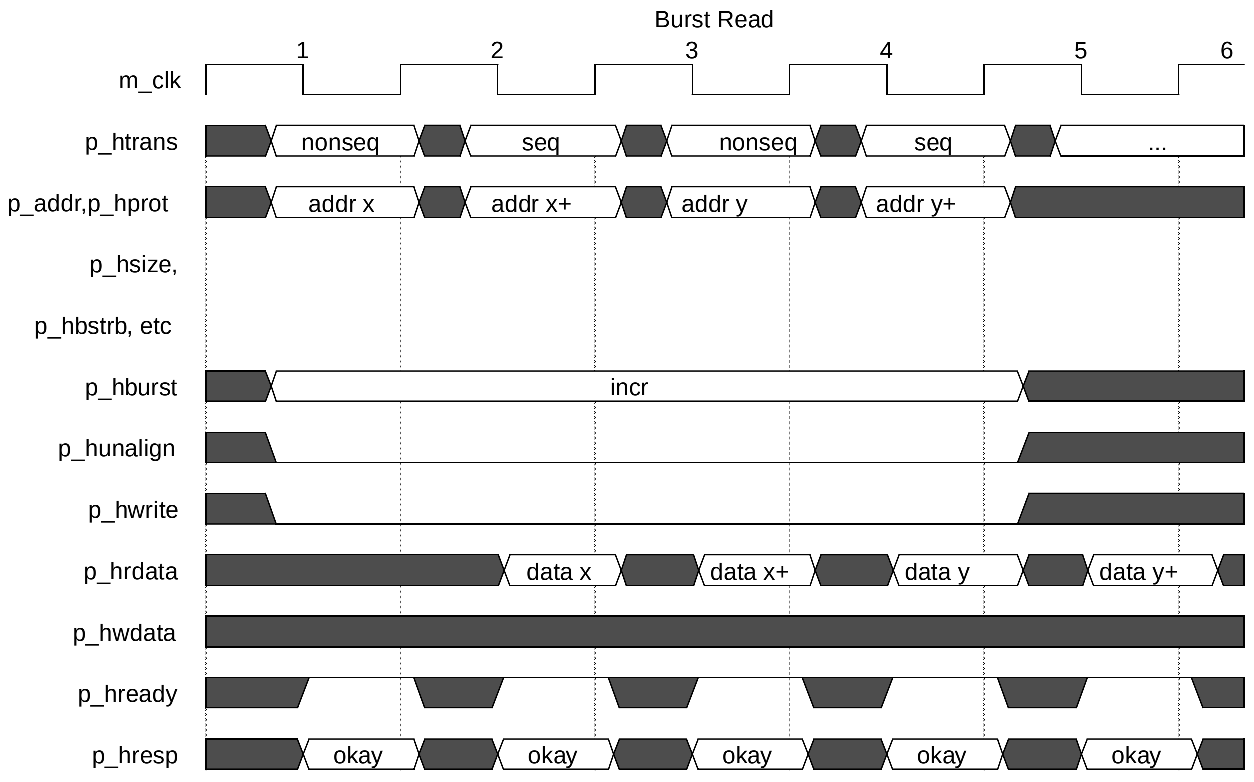


Figure 7-18. Burst Read Transfers

Note that in this example the first burst is two beats long and is followed immediately by a second burst, which is unrelated to the first.

NOTE

Bursts may be of any length (including a single beat) and may be followed immediately by any type of transfer. No idle cycles are required.

Figure 7-19 illustrates functional timing for a burst read with wait-state transfer where the second beat to addr x+ is retracted and replaced with a new burst transfer.

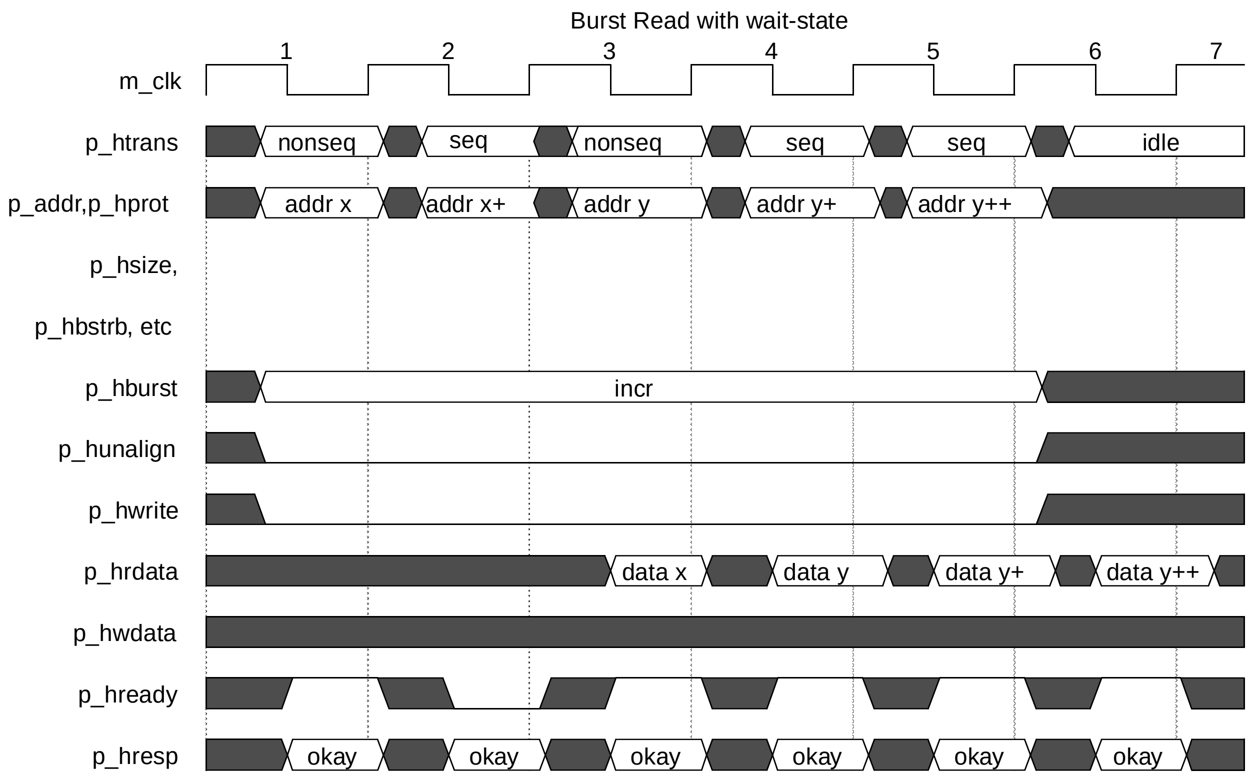


Figure 7-19. Burst Read with Wait-State Transfer, Retraction

The first cycle of the burst incurs a single wait-state, and the burst is replaced by another burst.

Figure 7-20 illustrates functional timing for a burst write transfer. The second burst is only one beat long.

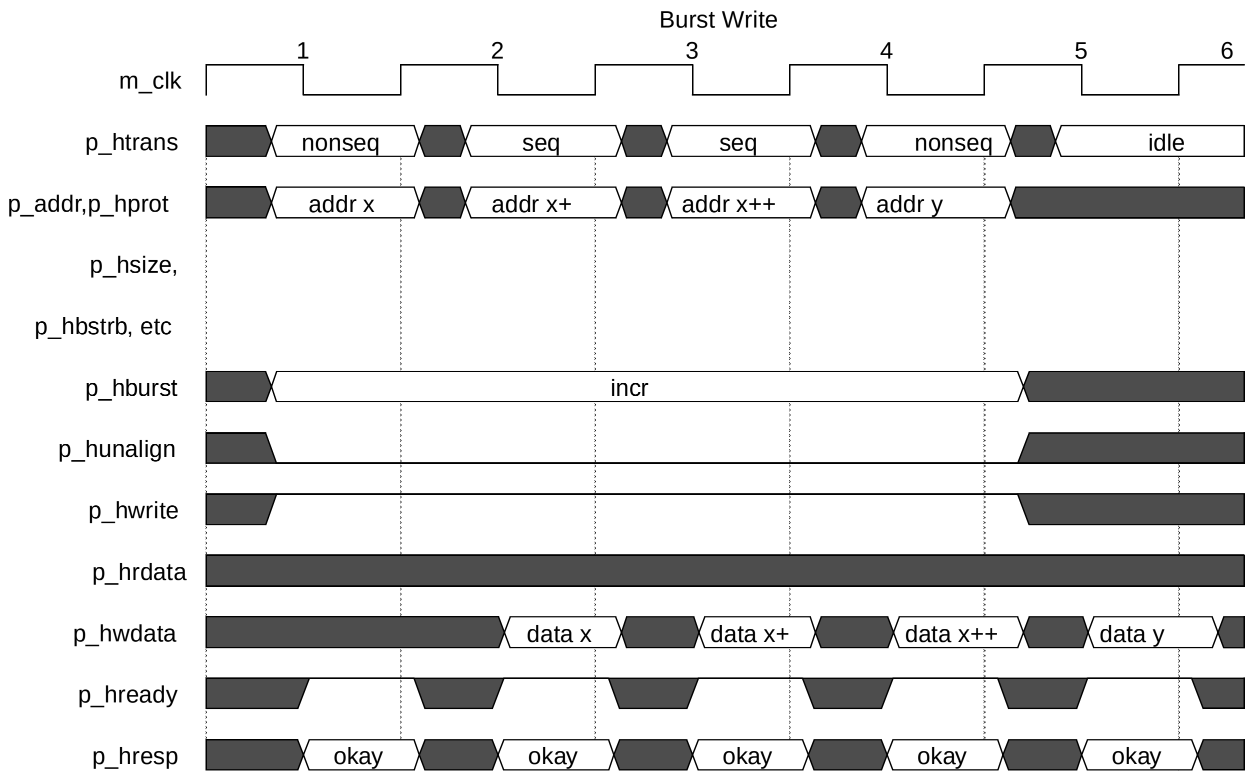


Figure 7-20. Burst Write Transfers, Single Beat Burst

This same scenario can occur for read bursts as well.

7.4.1.8 Address Retraction

Address retraction is the process of replacing an existing request with a new request unrelated to the first request. Although the AMBA AHB protocol requires an access request to remain driven unchanged once presented on the bus, higher system performance may be obtained if this aspect of the protocol is modified to allow an access request to be changed prior to being taken. e200z1 always performs address retraction under conditions in which performance may be optimized. Figure 7-21 shows an example of address retraction during wait state operation. Signal *p_hready* for the first request ($addr_x$) is not asserted during C2, so a wait state is inserted until *p_hready* is recognized (during C3).

Meanwhile, a subsequent request has been generated by the CPU for $addr_y$ which is not *taken* in C2, because the previous transaction is still outstanding. The address and transfer attributes are retracted in cycle C3, and a new access request to $addr_z$ is requested and are taken at the end of C3 because the previous access is completing. Data for $addr_x$ and a ready/OKAY response is driven back by the slave device. In cycle C4, a request for $addr_w$ is made. The request for access to $addr_w$ 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.

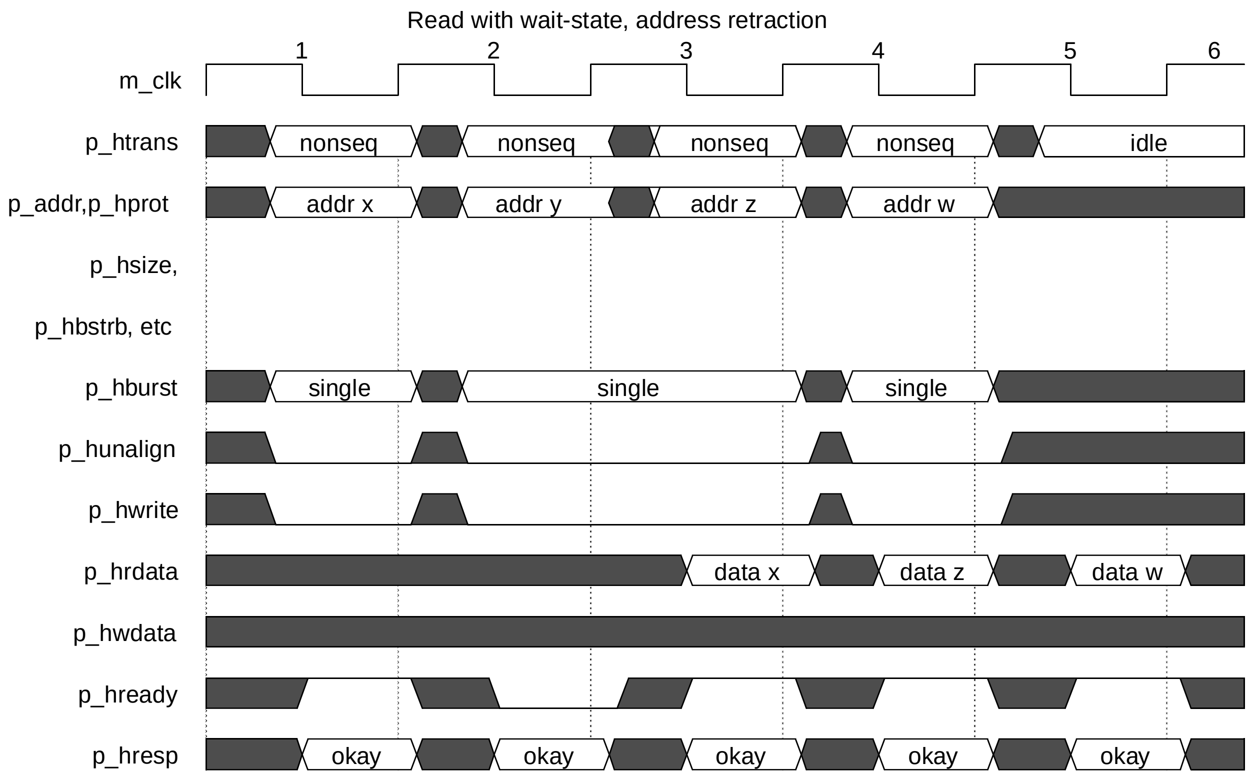


Figure 7-21. Read Transfer with Wait-State, Address Retraction

Figure 7-22 illustrates functional timing for a burst read with wait-state transfer where the second beat to addr x+ is retracted and replaced with a new burst transfer.

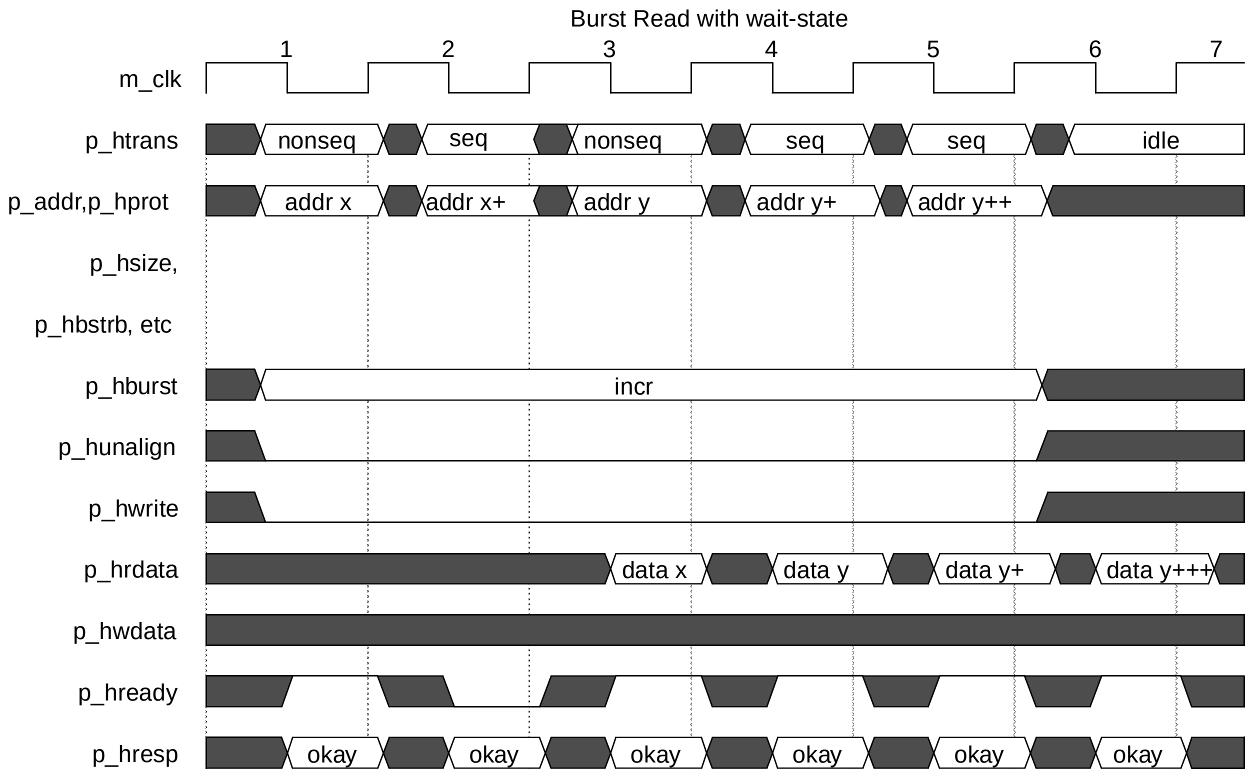


Figure 7-22. Burst Read with Wait-State Transfer, Retraction

The first cycle of the burst incurs a single wait-state, and the second beat of the burst driven in C2 burst is replaced by another burst in C3. Replacement by a single access is also possible.

Address retraction does not occur on a requested write cycle, only on read cycles, and may occur any time during a burst cycle as well.

7.4.1.9 Error Termination Operation

The *p_hresp*[2:0] inputs are used to signal an error termination for an access in progress. The ERROR encoding is used in conjunction with the assertion of *p_hready* to terminate a cycle with error. Error termination is a two-cycle termination; the first cycle consists of signaling the ERROR response on *p_hresp*[2:0] while holding *p_hready* negated, and during the second cycle, asserting *p_hready* while continuing to drive the ERROR response on *p_hresp*[2:0]. This two cycle termination allows the BIU to retract a pending access if it desires to do so. *p_htrans* may be driven to IDLE during the second cycle of the two-cycle error response, or may change to any other value, and a new access unrelated to the pending access may be requested. The cycle which may have been previously pending while waiting for a response which terminates with error may be changed. It is not required to remain unchanged when an error response is received.

Figure 7-23 shows an example of error termination.

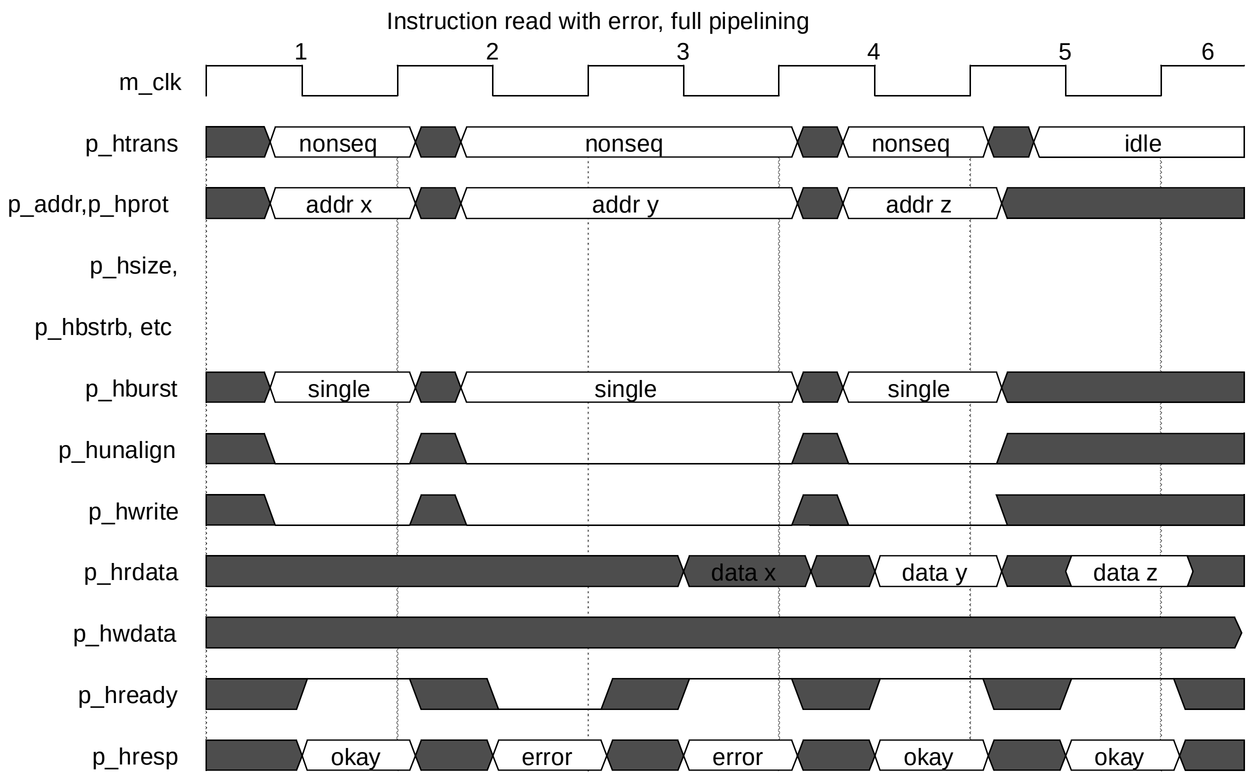


Figure 7-23. Read and Write Transfers, Instr. Read Error Termination

The first read request (addr$_x$) is *taken* at the end of cycle C1 because the bus is idle. It is an instruction prefetch.

The second read request (addr$_y$) is not *taken* at the end of C2 because the first access is still outstanding (no *p_hready* assertion). An error response is signaled by the addressed slave for addr$_x$ by driving ERROR onto the *p_hresp*[2:0] inputs. This is the first cycle of the two cycle error response protocol.

p_hready is asserted during C3 for the first read access (addr$_x$) while the ERROR encoding remains driven on *p_hresp*[2:0], terminating the access. The read data bus is undefined.

In this example of error termination, the CPU continues to request an access to addr$_y$. It is taken at the end of C3. During C4, read data is supplied for the addr$_y$ read, and the access is terminated normally during C4.

Also during C4, a request is generated for a read to addr$_z$, which is taken at the end of C4 because the second access is terminating.

Data for the addr$_z$ read cycle is provided in C5, the cycle after the access is *taken*.

During C5, a ready/OKAY response is signaled to complete the read 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 7-24 shows another example of error termination.

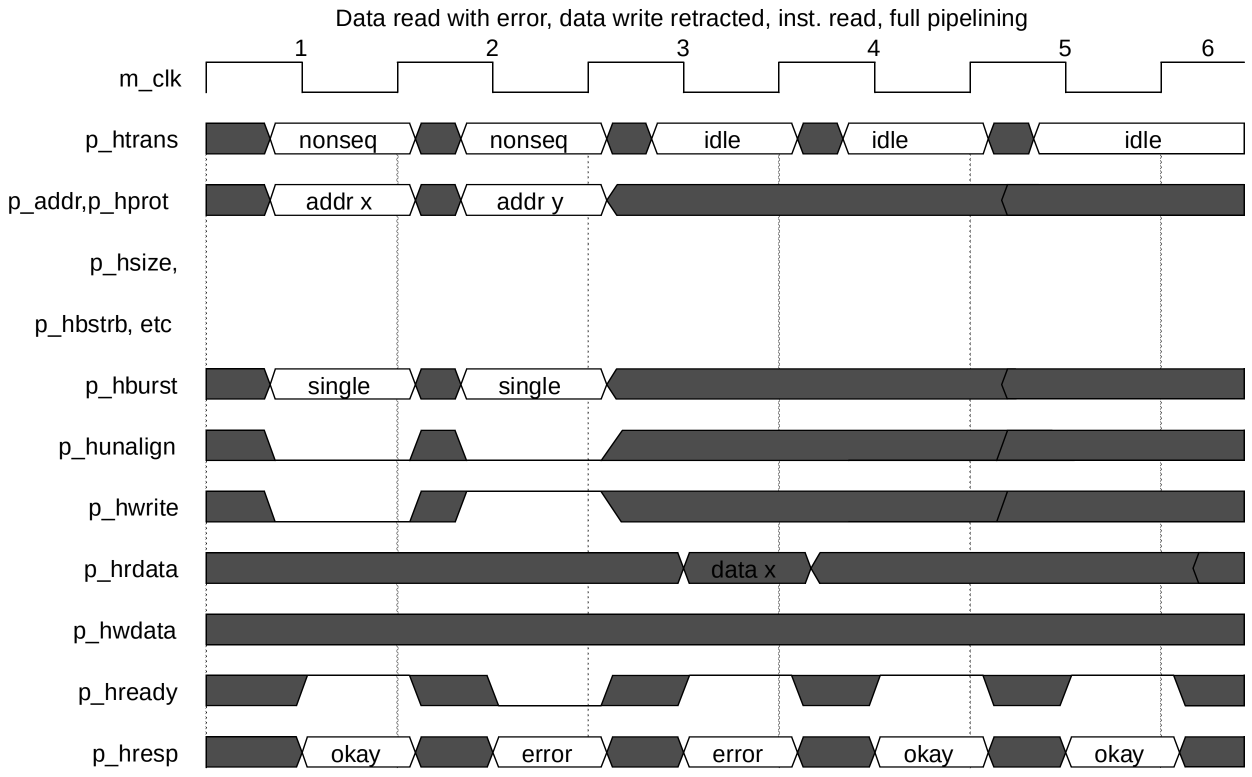


Figure 7-24. Data Read Error Termination

The first read request (addr$_x$) is *taken* at the end of cycle C1 because the bus is idle. It is a data read.

The second request (write to addr$_y$) is not *taken* at the end of C2 because the first access is still outstanding (no *p_hready* assertion). An error response is signaled by the addressed slave for addr$_x$ by driving ERROR onto the *p_hresp*[2:0] inputs. This is the first cycle of the two cycle error response protocol.

p_hready is asserted during C3 for the first read access (addr$_x$) while the ERROR encoding remains driven on *p_hresp*[2:0], terminating the access. The read data bus is undefined.

In this example of error termination, the CPU retracts the requested access to addr$_y$ by driving the *p_htrans* signals to the IDLE state during the second cycle of the two-cycle error response.

In this example of error termination, a subsequent access was aborted.

Figure 7-25 shows another example of error termination, this time on the initial portion of a misaligned write.

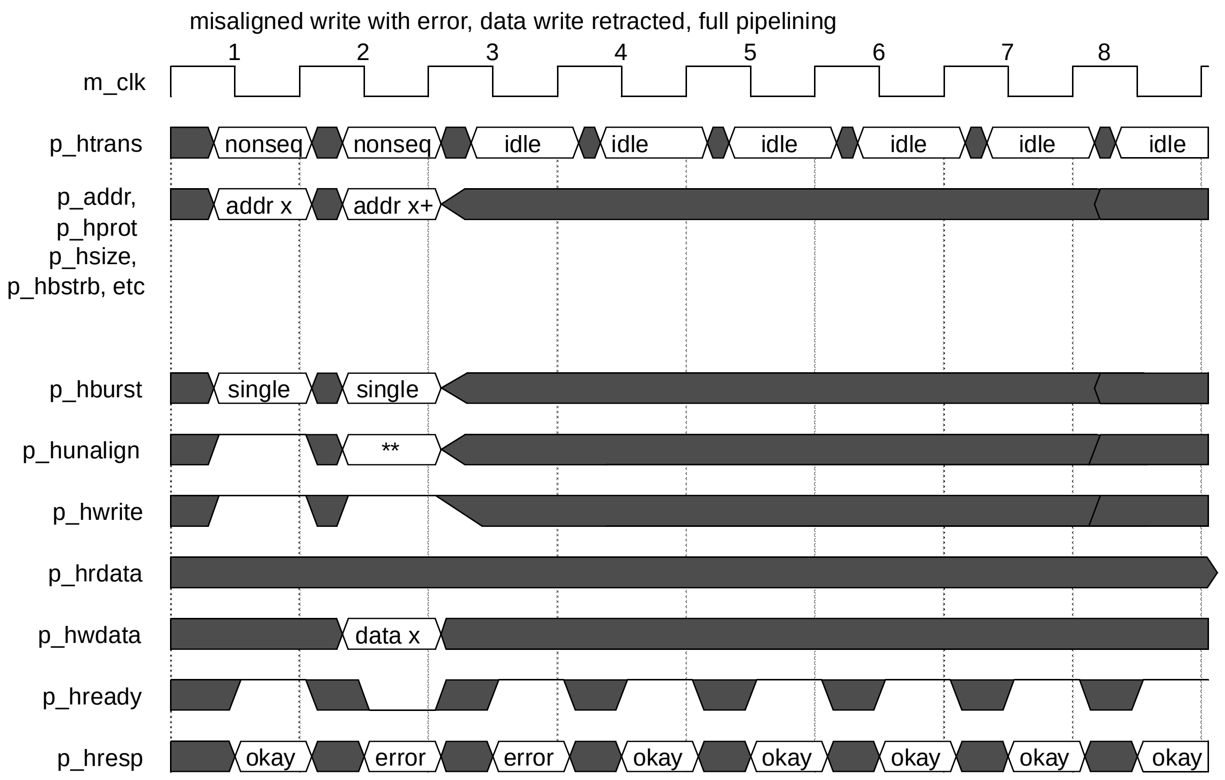


Figure 7-25. Misaligned Write Error Termination

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.

7.4.2 Power Management

The following diagram shows the relationship of the wakeup control signal *p_wakeup* to the relevant input signals.

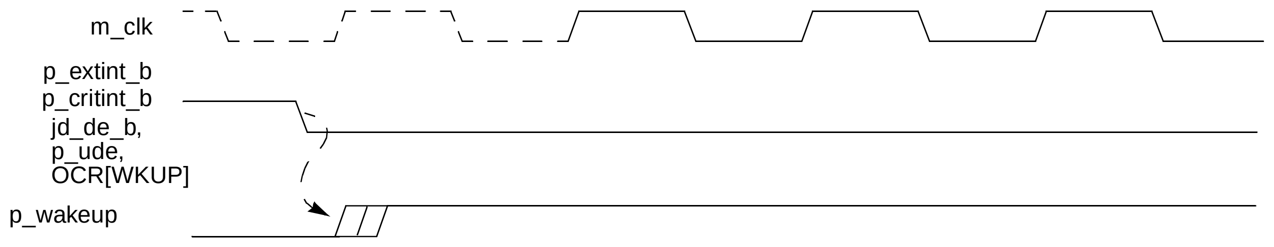


Figure 7-26. Wakeup Control Signal (*p_wakeup*)

7.4.3 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*[0:9] inputs must meet setup and hold timing relative to the rising edge of the *m_clk*. In addition, during each clock cycle in which either of the interrupt request inputs *p_extint_b* or *p_critint_b* are asserted, *p_avec_b* and *p_voffset*[0:9] are required to be in a valid state for the highest priority unmasked interrupt being requested.

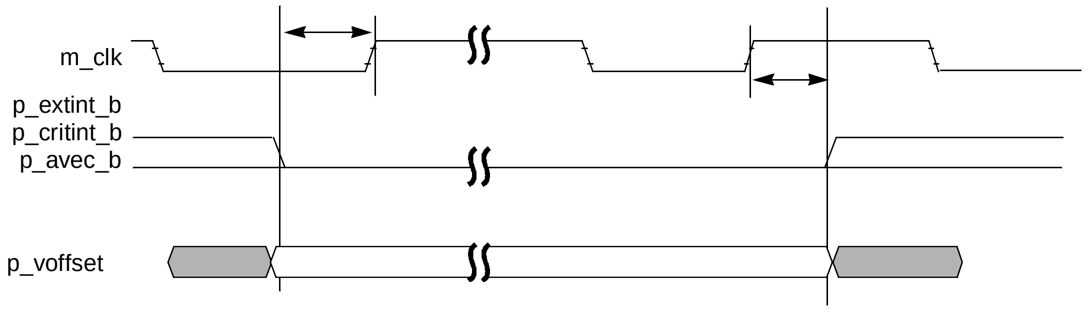


Figure 7-27. Interrupt Interface Input Signals

Figure 7-28 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* and *p_critint_b* inputs.

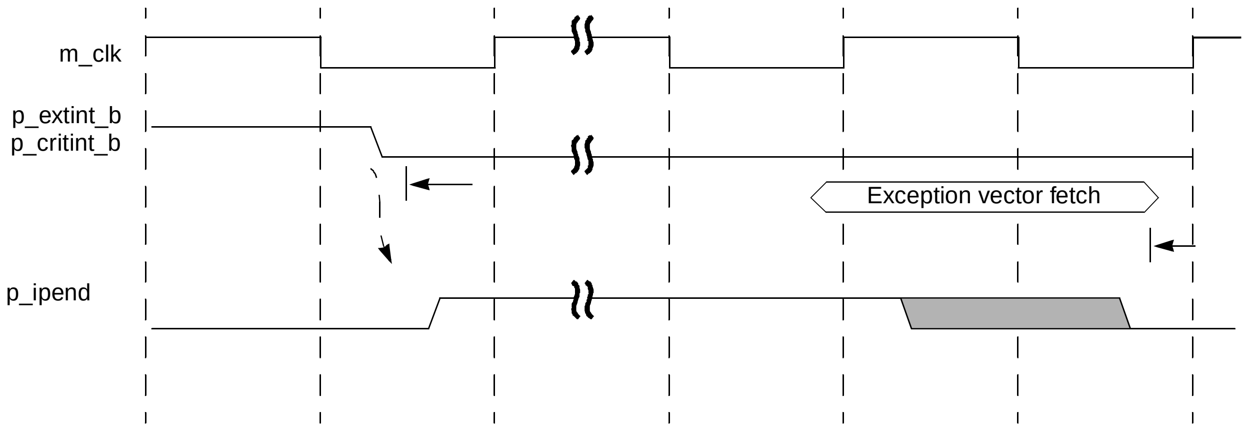


Figure 7-28. e200 Interrupt Pending Operation

Figure 7-29 shows the relationship of the interrupt acknowledge signal to the interrupt request inputs and exception vector fetching.

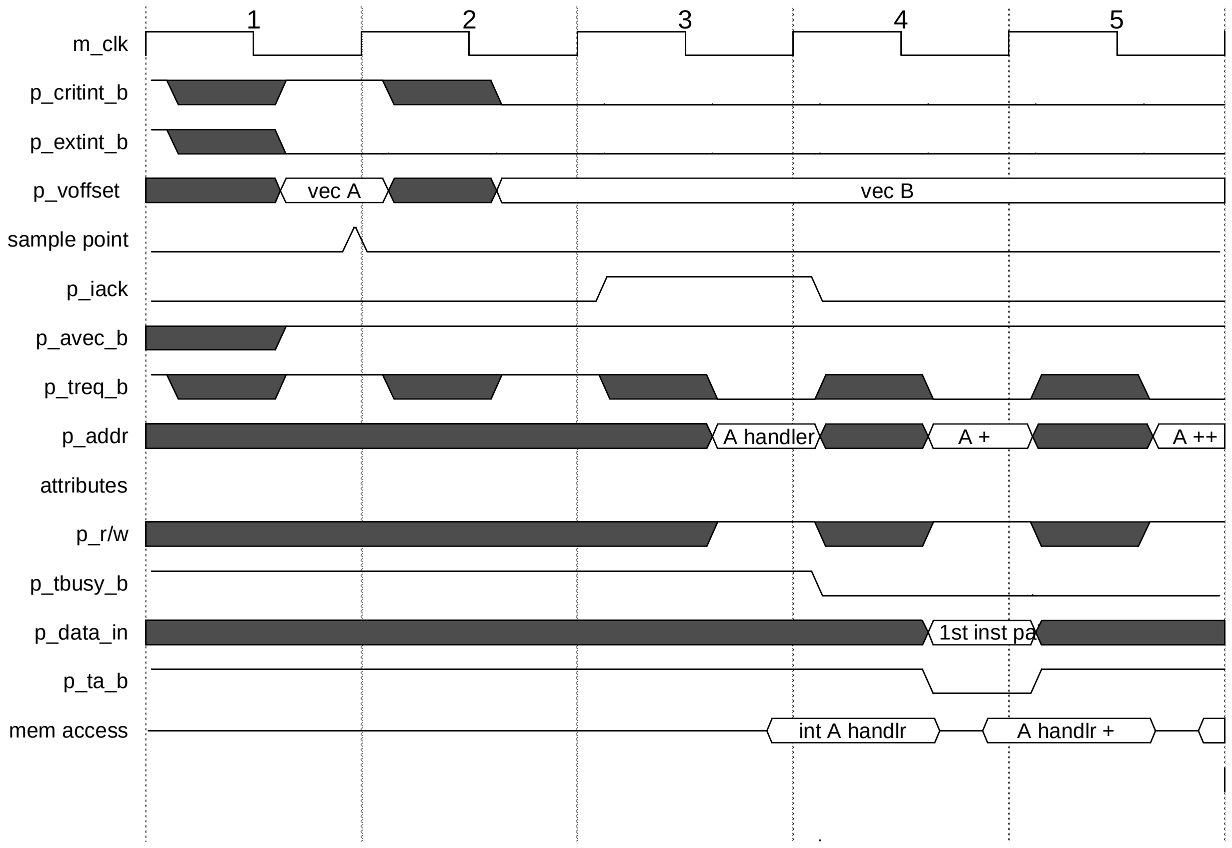


Figure 7-29. Interrupt Acknowledge Operation

In this example, an external input interrupt is requested in cycle 1. The *p_voffset*[0:9] inputs are driven with the vector offset for 'A', and *p_avec_b* is negated, indicating vectoring is desired. For this example, the bus is idle at the time of assertion. The CPU may sample a requested interrupt as early as the cycle it is initially requested, and does so in this example. The interrupt request and the vector offset and autovector input are sampled at the end of cycle 1. In cycle 3, the interrupt is acknowledged by the assertion of the *p_iack* output, indicating that the values present on interrupt inputs at the beginning of cycle 2 have been internally latched and committed to for servicing. Note that the interrupt vector lines have changed to a value of 'B' during cycle 2, and the *p_critint_b* input has been asserted by the interrupt controller. The vector number / autovector signals must be consistent with the higher priority critical input request, thus must change at the same time the state of the interrupt request inputs change. Because the *p_iack* output asserts in cycle 3, it is indicating that the values present at the rise of cycle 2 (vector 'A') have been committed to. During cycle 3, the CPU begins instruction fetching of the handler for vector 'A'. The new request for a subsequent critical interrupt 'B' was not received in time to be acted upon first. It is acknowledged after the fetch for the external input interrupt handler has been completed and has entered decode.

Note that the time between assertion of an interrupt request input and the acknowledgment of an interrupt may be multiple cycles, and the interrupt inputs may change during that interval. The CPU asserts the *p_iack* output to indicate the cycle at which an interrupt is committed to. In the following example, because the CPU was unable to acknowledge the external input interrupt during cycle 2 due to internal or external execution conditions, the critical input request was sampled. This case is shown in Figure 7-30.

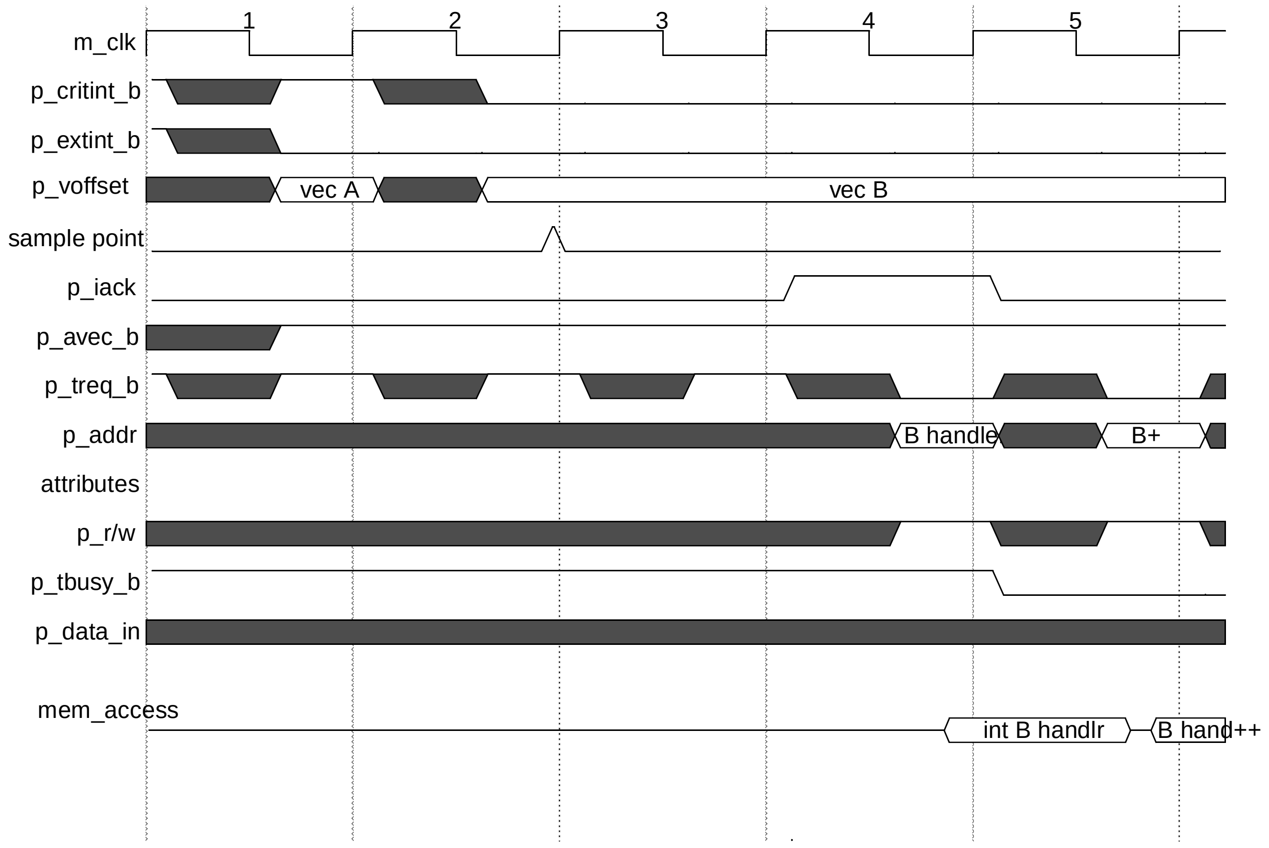


Figure 7-30. Interrupt Acknowledge Operation—2

7.4.4 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*.

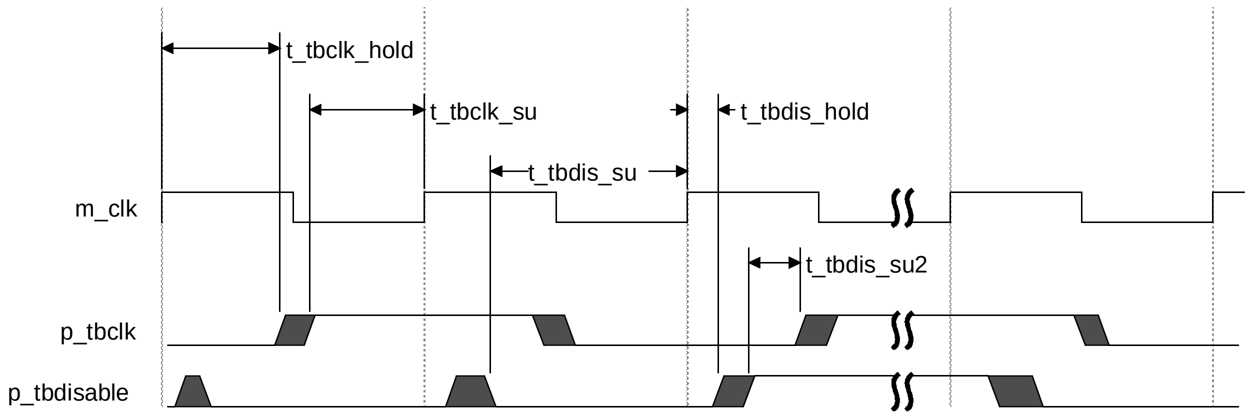


Figure 7-31. Time Base Input Timing

7.4.5 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*.

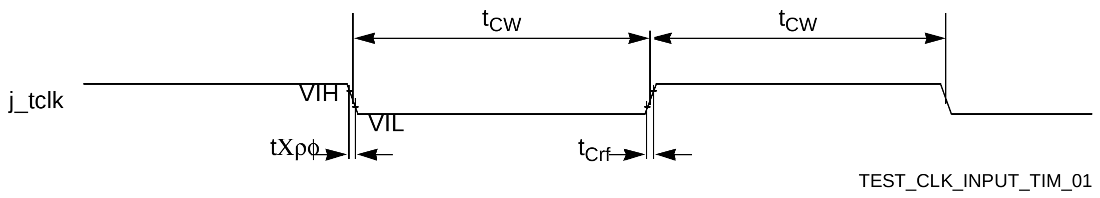


Figure 7-32. Test Clock Input Timing

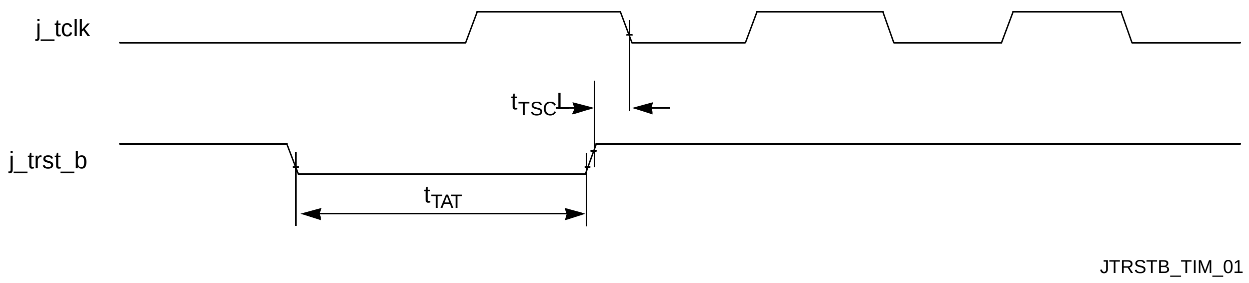


Figure 7-33. *j_trst_b* Timing

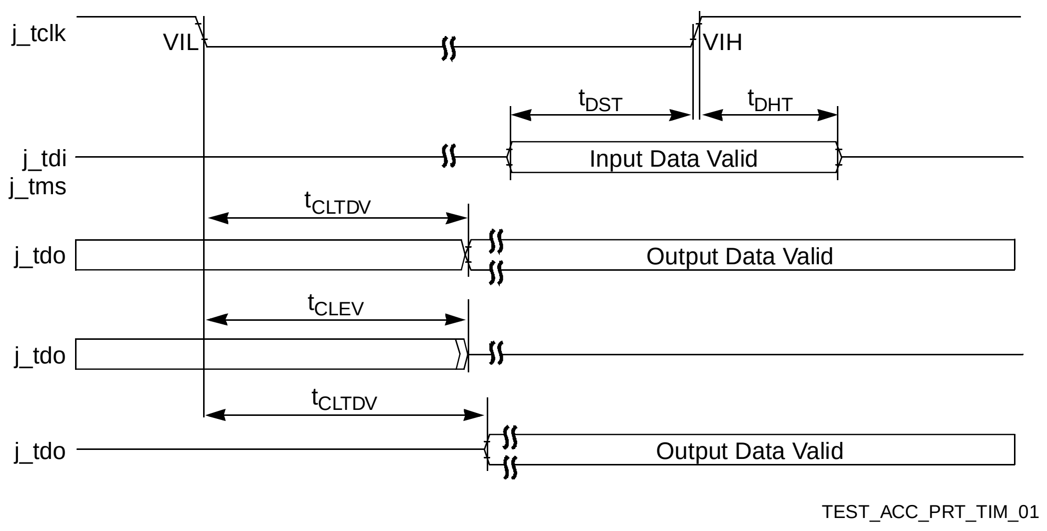


Figure 7-34. Test Access Port Timing

Chapter 8
Power Management

8.1 Power Management

Power management is supported by e200 cores to minimize overall system power consumption. The e200 core provides the ability to initiate power management from external sources as well as through software techniques. The power states on the e200 core are described below.

8.1.1 Active State

The Active state is the default state for the e200 core in which all of its internal units are operating at full processor clock speed. In this state, the e200 core still provides dynamic power management in which individual internal functional units may stop clocking automatically whenever they are idle.

8.1.2 Waiting State

The e200 core enters the Waiting state as a result of executing a **wait** instruction. Following entry into the waiting state, instruction execution and bus activity is suspended. Most internal clocks are gated off in this state. The e200 core asserts **p_waiting** to indicate it is in the waiting state. Prior to entering the waiting state, all outstanding instructions and bus transactions will be completed. The **m_clk** input should remain running while in the waiting state to allow for interrupt sampling, to allow further transitions into the Halted or Stopped state if requested and to keep the Time Base operational if it is using **m_clk** as the clock source.

In the waiting state, the core is waiting for a valid unmasked pending interrupt request. Once a pending interrupt request is received, the core 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), or a machine check interrupt (p_mcp_b assertion, etc.). Once the interrupt processing begins, the core will not return to the waiting state until another **wait** instruction is executed.

The waiting state can be temporarily exited and returned to if a request is made to enter hardware debug mode (various mechanisms), the Halted state, or the Stopped state. After exiting one of these states, the processor will return to the waiting state. While temporarily exited, the **p_waiting** output will negate, and will be re-asserted once the CPU returns to the waiting state.

8.1.3 Halted State

Instruction execution and bus activity is suspended in the Halted state. However, none of the internal clocks are gated off in this state. The e200 core asserts **p_halted** to indicate it is in the halted state. Prior to entering the halted state, all outstanding bus transactions will be completed. The **m_clk** input should remain running while in the Halted state to allow further transitions into the Stopped state if requested and to keep the Time Base operational if it is using **m_clk** as the clock source.

8.1.4 Stopped State

The Stopped state is characterized as having all internal functional units of the e200 core stopped except the Time Base unit and the clock control state machine logic. The internal **m_clk** may be kept running to keep the Time Base active and to allow quick recovery to the full on state. Clocks are not running to functional units in this state except for the Time Base. The Stopped state is reached after transitioning through the Halted state with the **p_stop** input asserted. The **p_stopped** output signal will be asserted once the Powerdown state is reached.

While in the Stopped state, further power savings may be achieved by disabling the Time Base by asserting **p_tbdisable**, or by stopping the **m_clk** input. This is done externally by the system after the e200 core is safely in the Stopped state and has asserted the **p_stopped** output signal. To exit from the Stopped state, the system must first restart the **m_clk** input.

Since the Time Base unit is off during the Stopped state if it is using **m_clk** as the clock source and **m_clk** is stopped, or if the Time Base clocking is disabled by the assertion of **p_tbdisable**, system software must usually have to access an external time base source after returning to the full on state in order to re-initialize the Time Base unit. In addition, it will not be possible to use a Time Base related interrupt source to exit low power states.

e200 also provides the capability of clocking the Time Base from an independent (but externally synchronized) clock source which would allow the Time Base to be maintained during the Powerdown state, and would allow a Time Base related interrupt to be generated to indicate an exit condition from the Stopped state.

8.1.5 Power Management Pins

p_waiting—output pin asserted when the e200 core is in the Waiting state.

p_halt—input pin is asserted by system logic to request the core to go into the Halted state. Negating this pin causes the e200 core to transition back into the Active or Waiting state if **p_stop** is also negated.

p_halted—output pin asserted when the e200 core is in the Halted state.

p_stop—input pin is asserted by system logic to request that the e200 core go into the Powerdown state. Negating this pin causes the e200 core to transition back into the Halted state from the Stopped state.

p_stopped—output pin asserted when the e200 core is in the Stopped state.

p_tbdisable—input pin is asserted by system logic when clocking of the Time Base should be disabled.

p_tbint—output pin is asserted when an internal Time Base interrupt request is signalled.

p_doze, **p_nap**, and **p_sleep** output pins that reflects the state of HID0[DOZE]. HID0[NAP]., and HID0[SLEEP] respectively. These pins are qualified with MSR[WE] = 1. Interpretation of these signals is done by the system logic.

p_wakeup—output pin asserted when an interrupt is pending or other condition which requires the clock to be running.

8.1.6 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 e200 core is intended to be the halted state with the clocks running.
- HID0[NAP]—The interpretation of the nap mode bit is done by the external system logic. Nap mode on the e200 core may be used for a powerdown state with the Time Base enabled.
- HID0[SLEEP]—The interpretation of the sleep mode bit is done by the external system logic. Sleep mode on the e200 core may be used for a powerdown state with the Time Base disabled.

8.1.7 Software Considerations for Power Management using Wait Instructions

Executing a **wait** instruction causes the e200 core to complete instruction fetch and execution activity and await an interrupt. The **p_waiting** output is asserted once the Waiting state is entered. External system hardware may interpret the state of this signal and activate the **p_halt** and/or **p_stop** inputs to cause the e200 core to enter a quiescent state in which clocks may be disabled for low power operation. Alternatively, system hardware may utilize some other clock control mechanism while the processor is in the Waiting state, and **p_wakeup** remains negated.

8.1.8 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 HID0[DOZE], HID0[NAP], and HID0[SLEEP] respectively. Note that the e200 core is not affected by assertion of these pins directly. External system hardware may interpret the state of these signals and activate the **p_halt** and/or **p_stop** inputs to cause the e200 core to enter a quiescent state in which clocks may be disabled for low power operation.

To ensure a clean transition into and out of a power saving mode, the following program sequence is recommended:

```
sync
mtmsr (WE)
isync
loop:br loop  (optionally use a wait instruction)
```

An interrupt is typically used to exit a power saving state. The **p_wakeup** output is used to indicate to the system logic that an interrupt (or a debug request) has become pending. System logic uses this output to re-enable the clocks and exit a low power state. The interrupt handler is responsible for determining how to exit the low power branching loop if one is used. Wait instructions will be exited automatically. The vectored interrupt capability provided by the core may be useful in assisting the determination if an external hardware interrupt is used to perform the wake-up.

8.1.9 Debug Considerations for Power Management

When a debug request is presented to the e200 core while in either the Waiting, Halted or Stopped state, the **p_wakeup** signal 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** or **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 9
Debug Support

This chapter describes the debug features of the e200 core.

9.1 Overview

Internal debug support in the e200 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 are protected from software modification.

Software debug facilities are defined as part of Power Architecture Book E. e200 supports a subset of these defined facilities. In addition to the facilities defined in Power Architecture Book E, e200 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 e200 core provides support for an external Nexus real-time debug module. Real-time debugging in a e200-based system is supported by a Nexus class 1 module.

The e200 core also provides support for run-time integrity checking via a Parallel Signature unit, which is capable of monitoring the CPU data read and data write AHB buses, and accumulating a pair of 32-bit MISR signatures of the data values transferred over these buses.

9.1.1 Software Debug Facilities

e200 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 (DBCR0-3), a set of address compare registers (IAC1, IAC2, IAC3, IAC4, DAC1, and DAC2), a configurable Debug Counter, a Debug Status Register (DBSR) for enabling and recording various kinds of debug events, and a special Debug interrupt type built into the interrupt mechanism (see Section 5.7.16, "Debug Interrupt (IVOR15)"). The debug facilities also provide a mechanism for software-controlled processor reset, and for controlling the operation of the timers in a debug environment.

Software debug facilities are enabled by setting the internal debug mode bit in Debug Control register 0 (DBCR0[IDM]). When internal debug mode is enabled, debug events can occur, and can be enabled to record exceptions in the Debug Status register (DBSR). If enabled by MSR[DE], these recorded exceptions cause Debug interrupts to occur. When DBCR0[IDM] is cleared, (and DBCR0[EDM] is cleared as well), no debug events occur, and no status flags are set in DBSR unless already set. In addition, when DBCR0[IDM] is cleared (or is overridden by DBCR0[EDM] being set) no Debug interrupts occur, regardless of the contents of DBSR. A software Debug interrupt handler may access all system resources and perform necessary functions appropriate for system debug.

9.1.1.1 Power Architecture Book E Compatibility

The e200 core implements a subset of the Power Architecture Book E internal debug features. The following restrictions on functionality are present:

- Instruction address compares do not support compare on physical (real) addresses.
- Data address compares do not support compare on physical (real) addresses.
- Data value compares are not supported.

9.1.2 Additional Debug Facilities

In addition to the debug functionality defined in Power Architecture Book E, e200 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.

e200 also defines two new debug events (CIRPT, CRET) for debugging around critical interrupts.

In addition, e200 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** and **se_rfdi** instruction.

9.1.3 Hardware Debug Facilities

The e200 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 e200 core, read and write internal state and debug facilities, single-step instructions, and resume normal execution.

Hardware Debug is enabled by setting the External Debug Mode enable bit in Debug Control register 0 (DBCR0[EDM]). Setting DBCR0[EDM] overrides the Internal Debug Mode enable bit DBCR0[IDM]. When the Hardware Debug facility is enabled, software is blocked from modifying the debug facilities. In addition, because resources are “owned” by the Hardware debugger, inconsistent values may be present if software attempts to read debug-related resources.

When hardware debug is enabled by setting DBCR0[EDM]=1, the registers and resources described in Section 9.3, “Debug Registers,” are reserved for use by the external debugger. The same events described in Section 9.2, “Software Debug Events and Exceptions,” are also used for external debugging, but exceptions are not generated to running software. Debug events enabled in the respective DBCR[0–3] registers are recorded in the DBSR regardless of MSR[DE], and no debug interrupts are generated. Instead, the CPU enters debug mode when an enabled event causes a DBSR bit to become set. DBCR0[EDM] may only be written through the OnCE port.

Access to most debug resources (registers) requires that the core clock (*m_clk*) be running in order to perform write accesses from the external hardware debugger.

Figure 9-1 shows the e200 debug resources.

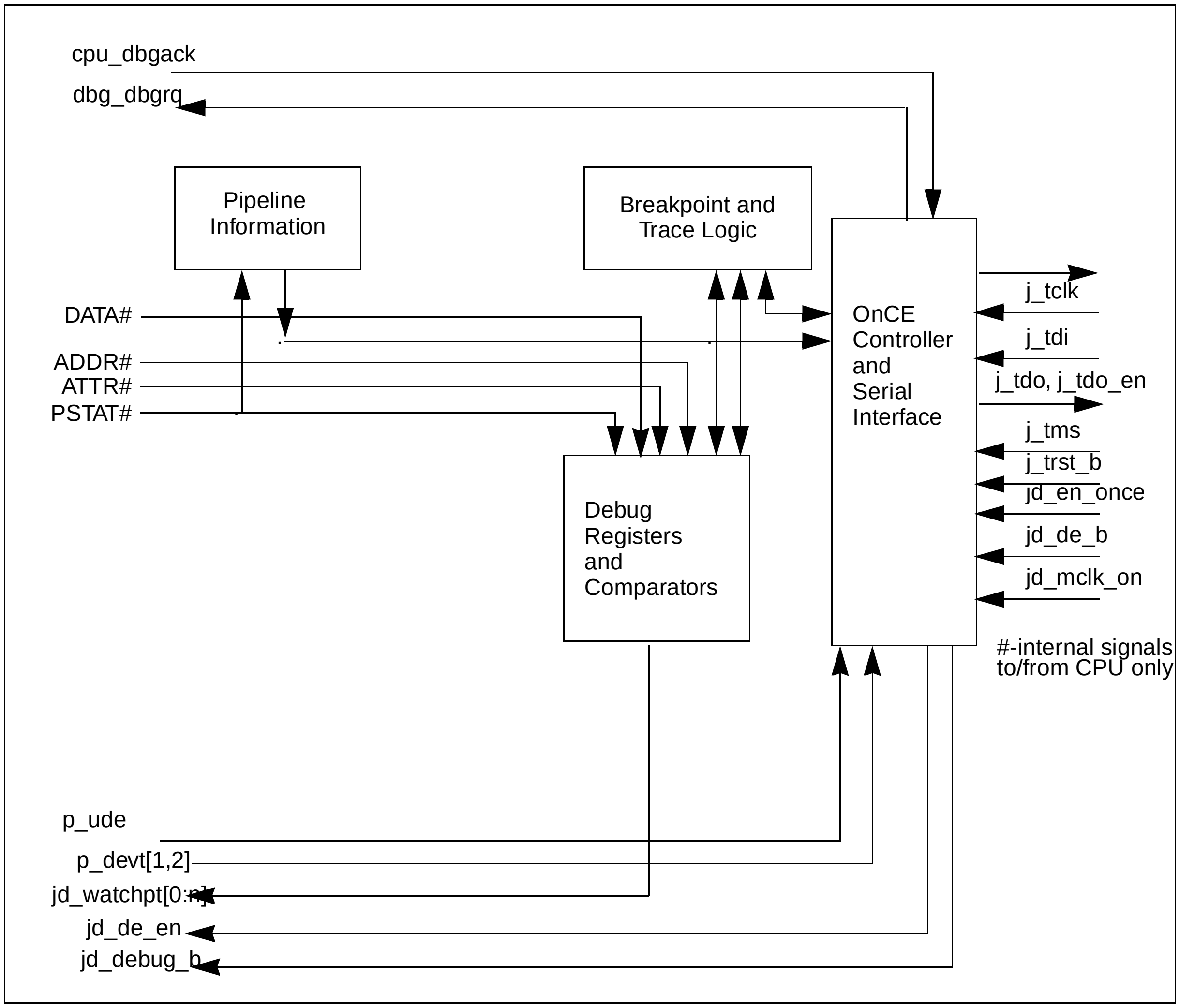


Figure 9-1. e200 Debug Resources

9.2 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 be cleared). 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 (DBCR0–3). The Unconditional Debug Event (UDE) is an exception to this rule; it is always enabled. Once a Debug Status Register (DBSR) bit is set (other than MRR and CNT1TRG), if Debug interrupts are enabled by MSR[DE], a Debug interrupt is generated. The debug interrupt handler is responsible for ensuring that multiple repeated debug interrupts do not occur by clearing the DBSR as appropriate.

Certain debug events are not allowed to occur when MSR[DE]=0 and DBCR0[EDM]=0. In such situations, no debug exception occurs and thus no DBSR bit is set. Other debug events may cause debug exceptions and set DBSR bits regardless of the state of MSR[DE]. A Debug interrupt is delayed until MSR[DE] is later set to ‘1’.

When a Debug Status Register bit is set while MSR[DE]=0 and DBCR0[EDM]=0, an Imprecise Debug Event flag (DBSR[IDE]) also is set to indicate that an exception bit in the Debug Status Register was set while Debug interrupts were disabled. Debug interrupt handler software can use this bit to determine whether the address recorded in Debug Save/Restore Register 0 is an address associated with the instruction causing the debug exception, or the address of the instruction which enabled a delayed Debug interrupt by setting the MSR[DE] bit. A **mtmsr** or **mtdbcr0** which causes both MSR[DE] and DBCR0[IDM] to become set, enabling precise debug mode, may cause an Imprecise (Delayed) Debug exception to be generated due to an earlier recorded event in the Debug Status register.

There are eight types of debug events defined by Power Architecture Book E, as follows:

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

In addition, e200 defines additional debug events:

- The Debug Counter debug events DCNT1 and DCNT2 which are described in Section 9.2.11, “Debug Counter Debug Event.”
- The External debug events DEVT1 and DEVT2 which are described in Section 9.2.12, “External Debug Event.”
- The Critical Interrupt Taken debug event CIRPT which is described in Section 9.2.8, “Critical Interrupt Taken Debug Event.”
- The Critical Return debug event CRET which is described in Section 9.2.10, “Critical Return Debug Event.”

The e200 debug configuration supports most of these event types. Unsupported Power Architecture Book E defined functionality is as follows:

- Instruction Address Compare and Data Address Compare real address mode is not supported.
- Data Value Compare 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.

9.2.1 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, IAC1, IAC2, IAC3, and IAC4 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. This event can occur and be recorded in DBSR regardless of the setting of MSR[DE]. IAC events do not occur when an instruction would not have normally begun execution due to a higher priority exception at an instruction boundary.

IAC compares perform a 31-bit compare for VLE instruction pages, and 30-bit compares for Book E instruction pages. Each halfword fetched by the instruction fetch unit is marked with a set of bits indicating whether an Instruction Address Compare occurred on that halfword. Debug exceptions 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 which miss in the TLB, Book E pages are assumed, and a 30-bit compare is performed.

9.2.2 Data Address Compare Event

Data Address Compare debug events occur when enabled and execution of a load or store class instruction results in a data access that meets the criteria specified in the DBCR0, DBCR2, DAC1, and DAC2 Registers. Data address compares may specify user/supervisor mode and data space (MSR[DS]), along with an effective address, masked effective address, or range of effective addresses for comparison. This event can occur and be recorded in DBSR regardless of the setting of MSR[DE]. Two address compare values (DAC1, DAC2) are provided.

NOTE

In contrast to the Power Architecture Book E definition, Data Address Compare events on e200 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.

If a load or store class instruction does not complete successfully due to a Data TLB or Data Storage exception, and a Data Address Compare debug exception also occurs, the result is an imprecise Debug interrupt, the address value saved in DSRR0 (or CSRR0 if the Debug APU is disabled) is the address of the load or store class instruction, and the DBSR[IDE] bit is set. In addition to occurring when DBCR0[IDM]=1, this circumstance can also occur when DBCR0[EDM]=1.

NOTE

DAC events are not recorded or counted if a **lmw** or **stmw** instruction is interrupted prior to completion by a critical input or external input interrupt.

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.

9.2.3 Linked Instruction Address and Data Address Compare Event

Data Address Compare debug events may be 'linked' with an Instruction Address Compare event by setting the DAC1LNK and/or DAC2LNK control bits in DBCR2 to further refine when a Data Address Compare debug event is generated. DAC1 may be linked with IAC1, and DAC2 (when not used as a mask or range bounds register) may be linked with IAC3. When linked, a DAC1 (or DAC2) debug event occurs when the same instruction which generates the DAC1 (or DAC2) 'hit' also generates an IAC1 (or IAC3) 'hit'. When linked, the IAC1 (or IAC3) event is not recorded in the Debug Status register, regardless of whether a corresponding DAC1 (or DAC2) event occurs, or whether the IAC1 (or IAC3) event enable is set.

When enabled and execution of a load or store class instruction results in a data access with an address that meets the criteria specified in the DBCR0, DBCR2, DAC1, and DAC2 Registers, and the instruction also meets the criteria for generating an Instruction Address Compare event, a Linked Data Address Compare debug event occurs. This event can occur and be recorded in DBSR regardless of the setting of MSR[DE]. The normal DAC1 and DAC2 status bits in the DBSR are used for recording these events. The IAC1 and IAC3 status bits are not set if the corresponding Instruction Address Compare register is linked.

Linking is enabled using control bits in DBCR2. If Data Address Compare debug events are used to control or modify operation of the Debug Counter, linking is also available, even though DBCR0 may not have enabled IAC or DAC events. Also, Instruction Address Compare events which are linked may still affect the Debug Counter (if enabled to), thus may be used to either trigger a counter, or be counted, in contrast to being blocked from affecting the DBSR.

NOTE

Linked DAC events are not recorded or counted if a load multiple word or store multiple word instruction is interrupted prior to completion by a critical input or external input interrupt.

9.2.4 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.

9.2.5 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, which is taken (either an unconditional branch, or a conditional branch whose branch condition is true), and MSR[DE]=1 or DBCR0[EDM]=1. Branch Taken debug events are not recognized if MSR[DE]=0 and DBCR0[EDM]=0 at the time of execution of the branch instruction and thus DBSR[IDE] can not be set by a Branch Taken debug event. When a Branch Taken debug event is recognized, the DBSR[BRT] bit is set to 1 to record the debug exception, and the address of the branch instruction is recorded in DSRR0.

9.2.6 Instruction Complete Debug Event

An Instruction Complete debug event (ICMP) occurs if Instruction Complete debug events are enabled (DBCR0[ICMP]=1), execution of any instruction is completed, and MSR[DE]=1 or DBCR0[EDM]=1. If execution of an instruction is suppressed due to the instruction causing some other exception which is enabled to generate an interrupt, then the attempted execution of that instruction does not cause an Instruction Complete debug event. The ***sc*** instruction does not fall into the category of an instruction whose execution is suppressed, because the instruction actually executes and then generates a System Call interrupt. In this case, the Instruction Complete debug exception is also set. When an Instruction Complete debug event is recognized, DBSR[ICMP] is set to 1 to record the debug exception and the address of the next instruction to be executed is recorded in DSRR0.

Instruction Complete debug events are not recognized if MSR[DE]=0 and DBCR0[EDM]=0 at the time of execution of the instruction, thus DBSR[IDE] is not generally set by an ICMP debug event.

NOTE

Instruction complete debug events are not generated by the execution of an instruction which sets MSR[DE] to ‘1’ while DBCR0[ICMP]=1, nor by the execution of an instruction which sets DBCR0[ICMP] to ‘1’ while MSR[DE]=1 or DBCR0[EDM]=1.

9.2.7 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 is the address of the non-critical interrupt handler.

9.2.8 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 is 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.

9.2.9 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** (such as before the MSR is updated by the **rfi** or **se_rfi**), then DBSR[IDE] is also set to 1 to record the imprecise debug event.

If MSR[DE]=1 at the time of the execution of the **rfi** or **se_rfi**, a Debug interrupt occurs provided there exists no higher priority exception which is enabled to cause an interrupt. Debug Save/Restore Register 0 is set to the address of the **rfi** or **se_rfi** instruction.

9.2.10 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** (such as 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 occurs provided there exists no higher priority exception which is enabled to cause an interrupt. Debug Save/Restore Register 0 is set to the address of the **rfci** or **se_rfci** instruction. Note that this debug event should not normally be enabled unless the Debug APU is also enabled to avoid corruption of CSRR0/1.

9.2.11 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.

9.2.12 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.

9.2.13 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 which 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.

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

9.3.1 Debug Address and Value Registers

Instruction Address Compare registers IAC1, IAC2, IAC3, and IAC4 are used to hold instruction addresses for address comparison purposes. In addition, IAC2 and IAC4 hold mask information for IAC1 and IAC3 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 Book E 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.

9.3.2 Debug Counter Register (DBCNT)

The Debug Counter Register (DBCNT) contains two 16-bit counters (CNT1 and CNT2) which can be configured to operate independently, or 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 9-2.

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 9-2. DBCNT Register

Refer to Section 9.3.3.4, “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.

9.3.3 Debug Control and Status Registers

Debug Control Registers 0-3 (DBCR0, DBCR1, DBCR2, and DBCR3) are used to enable debug events, reset the processor, control timer operation during debug events, and set the debug mode of the processor. The Debug Status register (DBSR) records debug exceptions while Internal or External Debug Mode is enabled.

e200 requires that a context synchronizing instruction follow a **mtspr** DBCR0-3 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, DBCR2, and DBCR3 are used, even though the corresponding event(s) may be disabled (via DBCR0) from setting DBSR flags.

9.3.3.1 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 flags. e200 adds some implementation specific bits to this register, as seen in Figure 9-3.

EDM	IDM	RST	ICMP	BRT	IRPT	TRAP	IAC1	IAC2	IAC3	IAC4	DAC1	DAC2	RET	0	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 9-3. DBCR0 Register

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

Table 9-1 provides bit definitions for Debug Control Register 0.

Table 9-1. DBCR0 Bit Definitions

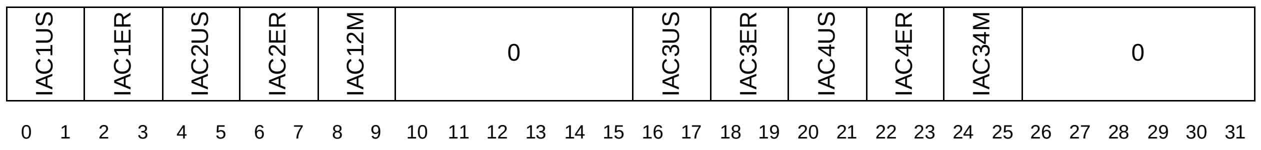
Bit(s)	Name	Description
0	EDM	External Debug Mode. This bit is read-only by software. 0 External debug mode disabled. Internal debug events not mapped into external debug events. 1 External debug mode enabled. Events do not cause the CPU to vector to interrupt code. Software is not permitted to write to debug registers {DBCRx, DBSR, DBCNT, IAC1-4, DAC1-2}.
		Programming Notes: It is recommended that debug status bits in the Debug Status Register 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 unless EDM is set. 1 Debug exceptions are enabled. Enabled debug events update the DBSR. If MSR[DE]=1, the occurrence of a debug event, or the recording of an earlier debug event in the Debug Status Register when MSR[DE] was cleared, causes a Debug interrupt.
2:3	RST	Reset Control 00 No function 01 Reserved 10 *p_resetout_b* pin asserted by Debug Reset Control. Allows external device to initiate processor 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

Table 9-1. DBCR0 Bit Definitions (continued)

Bit(s)	Name	Description
11	IAC4	Instruction Address Compare 4 Debug Event Enable 0 IAC4 debug events are disabled 1 IAC4 debug events are enabled
12:13	DAC1	Data Address Compare 1 Debug Event Enable 00 DAC1 debug events are disabled 01 DAC1 debug events are enabled only for store-type data storage accesses 10 DAC1 debug events are enabled only for load-type data storage accesses 11 DAC1 debug events are enabled for load-type or store-type data storage accesses
14:15	DAC2	Data Address Compare 2 Debug Event Enable 00 DAC2 debug events are disabled 01 DAC2 debug events are enabled only for store-type data storage accesses 10 DAC2 debug events are enabled only for load-type data storage accesses 11 DAC2 debug events are enabled for load-type or store-type data storage accesses
16	RET	Return Debug Event Enable 0 RET debug events are disabled 1 RET debug events are enabled
17:20	—	Reserved
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 bits 1 Disable clocking of TimeBase timers if any DBSR bit is set (except MRR or CNT1TRG)

9.3.3.2 Debug Control Register 1 (DBCR1)

Debug Control Register 1 is used to configure Instruction Address Compare operation. The DBCR1 register is shown in Figure 9-4.



SPR—309; Read/Write; Reset—0x0

Figure 9-4. DBCR1 Register

Table 9-2 provides bit definitions for Debug Control Register 1.

Table 9-2. DBCR1 Bit Definitions

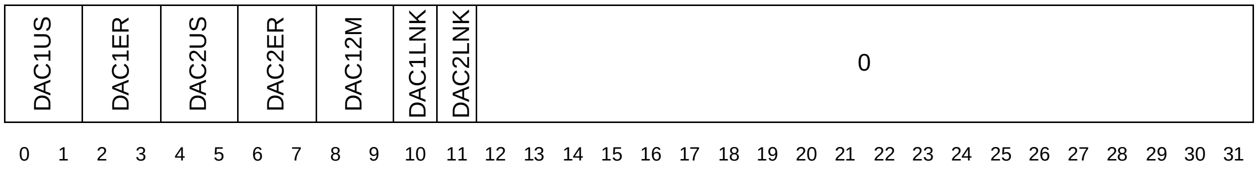
Bit(s)	Name	Description
0:1	IAC1US	Instruction Address Compare 1 User/Supervisor Mode 00 IAC1 debug events not affected by MSR[PR] 01 Reserved 10 IAC1 debug events can only occur if MSR[PR]=0 (Supervisor mode) 11 IAC1 debug events can only occur if MSR[PR]=1. (User mode)
2:3	IAC1ER	Instruction Address Compare 1 Effective/Real Mode 00 IAC1 debug events are based on effective address 01 Unimplemented in e200 (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 e200 (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.

Table 9-2. DBCR1 Bit Definitions (continued)

Bit(s)	Name	Description
10:15	—	Reserved
16:17	IAC3US	Instruction Address Compare 3 User/Supervisor Mode 00 IAC3 debug events not affected by MSR[PR] 01 Reserved 10 IAC3 debug events can only occur if MSR[PR]=0 (Supervisor mode) 11 IAC3 debug events can only occur if MSR[PR]=1 (User mode)
18:19	IAC3ER	Instruction Address Compare 3 Effective/Real Mode 00 IAC3 debug events are based on effective address 01 Unimplemented in e200 (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 4Effective/Real Mode 00 IAC4 debug events are based on effective address 01 Unimplemented in e200 (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

9.3.3.3 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 9-5.



SPR—310; Read/Write; Reset—0x0

Figure 9-5. DBCR2 Register

Table 9-3 provides bit definitions for Debug Control Register 2.

Table 9-3. DBCR2 Bit Definitions

Bit(s)	Name	Description
0:1	DAC1US	Data Address Compare 1 User/Supervisor Mode 00 DAC1 debug events not affected by MSR[PR] 01 Reserved 10 DAC1 debug events can only occur if MSR[PR]=0 (Supervisor mode) 11 DAC1 debug events can only occur if MSR[PR]=1. (User mode)
2:3	DAC1ER	Data Address Compare 1 Effective/Real Mode 00 DAC1 debug events are based on effective address 01 Unimplemented in e200 (Book E real address compare), no match can occur 10 DAC1 debug events are based on effective address and can only occur if MSR[DS]=0 11 DAC1 debug events are based on effective address and can only occur if MSR[DS]=1
4:5	DAC2US	Data Address Compare 2 User/Supervisor Mode. 00 DAC2 debug events not affected by MSR[PR] 01 Reserved 10 DAC2 debug events can only occur if MSR[PR]=0 (Supervisor mode) 11 DAC2 debug events can only occur if MSR[PR]=1. (User mode)
6:7	DAC2ER	Data Address Compare 2 Effective/Real Mode 00 DAC2 debug events are based on effective address 01 Unimplemented in e200 (Book E real address compare), no match can occur 10 DAC2 debug events are based on effective address and can only occur if MSR[DS]=0 11 DAC2 debug events are based on effective address and can only occur if MSR[DS]=1

Table 9-3. DBCR2 Bit Definitions (continued)

Bit(s)	Name	Description
8:9	DAC12M	Data Address Compare 1/2 Mode 00 Exact address compare. DAC1 debug events can only occur if the address of the data access is equal to the value specified in DAC1. DAC2 debug events can only occur if the address of the data access is equal to the value specified in DAC2. 01 Address bit match. DAC1 debug events can occur only if the address of the data access ANDed with the contents of DAC2, are equal to the contents of DAC1 also ANDed with the contents of DAC2. DAC2 debug events do not occur. DAC1US and DAC1ER settings are used. 10 Inclusive address range compare. DAC1 debug events can occur only if the address of the data access is greater than or equal to the value specified in DAC1 and less than the value specified in DAC2. DAC2 debug events do not occur. DAC1US and DAC1ER settings are used. 11 Exclusive address range compare. DAC1 debug events can occur only if the address of the data access is less than the value specified in DAC1 or is greater than or equal to the value specified in DAC2. DAC2 debug events do not occur. DAC1US and DAC1ER settings are used.
10	DAC1LNK	Data Address Compare 1 Linked 0 no affect 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 affect 1 DAC 2 debug events are linked to IAC3 debug events. IAC3 debug events do not affect DBSR When linked to IAC3, DAC2 debug events are conditioned based on whether the instruction also generated an IAC3 debug event. DAC2 can only be linked if DAC12M specifies *Exact Address Compare* because DAC2 debug events are not generated in the other compare modes.
12:31	—	Reserved for Data Value Compare control (not supported by e200)

9.3.3.4 Debug Control Register 3 (DBCR3)

Debug Control Register 3 is used to enable and configure the Debug Counter and debug counter events. For counter operation, the specific debug events which cause counters to decrement are specified in DBCR3. Note that the corresponding events do not need to be 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 setting of bits in DBSR by corresponding events may be disabled via DBCR0. Multiple count-enabled events which 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 9.4.1, “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 the triggering events occurs, and is then enabled to begin operation. Triggering status for Counter 1 is provided in the Debug Status Register. Triggering mode is enabled by 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] 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)

Due to pipelining and detection of IAC events early in the pipeline and DAC events late in the pipeline, no guarantee is made on the exact instruction boundary that a debug exception will be generated when IAC and DAC events are combined for counting. This also applies to the case where Counter 1 is being triggered by Counter 2, and a combination of IAC and DAC events are being enabled for the counters, even if only one of these types is enabled for a particular counter. In general, when an IAC event logically follows closely behind a DAC event (within several instructions), it cannot be recognized immediately since the DAC event has not necessarily been generated in the pipeline at the time the IAC is seen, and thus the counter may not decrement to zero for the IAC event until after the instruction with the IAC (and perhaps several additional instructions) has proceeded down the execution pipeline. The instruction boundary where the debug exception is actually generated in this case will typically follow the IAC by up to several instructions.

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

If Counter 2 is being used to trigger Counter 1, Counter 2 events should not normally be enabled in DBCR0, and will not be blocked.

NOTE

Multiple IAC or DAC events will not be counted during a **lmw** or **stmw** instruction, and no count will occur if either is interrupted by a critical input or external input interrupt prior to completion.

DBCR3 is a e200 implementation specific register and is shown in Figure 9-6.

DEVT1C1	DEVT2C1	ICMPC1	IAC1C1	IAC2C1	IAC3C1	IAC4C1	DAC1RC1	DAC1WC1	DAC2RC1	DAC2WC1	IRPTC1	RETC1	DEVT1C2	DEVT2C2	ICMPC2
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

IAC1C2	IAC2C2	IAC3C2	IAC4C2	DAC1RC2	DAC1WC2	DAC2RC2	DAC2WC2	DEVT1T1	DEVT2T1	IAC1T1	IAC3T1	DAC1RT1	DAC1WT1	CNT2T1	CONFIG
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

SPR—561; Read/Write; Reset—0x0

Figure 9-6. DBCR3 Register

Table 9-4 provides bit definitions for Debug Control Register 3.

Table 9-4. DBCR3 Bit Definitions

Bit(s)	Name	Description
0	DEVT1C1	External Debug Event 1 Count 1 Enable 0 Counting DEVT1 debug events by Counter 1 is disabled 1 Counting DEVT1 debug events by Counter 1 is enabled
1	DEVT2C1	External Debug Event 2 Count 1 Enable 0 Counting DEVT2 debug events by Counter 1 is disabled 1 Counting DEVT2 debug events by Counter 1 is enabled
2	ICMPC1	Instruction Complete Debug Event Count 1 Enable 0 Counting ICMP debug events by Counter 1 is disabled 1 Counting ICMP debug events by Counter 1 is enabled (Note that ICMP events are masked by MSR[DE]=0 when operating in Internal Debug Mode)
3	IAC1C1	Instruction Address Compare 1 Debug Event Count 1 Enable 0 Counting IAC1 debug events by Counter 1 is disabled 1 Counting IAC1 debug events by Counter 1 is enabled
4	IAC2C1	Instruction Address Compare2 Debug Event Count 1 Enable 0 Counting IAC2 debug events by Counter 1 is disabled 1 Counting IAC2 debug events by Counter 1 is enabled
5	IAC3C1	Instruction Address Compare 3 Debug Event Count 1 Enable 0 Counting IAC3 debug events by Counter 1 is disabled 1 Counting IAC3 debug events by Counter 1 is enabled
6	IAC4C1	Instruction Address Compare 4 Debug Event Count 1 Enable 0 Counting IAC4 debug events by Counter 1 is disabled 1 Counting IAC4 debug events by Counter 1 is enabled
7	DAC1RC1	Data Address Compare 1 Read Debug Event Count 1 Enable[1] 0 Counting DAC1R debug events by Counter 1 is disabled 1 Counting DAC1R debug events by Counter 1 is enabled
8	DAC1WC1	Data Address Compare 1 Write Debug Event Count 1 Enable[1] 0 Counting DAC1W debug events by Counter 1 is disabled 1 Counting DAC1W debug events by Counter 1 is enabled
9	DAC2RC1	Data Address Compare 2 Read Debug Event Count 1 Enable[1] 0 Counting DAC2R debug events by Counter 1 is disabled 1 Counting DAC2R debug events by Counter 1 is enabled

Table 9-4. DBCR3 Bit Definitions (continued)

Bit(s)	Name	Description
10	DAC2WC1	Data Address Compare 2 Write Debug Event Count 1 Enable[1] 0 Counting DAC2W debug events by Counter 1 is disabled 1 Counting DAC2W debug events by Counter 1 is enabled
11	IRPTC1	Interrupt Taken Debug Event Count 1 Enable 0 Counting IRPT debug events by Counter 1 is disabled 1 Counting IRPT debug events by Counter 1 is enabled
12	RETC1	Return Debug Event Count 1 Enable 0 Counting RET debug events by Counter 1 is disabled 1 Counting RET debug events by Counter 1 is enabled
13	DEVT1C2	External Debug Event 1 Count 2 Enable 0 Counting DEVT1 debug events by Counter 2 is disabled 1 Counting DEVT1 debug events by Counter 2 is enabled
14	DEVT2C2	External Debug Event 2 Count 2 Enable 0 Counting DEVT2 debug events by Counter 2 is disabled 1 Counting DEVT2 debug events by Counter 2 is enabled
15	ICMPC2	Instruction Complete Debug Event Count 2 Enable 0 Counting ICMP debug events by Counter 2 is disabled 1 Counting ICMP debug events by Counter 2 is enabled (Note that ICMP events are masked by MSR[DE]=0 when operating in Internal Debug Mode)
16	IAC1C2	Instruction Address Compare 1 Debug Event Count 2Enable 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

Table 9-4. DBCR3 Bit Definitions (continued)

Bit(s)	Name	Description
24	DEVT1T1	External Debug Event 1 Trigger Counter 1 Enable 0 No effect 1 A DEVT1 debug event will trigger Counter 1 operation
25	DEVT2T1	External Debug Event 2 Trigger Counter 1 Enable 0 No effect 1 A DEVT2 debug event will trigger Counter 1 operation
26	IAC1T1	Instruction Address Compare 1 Trigger Counter 1 Enable 0 No effect 1 An IAC1 debug event will trigger Counter 1 operation
27	IAC3T1	Instruction Address Compare 3 Trigger Counter 1 Enable 0 No effect 1 An IAC3 debug event will trigger Counter 1 operation
28	DAC1RT1	Data Address Compare 1 Read Trigger Counter 1 Enable 0 No effect 1 A DAC1R debug event will trigger Counter 1 operation
29	DAC1WT1	Data Address Compare 1 Write Trigger Counter 1 Enable 0 No effect 1 A DAC1W debug event will trigger Counter 1 operation
30	CNT2T1	Debug Counter 2 Trigger Counter 1 Enable 0 No effect 1 Counter 2 decrementing to a value of ‘0’ will trigger Counter 1 operation
31	CONFIG	Debug Counter Configuration 0 Counter 1 and Counter 2 are independent counters 1 Counter 1 and Counter 2 are concatenated into a single 32-bit counter. The event count control bits for Counter 1 are used and the event count control bits for Counter 2 are ignored.

[1] If the DACx field in DBCR0 is set to restrict events to only reads or only writes, only those events will be counted if enabled in DBCR3. In general, DAC events should be disabled in DBCR0.

NOTE

Updates to the DBCR0, DBSR, DBCR3, and DBCNT registers should be performed carefully if the counters are currently enabled for counting ICMP events. For these cases, it is possible that the instruction which 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 which 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 which modifies a counter, regardless of whether a debug event is generated based on the old counter value. To avoid this, it is recommended that the DBCNT and DBCR3 values be modified only when no possibility of a counter related debug event on the **mtspr** instruction is possible. Modifying DBCR0 to affect counter event enabling/disabling may have similar issues, as may modifying the DBSR[CNT1TRG] bit.

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 which 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 which 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 which 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.

9.3.3.5 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 or External Debug Mode is enabled. When debug interrupts are enabled (MSR[DE]=1, DBCR0[IDM]=1, and DBCR0[EDM]=0), a set bit in DBSR other than MRR or VLES causes a debug interrupt to be generated. The debug interrupt handler is responsible for clearing DBSR bits prior to returning to normal execution. The Power Architecture VLE APU adds the DBSR[VLES] status bit to indicate debug events occurring due to a Power Architecture VLE instruction. The DBSR register is shown in Figure 9-7.

IDE	UDE	MRR		ICMP	BRT	IRPT	TRAP	IAC1	IAC2	IAC3	IAC4	DAC1R	DAC1W	DAC2R	DAC2W	RET	0				DEVT1	DEVT2	DCNT1	DCNT2	CIRPT	CRET	VLES	0			CNT1TRG
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

SPR—304; Read/Write; Reset—0x1000_0000

Figure 9-7. DBSR Register

Table 9-5 provides bit definitions for the Debug Status Register.

Table 9-5. DBSR Bit Definitions

Bit(s)	Name	Description
0	IDE	Imprecise Debug Event Set to 1 if MSR[DE]=0 and DBCR0[EDM]=0 and a debug event causes its respective Debug Status Register bit to be set to 1. It may also be set to ‘1’ if DBCR0[EDM]=1 and an imprecise debug event occurs due to a DAC event on a load or store which is terminated with error.
1	UDE	Unconditional Debug Event Set to 1 if an Unconditional debug event occurred.
2:3	MRR	Most Recent Reset. 00 No reset occurred because these bits were last cleared by software 01 A hard reset occurred because 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	Instruction Address Compare 4 Debug Event Set to 1 if an IAC4 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

Table 9-5. DBSR Bit Definitions (continued)

Bit(s)	Name	Description
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
21	DEVT1	External Debug Event 1 Debug Event Set to 1 if a DEVT1 debug event occurred
22	DEVT2	External Debug Event 2 Debug Event Set to 1 if a DEVT2 debug event occurred
23	DCNT1	Debug Counter 1 Debug Event Set to 1 if a DCNT1 debug event occurred
24	DCNT2	Debug Counter 2 Debug Event Set to 1 if a DCNT2 debug event occurred
25	CIRPT	Critical Interrupt Taken Debug Event Set to 1 if a Critical Interrupt Taken debug event occurred.
26	CRET	Critical Return Debug Event Set to 1 if a Critical Return debug event occurred
27	VLES	VLE Status Set to 1 if an ICMP, BRT, TRAP, RET, CRET, IAC, or DAC debug event occurred on a Power Architecture VLE Instruction. Undefined for IRPT, CIRPT, DEVT[1,2], DCNT[1,2], and UDE events
28:30	—	Reserved
31	CNT1TRG	Counter 1 Triggered Set to 1 if Debug Counter 1 is triggered by a trigger event.

9.4 External Debug Support

External debug support is supplied through the e200 OnCE controller serial interface which allows access to internal core registers and other system state while in External Debug Mode (EDM). All debug resources including DBCR0-3, DBSR, IAC1-4, DAC1-2, and DBCNT are accessible through the serial OnCE interface in external debug mode. Setting the DBCR0[EDM]bit to ‘1’ through the OnCE interface enables external debug mode and disables software updates to the debug registers. When DBCR0[EDM] is set, debug events enabled to set respective DBSR status bits also cause the CPU to enter Debug Mode as opposed to generating Debug Interrupts. In Debug Mode, the CPU is halted at a recoverable boundary, and an external Debug Control Module may control CPU operation through the On-Chip Emulation logic (OnCE). No Debug interrupts can occur while DBCR0[EDM] remains set.

NOTE

On the initial setting of DBCR0[EDM] to ‘1’, other bits in DBCR0 remain unchanged. After DBCR0[EDM] has been set, all debug register resources may be subsequently controlled through the OnCE interface. The DBSR register should be cleared as part of the process of enabling external debug activity. The core 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 which could cause unintended operation.

NOTE

It is intended for the core 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 is 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 is 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), 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.

9.4.1 OnCE Introduction

The e200 on-chip emulation circuitry (OnCE™/Nexus Class 1 interface) provides a means of interacting with the e200 core and integrated system so that a user may examine registers, memory, or on-chip peripherals facilitating hardware/software development. OnCE operation is controlled via an industry standard IEEE 1149.1 TAP controller. By using public instructions, the external hardware debugger can freeze or halt the CPU, read and write internal state, and resume normal execution. The core does not contain IEEE 1149.1 standard boundary cells on its interface, as it is a building block for further integration. It does not support the JTAG related boundary scan instruction functionality, although JTAG public instructions may be decoded and signaled to external logic.

The OnCE logic provides for Nexus Class 1 static debug capability (utilizing the same set of resources available to software while in internal debug mode), and is present in all e200-based designs.

The e200 core is designed to be a fully integrateable 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 e200 module can be easily integrated with existing JTAG designs or as a stand-alone controller.

In order to enable full OnCE operation, the *jd_enable_once* input signal must be asserted. In some system integrations, this is automatic, because the input is tied asserted. Other integrations may require the execution of the Enable OnCE command via the TAP and appropriate entry of serial data. Exact requirements are 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.

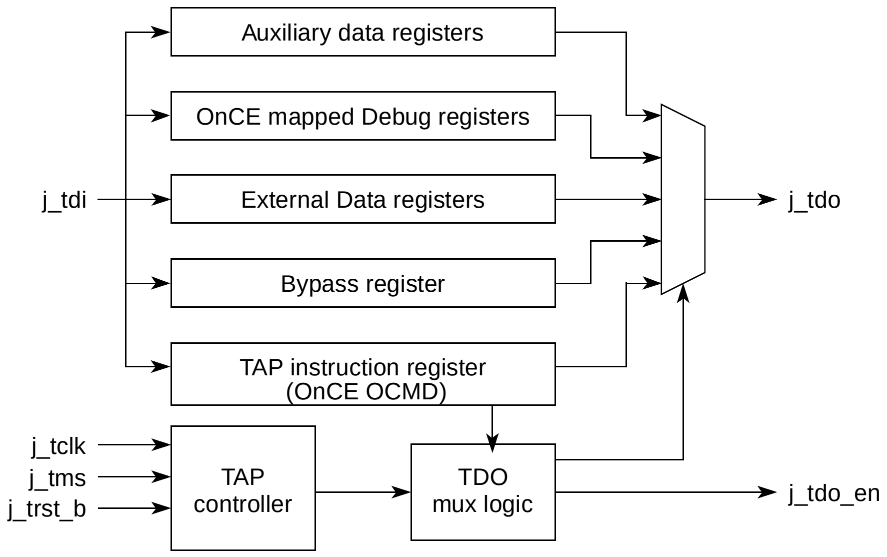
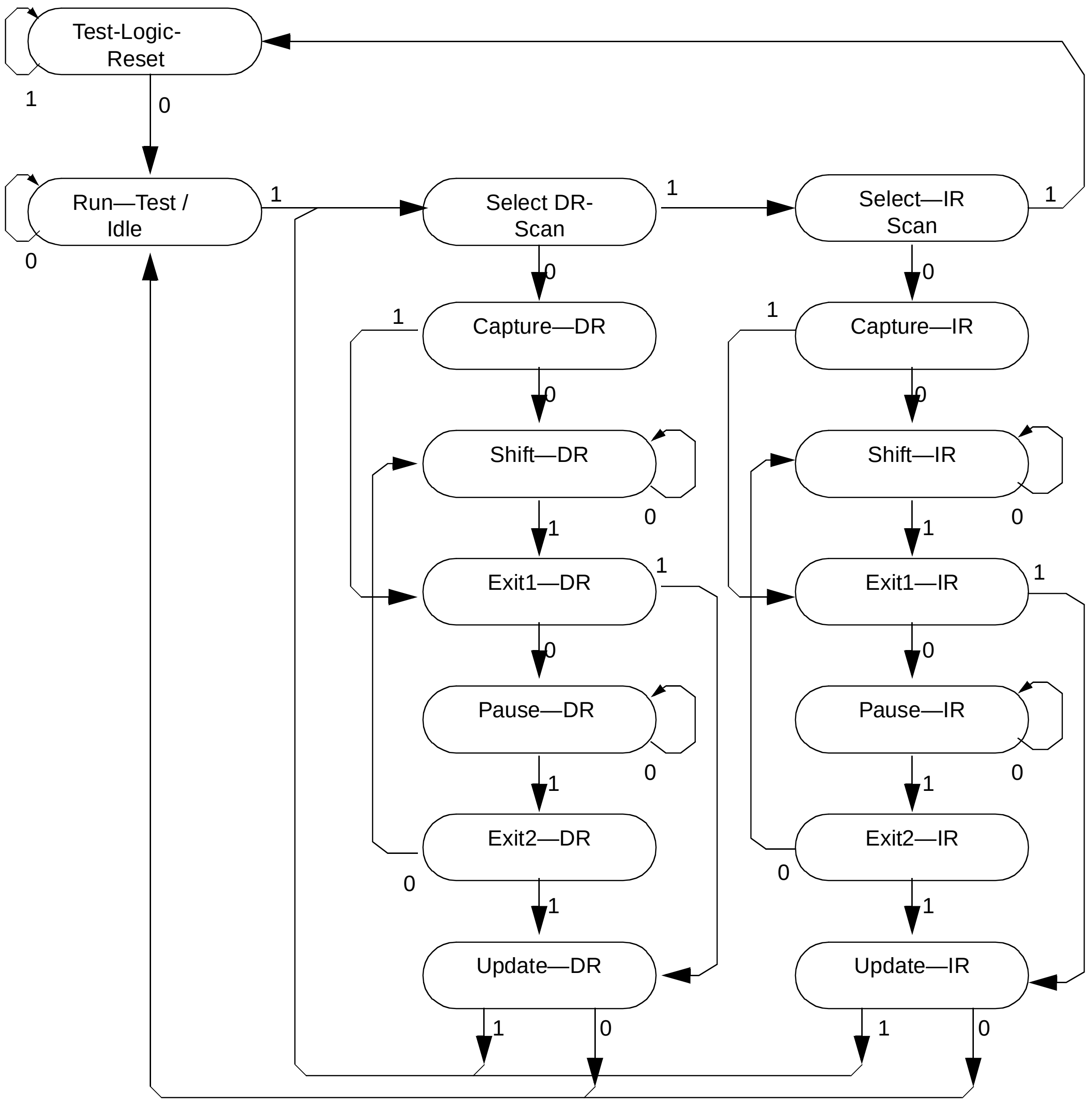


Figure 9-8. OnCE TAP Controller and Registers

The OnCE controller is implemented as a 16-state FSM, with a one-to-one correspondence to the states defined for the JTAG TAP controller.



Access to e200 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 e200 CPU Scan Chain (CPUSCR); execution of scanned instructions by the e200 is used as the method to access required data. Memory locations may be read by scanning a load instruction into the e200 core, which references the desired memory location, executing the load instruction, and then scanning out the result of the load. Other resources are accessed in a similar manner.

The initial entry by the CPU into the debug state (or mode) from normal, stopped, halted, or checkstop states (all indicated via the OnCE Status Register (OSR), Section 9.4.5.1, “e200 OnCE Status Register”) by assertion of one or more debug requests, begins a *debug session*. The *jd_debug_b* output signal indicates that a debug session is in progress, and the OSR indicates the CPU is in the debug state.

Instructions may the be single-stepped by scanning new values into the CPUSCR, and performing a OnCE go+noexit command (See Section 9.4.5.2, "e200 OnCE Command Register (OCMD)"). The CPU then temporarily exits the debug state (but not the debug session) to execute the instruction, and then returns to the debug state (again indicated via the OnCE Status Register (OSR)). The debug session remains in force until the final OnCE go+exit command is executed, at which time the CPU returns to 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.

9.4.2 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* causes the chip to exit the low power mode enabled by the setting of MSR[WE], as well as temporarily exiting the waiting, stopped, or halted power management states.

Table 9-6 details the primary JTAG/OnCE interface signals.

Table 9-6. 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 7.3.15, "JTAG Support Signals."

9.4.3 OnCE Internal Interface Signals

The following paragraphs describe the e200 OnCE interface signals to other internal blocks associated with the e200 OnCE controller.

9.4.3.1 CPU Debug Request (*dbg_dbgrq*)

The *dbg_dbgrq* signal is asserted by the e200 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.

9.4.3.2 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 e200 OnCE control logic and the rest of the CPU. The CPU core may enter debug mode either through a software or hardware event.

9.4.4 OnCE Interface Signals

The following paragraphs describe additional e200 OnCE interface signals to other external blocks such as a Nexus Controller and external blocks which may need information pertaining to debug operation.

9.4.4.1 OnCE Enable (*jd_en_once*)

The OnCE enable signal *jd_en_once* is used to enable the OnCE controller to allow certain instructions and operations to be executed. Assertion of this signal enables the full OnCE command set, as well as operation of control signals and OnCE Control register functions. When this signal is disabled, only the Bypass, ID and Enable_OnCE commands are executed by the OnCE unit, and all other commands default to a “Bypass” command. The OnCE Status register (OSR) is not visible when OnCE operation is disabled. In addition, OnCE Control register (OCR) functions are disabled, as is the operation of the *jd_de_b* input. Secure systems may choose to leave the *jd_en_once* signal negated until a security check has been performed. Other systems should tie this signal asserted to enable full OnCE operation. The *j_en_once_regsel* output signal is provided to assist external logic performing security checks. Refer to Section 7.3.15.15, “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 takes effect 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.

9.4.4.2 OnCE Debug Request/Event (*jd_de_b, jd_de_en*)

If implemented at the SoC level, a system level bidirectional open drain debug event pin *DE_b* (not part of the e200 interface) provides a fast means of entering the Debug Mode of operation from an external command controller (when input) as well as a fast means of acknowledging the entering of the Debug Mode of operation to an external command controller (when output). The assertion of this pin by a command controller causes the CPU core to finish the current instruction being executed, save the instruction pipeline information, enter Debug Mode, and wait for commands to be entered. If *DE_b* was used to enter the Debug Mode then *DE_b* must be negated after the OnCE controller responds with an acknowledge and before sending the first OnCE command. The assertion of this pin by the CPU Core acknowledges that it has entered the Debug Mode and is waiting for commands to be entered.

To support operation of this system pin, the OnCE logic supplies the *jd_de_en* output and samples the *jd_de_b* input when OnCE is enabled (*jd_en_once* asserted). Assertion of *jd_de_b* causes the OnCE logic to place the CPU into Debug Mode. Once Debug Mode has been entered, the *jd_de_en* output is asserted for three *j_tclk* periods to signal an acknowledge. *jd_de_en* can be used to enable the open-drain pulldown of the system level *DE_b* pin.

For systems which do not implement a system level bidirectional open drain debug event pin *DE_b*, the *jd_de_en* and *jd_de_b* signals may still be used to handshake debug entry.

9.4.4.3 e200 OnCE Debug Output (*jd_debug_b*)

The e200 OnCE Debug output *jd_debug_b* is used to indicate to on-chip resources that a debug session is in progress. Peripherals and other units may use this signal to modify normal operation for the duration of a debug session, which may involve the CPU executing a sequence of instructions solely for the purpose of visibility/system control which are not part of the normal instruction stream the CPU would have executed had it not been placed in debug mode. This signal is asserted the first time the CPU enters the debug state, and remains asserted until the CPU is released by a write to the e200 OnCE Command Register with the GO and EX bits set, and a register specified as either “No Register Selected” or the CPUSCR. This signal remains asserted even though the CPU may enter and exit the debug state for each instruction executed under control of the e200 OnCE controller. See Section 9.4.5.2, “e200 OnCE Command Register (OCMD),” for more information on the function of the GO and EX bits. This signal is not normally used by the CPU.

9.4.4.4 e200 CPU Clock On Input (*jd_mclk_on)*

The e200 CPU Clock On input *jd_mclk_on* is used to indicate that the CPU’s *m_clk* input is active. This input signal is expected to be driven by system logic external to the e200 core, is synchronized to the *j_tclk* (scan clock) clock domain, and is presented as a status flag on the *j_tdo* output during the Shift_IR state. External firmware may use this signal to ensure proper scan sequences occur to access debug resources in the *m_clk* clock domain.

9.4.4.5 Watchpoint Events (*jd_watchpt*[0:7])

The *jd_watchpt*[0:7] signals may be asserted by the e200 OnCE control logic to signal that a watchpoint condition has occurred. Watchpoints do not cause the CPU to be affected. They are provided to allow external visibility only. Watchpoint events are conditioned by the settings in the DBCR0, DBCR1, and DBCR2 registers.

9.4.5 e200 OnCE Controller and Serial Interface

The e200 OnCE Controller contains the e200 OnCE command register, the e200 OnCE decoder, and the status/control register. Figure 9-9 is a block diagram of the e200 OnCE controller. In operation, the e200 OnCE Command register acts as the IR for the e200 TAP controller, and all other OnCE resources are treated as data registers (DR) by the TAP controller. The Command register is loaded 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.

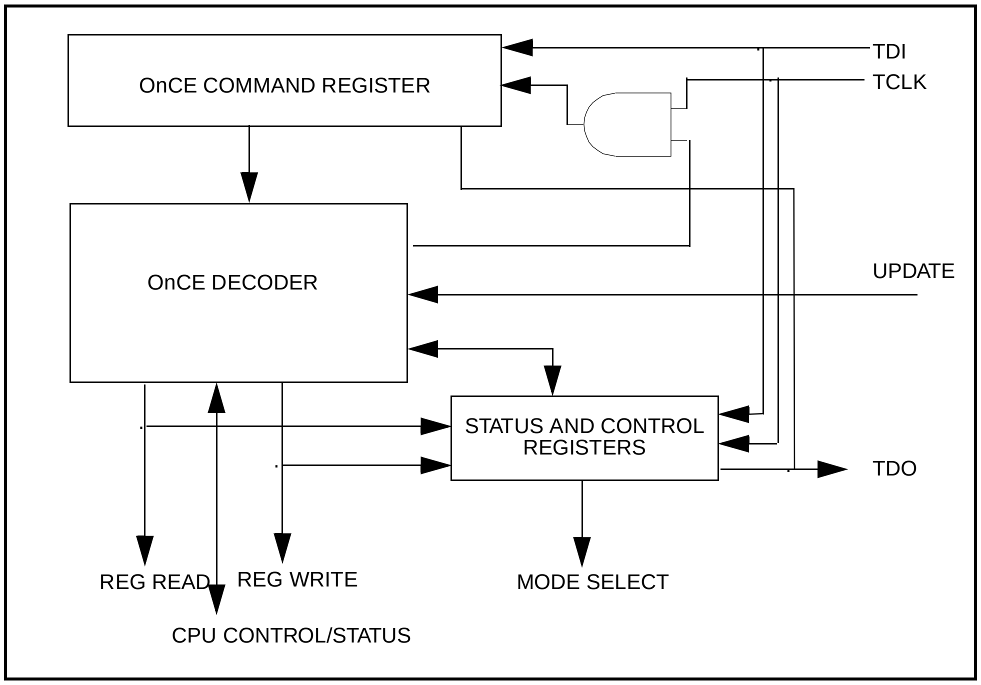


Figure 9-9. e200 OnCE Controller and Serial Interface

9.4.5.1 e200 OnCE Status Register

Status information regarding the state of the e200 CPU is latched into the OnCE Status register when the OnCE controller state machine enters the Capture-IR state. When OnCE operation is enabled, this information is provided on the *j_tdo* output in serial fashion when the Shift_IR state is entered following a Capture-IR. Information is shifted out least significant bit first.

MCLK	ERR	CHKSTOP	RESET	HALT	STOP	DEBUG	WAIT	0	1
0	1	2	3	4	5	6	7	8	9

Figure 9-10. OnCE Status Register

Table 9-7 provides bit definitions for the Once Status Register.

Table 9-7. OnCE Status Register Bit Definitions

Bit(s)	Name	Description
0	MCLK	MCLK *m_clk* Status Bit 0 Inactive state 1 Active state This status bit reflects the logic level on the *jd_mclk_on* input signal after capture by *j_tclk*.
1	ERR	ERROR This bit is used to indicate that an error condition occurred during attempted execution of the last single-stepped instruction (GO+NoExit with CPUSCR or No Register Selected in OCMD), and that the instruction may not have been properly executed. This could occur if an Interrupt (all classes including External, Critical, machine check, Storage, Alignment, Program, TLB, etc.) occurred while attempting to perform the instruction single step. In this case, the CPUSCR contains information related to the first instruction of the Interrupt handler, and no portion of the handler will have been executed.
2	CHKSTOP	CHECKSTOP Mode This bit reflects the logic level on the CPU *p_chkstop* output after capture by *j_tclk*.
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	—	Reserved, set to 0 for 1149.1 compliance
9	—	Reserved, set to 1 for 1149.1 compliance

9.4.5.2 e200 OnCE Command Register (OCMD)

The OnCE Command Register (OCMD) is a 10-bit shift register that receives its serial data from the TDI pin and serves as the instruction register (IR). It holds the 10-bit commands to be used as input for the e200 OnCE Decoder. The Command Register is shown in Figure 9-11. 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 9-11. OnCE Command Register

Table 9-8 provides bit definitions for the Once Command Register.

Table 9-8. OnCE Command Register Bit Definitions

Bit(s)	Name	Description
0	R/W	Read/Write Command Bit The R/W bit specifies the direction of data transfer. The table below describes the options defined by the R/W bit. 0 Write the data associated with the command into the register specified by RS[0:6] 1 Read the data contained in the register specified by RS[0:6] **Note:** The R/W bit generally ignored for read-only or write-only registers. 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 executes 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 leaves 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.
2	EX	Exit Command Bit 0 Remain in debug mode 1 Leave debug mode If the EX bit is set, the processor leaves 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 leaves the debug mode after the TAP controller Update-DR state is entered. Note that if the DR bit in the OnCE control register is set or remains set, or if a bit in the DBSR is set 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.
3:9	RS	Register Select The Register Select bits define which register is source (destination) for the read (write) operation. Table 9-11 indicates the e200 OnCE register addresses. Attempted writes to read-only registers are ignored.

Table 9-9 indicates the e200 OnCE register addresses.

Table 9-9. e200 OnCE Register Addressing

RS[0:6]	Register Selected
000 0000	Reserved
000 0001	Reserved
000 0010	JTAG ID (read-only)
000 0011–000 1111	Reserved
001 0000	CPU Scan Register (CPUSCR)
001 0001	No Register Selected (Bypass)
001 0010	OnCE Control Register (OCR)
001 0011	Reserved
001 0100–001 1111	Reserved
010 0000	Instruction Address Compare 1 (IAC1)
010 0001	Instruction Address Compare 2 (IAC2)
010 0010	Instruction Address Compare 3 (IAC3)
010 0011	Instruction Address Compare 4 (IAC4)
010 0100	Data Address Compare 1 (DAC1)
010 0101	Data Address Compare 2 (DAC2)
010 0110	Reserved (DVC1 future use)
010 0111	Reserved (DVC2 future use)
010 1000–010 1011	Reserved
010 1100	Debug Counter Register (DBCNT)
010 1101–010 1111	Reserved
011 0000	Debug Status Register (DBSR)
011 0001	Debug Control Register 0 (DBCR0)
011 0010	Debug Control Register 1 (DBCR1)
011 0011	Debug Control Register 2 (DBCR2)
011 0100	Debug Control Register 3 (DBCR3)
011 0101–101 1111	Reserved (do not access)
110 0000–110 1111	Reserved (do not access)
111 0000–111 1001	General Purpose register selects [0:9]
111 1010	(Reserved))
111 1011	(Reserved)
111 1100	Reserved
111 1101	Reserved

Table 9-9. e200 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 7.3.15.15, "Enable Once Register Select (j_en_once_regsel)."

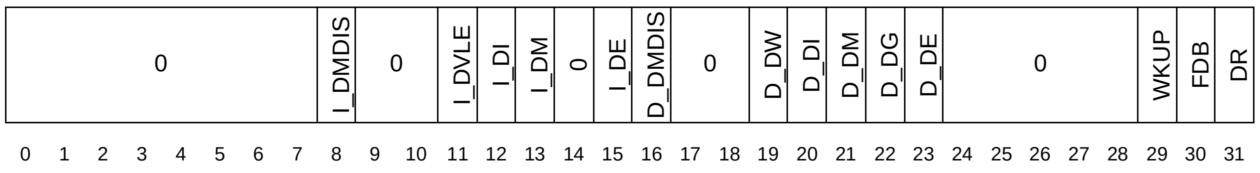
The Once Decoder receives as input the 10-bit command from the OCMD, and status signals from the processor, and generates all the strobes required for reading and writing the selected OnCE registers.

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 re-enters 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 correspond to the first instruction of the exception handler.

Additionally, the 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 DBCRO[EDM]. They may still be modified during a debug session via a single-stepped **mtspr** instruction if DBCRO[EDM] is programmed to a '0', or via OnCE access if DBCR0[EDM] is set.

9.4.5.3 e200 OnCE Control Register (OCR)

The e200 OnCE Control Register is a 32-bit register used to force the e200 core into debug mode and to enable / disable sections of the e200 OnCE control logic. It also provides control over the MMU during a debug session. 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 9-12.



Reset—0xo000_0000 on *m_por*, *j_trst_b*, or entering Test_logic_Reset state

Figure 9-12. OnCE Control Register

Table 9-10 provides bit definitions for the OnCE Control Register.

Table 9-10. OnCE Control Register Bit Definitions

Bit(s)	Name	Description
0:7	—	Reserved
8	I_DMDIS	Instruction Side Debug MMU Disable Control Bit (I_DMDIS) 0 MMU not disabled for debug sessions 1 MMU disabled for debug sessions This bit may be used to control whether the MMU is enabled normally, or whether the MMU is disabled during a debug session for Instruction Accesses. When enabled, the MMU functions normally. When disabled, for Instruction Accesses, no address translation is performed (1:1 address mapping), and the TLB VLE, I,M, and E bits are taken from the OCR bits I_VLE, I_DI, I_DM, and I_DE bits. The W and G bits are assumed '0'. The SX and UX access permission control bits are set to'1' to allow full access. When disabled, no TLB miss or TLB exceptions are generated for Instruction accesses. External access errors can still occur.
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 9-10. OnCE Control Register Bit Definitions (continued)

Bit(s)	Name	Description
22	D_DG	Data Side Debug TLB 'G' Attribute Bit (D_DG) This bit is used to provide the 'G' attribute bit to be used for Data accesses when the MMU is disabled for Data accesses during a debug session.
23	D_DE	Data Side Debug TLB 'E' Attribute Bit (D_DE) This bit is used to provide the 'E' attribute bit to be used for Data accesses when the MMU is disabled for Data accesses during a debug session.
24:28	—	Reserved
29	WKUP	Wakeup Request Bit (WKUP) This control bit may be used to force the e200 *p_wakeup* output signal to be asserted. This control function may be used by debug firmware to request that the chip-level clock controller restore the *m_clk* input to normal operation regardless of whether the CPU is in a low power state to ensure that debug resources may be properly accessed by external hardware through scan sequences.
30	FDB	Force Breakpoint Debug Mode Bit (FDB) This control bit is used to determine whether the processor is operating in breakpoint debug enable mode or not. The processor may be placed in breakpoint debug enable mode by setting this bit. In breakpoint debug enable mode, execution of the '*bkpt*' pseudo- instruction causes the processor to enter debug mode, as if the $\overline{jd\_de\_b}$ input had been asserted. This bit is qualified with DBCR0[EDM], which must be set for FDB to take effect.
31	DR	CPU Debug Request Control Bit This control bit is used to unconditionally request the CPU to enter the Debug Mode. The CPU indicates that Debug Mode has been entered via the data scanned out in the shift-IR state. 0 No Debug Mode request 1 Unconditional Debug Mode request When the DR bit is set the processor enters Debug mode at the next instruction boundary.

9.4.6 Access to Debug Resources

Resources contained in the e200 OnCE Module which do not require the e200 processor core to be halted for access may be accessed while the e200 core is running, and does not interfere with processor execution. Accesses to other resources such as the CPUSCR require the e200 core to be placed in debug mode to avoid synchronization hazards. Debug firmware may ensure that it is safe to access these resources by determining the state of the e200 core prior to access. Note that a scan operation to update the CPUSCR is required prior to exiting debug mode if debug mode has been entered.

Some cases of write accesses other than accesses to the OnCE Command and Control registers, or the EDM bit of DBCR0 require the e200 *m_clk* to be running for proper operation. The OnCE control register provides a means of signaling this need to a system level clock control module.

In addition, because 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 9-11 provides a list of access requirements for OnCE registers.

Table 9-11. OnCE Register Access Requirements

Register Name	Access Requirements					Notes
	Requires *jd_en_once* to be Asserted	Requires DBCR0 [EDM] = 1	Requires *m_clk* Active for Write Access	Requires CPU to be Halted for Read Access	Requires CPU to be Halted for Write Access	
Enable_OnCE	N	N	N	N	—	—
Bypass	N	N	N	N	N	—
CPUSCR	Y	Y	Y	Y	Y	—
DAC1	Y	Y	Y	N	*[1]	—
DAC2	Y	Y	Y	N	*[1]	—
DBCNT	Y	Y	Y	N[2]	*[1]	—
DBCR0	Y	Y	Y	N	*[1]	*DBCR0[EDM] access only requires *jd_en_once* asserted
DBCR1	Y	Y	Y	N	*[1]	—
DBCR2	Y	Y	Y	N	*[1]	—
DBCR3	Y	Y	Y	N	*[1]	—
DBSR	Y	Y	Y	N[3]	*[1]	—
IAC1	Y	Y	Y	N	*[1]	—
IAC2	Y	Y	Y	N	*[1]	—
IAC3	Y	Y	Y	N	*[1]	—
IAC4	Y	Y	Y	N	*[1]	—
JTAG ID	N	N	—	N	—	Read-only
OCR	Y	N	N	N	N	—
OSR	Y	N	—	N	—	Read-only, accessed by scanning out IR while *jd_en_once* is asserted
Cache Debug Access Control (CDACNTL)[3]	Y	N	Y	Y	Y	CPU must be in debug mode with clocks running
Cache Debug Access Data (CDADATA)[3]	Y	N	Y	Y	Y	CPU must be in debug mode with clocks running
External GPRs	Y	N	N	N	N	—
LSRL Select	Y	N	?	?	?	System Test logic implementation determines LSRL functionality

Notes:

[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.

[3] Not present on e200z1

9.4.7 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 e200 Debug Mode is entered assuming the OnCE circuitry has been enabled. e200 OnCE operation is enabled by the assertion of the *jd_en_once* input (see Section 9.4.4.1).

9.4.7.1 External Debug Request During RESET

Holding the *jd_de_b* signal asserted during the assertion of *p_reset_b*, and continuing to hold it asserted following the negation of *p_reset_b* causes the e200 core to enter Debug Mode. After receiving an acknowledge via the OnCE Status Register DEBUG bit, the external command controller should negate the *jd_de_b* signal before sending the first command. Note that in this case the e200 core does not execute an instruction before entering Debug Mode, although the first instruction to be executed may be fetched prior to entering Debug Mode.

In this case, all values in the debug scan chain are undefined, and the external Debug Control Module is responsible for proper initialization of the chain before debug mode is exited. In particular, the exception processing associated with reset, may not be performed when the debug mode is exited, 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.

9.4.7.2 Debug Request During RESET

Asserting a debug request by setting the DR bit in the OCR during the assertion of *p_reset_b* causes the chip to enter debug mode. In this case the chip may fetch the first instruction of the reset exception handler, but does not execute an instruction before entering debug mode. In this case, all values in the debug scan chain are undefined, and the external Debug Control Module is responsible for proper initialization of the chain before debug mode is exited. In particular, the exception processing associated with reset may not be performed when the debug mode is exited, thus, the Debug controller must initialize the PC, MSR, and IR to the image that the processor would have obtained in performing reset exception processing, or must cause the appropriate reset to be re-asserted.

9.4.7.3 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 are aborted by the interrupt processing.

9.4.7.4 Debug Request During Waiting, Halted or Stopped State

Asserting a debug request by setting the DR bit in the OCR when the chip is in the Waiting state (*p_waiting* asserted), Halted state (*p_halted* asserted) or Stopped state (*p_stopped* asserted) causes the CPU to exit the state and enter the debug mode once the CPU clock *m_clk* has been restored. Note that in this case, the CPU negates the *p_waiting*, *p_halted* and *p_stopped* outputs. Once the debug session has ended, the CPU returns to the state it was in prior to entering debug mode.

To signal the chip-level clock generator to re-enable *m_clk*, the *p_wakeup* output is asserted whenever the debug block is asserting a debug request to the CPU due to OCR[DR] being set, or *jd_de_b* assertion, and remains set from then until the debug session ends (*jd_debug_b* goes from 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.

9.4.7.5 Software Request During Normal Activity

Upon executing a “*bkpt*” pseudo-instruction (for e200, defined to be an all 0’s instruction opcode) when the OCR register’s (FDB) bit is set (debug mode enable control bit is true), and DBCR0[EDM]=1, the CPU enters the debug mode after the instruction following the “*bkpt*” pseudo-instruction has entered the instruction register.

9.4.8 CPU Status and Control Scan Chain Register (CPUSCR)

A number of on-chip registers store the CPU pipeline status and are configured in a single scan chain for access by the e200 OnCE controller. The CPUSCR register contains these processor resources, which are used to restore the pipeline and resume normal chip activity upon return from the debug mode, as well as a mechanism for the emulator software to access processor and memory contents. Figure 9-13 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.

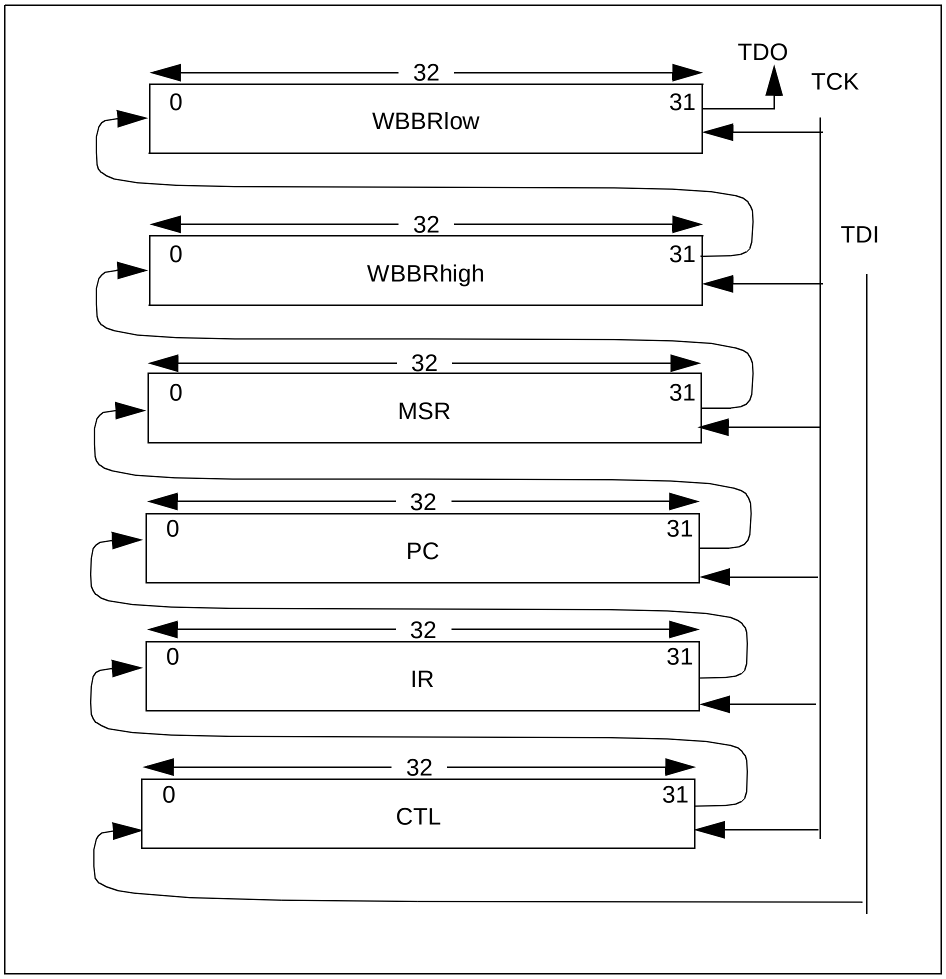


Figure 9-13. CPU Scan Chain Register (CPUSCR)

9.4.8.1 Instruction Register (IR)

The Instruction Register (IR) provides a mechanism for controlling the debug session by serving as a means for forcing in selected instructions, and then causing them to be executed in a controlled manner by the debug control block. The opcode of the next instruction to be executed when entering debug mode is contained in this register when the scan-out of this chain begins. This value should be saved for later restoration if continuation of the normal instruction stream is desired.

On scan-in, in preparation for exiting debug mode, this register is filled with an instruction opcode selected by debug control software. By selecting appropriate instructions and controlling the execution of those instructions, the results of execution may be used to examine or change memory locations and processor registers. The debug control module external to the processor core controls execution by providing a single-step capability. Once the debug session is complete and normal processing is to be resumed, this register may be loaded with the value originally scanned out.

9.4.8.2 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. Figure 9-14.

*															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 9-14. Control State Register (CTL)

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 determines 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

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

* — Internal State Bits

These control bits represent internal processor state and should be restored to their original value after a debug session is completed, i.e when a e200 OnCE command is issued with the GO and EX bits set and not ignored. When performing instruction execution during a debug session (see Section 9.4.4.3, “e200 OnCE Debug Output (jd_debug_b)”) which is not part of the normal program execution flow, these bits should be set to a 0.

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 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.

FFRA— Feed Forward RA Operand Bit

This control bit causes the content of the WBBR_{low} to be used as the RA (RS for logical and shift operations or RX for VLE se_ instructions) operand value of the first instruction to be executed following an update of the CPUSCR. This allows the debug firmware to update processor registers — initialize the WBBR_{low} 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_{low} used as RA (RS for logical and shift operations) operand value

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.

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.

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.

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.

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.

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.

IRStat6—IR Status Bit 6

This control bit indicates a Parity Error status for the IR. (Note: this bit is reserved.)

0 = No Parity Error occurred on the fetch of this instruction.
1 = Parity Error occurred on the fetch of this instruction.

IRStat7—IR Status Bit 7

This control bit indicates a Precise External Termination Error status for the IR, or a 2nd half TLB Miss for the instruction in the IR.

0 = 0 = No Precise External Termination Error occurred on the fetch of this instruction.
1 = If IRStat1 = ‘0’, a Precise External Termination Error occurred on the fetch of this instruction.
If IRStat1 = ‘1’, a TLB Miss occurred on the 2nd half of this instruction.

IRStat8—IR Status Bit 8

This control bit indicates the Power Architecture VLE status for the IR.

0 = IR contains a Book E instruction.
1 = IR contains a Power Architecture VLE instruction, aligned in the Most Significant Portion of IR if 16-bit.

IRStat9—IR Status Bit 9

This control bit indicates the Power Architecture VLE Byte-ordering Error status for the IR, or a Book E 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 Book E 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 is used to determine the type of instruction present in the IR.

Just prior to exiting debug mode with a go+exit, the PCINV status bit which 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 generally points to the last completed instruction, although that instruction is not re-executed. The **nop** instruction is executed instead, and instruction fetch and execution resumes at location PC+4. IRStat8 is used to determine the type of instruction present in the IR, thus should be cleared in this case.

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 IRStat2-5 should be set to ‘0’ to avoid re-entry into debug mode for an instruction breakpoint request. Upon exiting debug mode with go+exit, if one of these bits is set, debug mode is re-entered prior to any further instruction execution.

9.4.8.3 Program Counter Register (PC)

The PC is a 32-bit register that stores the value of the program counter which was present when the chip entered the debug mode. It is affected by the operations performed during the debug mode and must be restored by the external command controller when the CPU returns to normal mode. 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** is executed, and instruction fetch and execution resumes at PC+4.

9.4.8.4 Write-Back Bus Register ($WBBR_{low}$, $WBBR_{high}$)

WBBR is used as a means of passing operand information between the CPU and the external command controller. Whenever the external command controller needs to read the contents of a register or memory location, it forces the chip to execute an instruction that brings that information to WBBR. $WBBR_{low}$ holds the 32-bit result of most instructions including load data returned for a load or load with update instruction. $WBBR_{high}$ holds the updated effective address calculated by a load with update instruction. It is undefined for other instructions.

As an example, to read the lower 32 bits of processor register **r1**, an **ori r1,r1,0** instruction is executed, and the result value of the instruction is latched into $WBBR_{low}$. The contents of $WBBR_{low}$ can 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 9.4.8.2, “Control State Register (CTL),” for more detail on the CTL[FFRA] bit).

$WBBR_{low}$ and $WBBR_{high}$ are generally undefined on instructions which do not writeback a result, and due to control issues are not defined on **lmw** or branch instructions as well.

9.4.8.5 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.

9.4.9 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.

9.5 Watchpoint Support

e200 supports the generation and signalling of watchpoints when operating in internal debug mode (DBCR0[IDM]=1) or in external debug mode (DBCR0[EDM]=1). Watchpoints are indicated with a dedicated set of interface signals. The *jd_watchpoint*[0:7] output signals are used to indicate that a watchpoint has occurred.

Each debug address compare function (IAC1-4, DAC1-2), and Debug Counter event (DCNT1-2) is 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 (except during a debug session). If not desired, the base address values for these events may be programmed to an unused system address. MSR[DE] has no effect on watchpoint generation.

External logic may monitor the assertion of these signals for debugging purposes. Watchpoints are signaled in the clock cycle following the occurrence of the actual event. The module also monitors assertion of these signals for various development control purposes.

Table 9-12. Watchpoint Output Signal Assignments

Signal Name	Type	Description
jd_watchpt[0]	IAC1	Instruction Address Compare 1 watchpoint Asserted whenever an IAC1 compare occurs regardless of being enabled to set DBSR status
jd_watchpt[1]	IAC2	Instruction Address Compare 2 watchpoint Asserted whenever an IAC2 compare occurs regardless of being enabled to set DBSR status
jd_watchpt[2]	IAC3	Instruction Address Compare 3 watchpoint Asserted whenever an IAC3 compare occurs regardless of being enabled to set DBSR status
jd_watchpt[3]	IAC4	Instruction Address Compare 4 watchpoint Asserted whenever an IAC4 compare occurs regardless of being enabled to set DBSR status
jd_watchpt[4]	DAC1[1]	Data Address Compare 1 watchpoint Asserted whenever a DAC1 compare occurs regardless of being enabled to set DBSR status

Table 9-12. Watchpoint Output Signal Assignments (continued)

Signal Name	Type	Description
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

[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.

9.6 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

- The debugger should ensure that the *jd_en_once* control signal is asserted in order to enable OnCE operation
- Select the OCR and write a value to it in which OCR[DR], OCR[WKUP], are set to ‘1’. The tap controller must step through the proper states as outlined earlier. This step places the CPU in a debug state in which it is halted and awaiting single-step commands or a release to normal mode
- Scan out the value of the OSR to determine that the CPU clock is running and the CPU has entered the Debug state. This can be done in conjunction with a Read of the CPUSCR. The OSR is shifted out during the Shift_IR state. The CPUSCR is shifted out during the Shift_DR state. The debugger should save the scanned-out value of CPUSCR for later restoration.
- Select the DBCR0 register and update it with the DBCR0[EDM] bit set
- Clear the DBSR status bits
- Write appropriate values to the DBCRx, IAC, DAC, and DBCNT registers. Note that the initial write to DBCR0 only affects the EDM bit, so the remaining portion of the register must now be initialized, keeping the EDM bit set

At this point the system is ready to commence debug operations. Depending on the desired operation, different steps must occur.

- Optionally, set the OCR[I_DMDIS] and/or OCR[I_DMDIS] control bits to ensure that no TLB misses occur while performing the debug operations
- Optionally, ensure that the values entered into the MSR portion of the CPUSCR during the following steps cause interrupt to be disabled (clearing MSR[EE] and MSR[CE). This ensures that external interrupt sources do not cause single-step errors.

To single-step the CPU:

- 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 9.4.8.2, "Control State Register (CTL)"), with a Go+Noexit OnCE Command value.
- 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)

- The OCR[DMDIS], OCR[DR], control bits should be cleared, leaving the OCR[WKUP] bit set
- The debugger restores the CPUSCR with a previously saved value of the CPUSCR (with appropriate modification of the PC and IR as described in Section 9.4.8.2, "Control State Register (CTL)"), with a Go+Exit OnCE Command value.
- The OCR[WKUP] bit may then be cleared

To exit External Debug Mode

- The debugger should place the CPU in the debug state via the OCR[DR] with OCR[WKUP] asserted, scanning out and saving the CPUSCR
- The debugger should write the DBCRx registers as needed, likely clearing every enable except the DBCR0[EDM] bit
- The debugger should write the DBSR to a cleared state
- The debugger should re-write the DBCR0 with all bits including EDM cleared
- The debugger should clear the OCR[DR] bit
- 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 9.4.8.2, "Control State Register (CTL)"), with a Go+Exit OnCE Command value.
- 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.

9.7 Parallel Signature Unit

To support applications requiring system integrity checking during operation, the e200 core provides a Parallel Signature unit, which is capable of monitoring the CPU data read and data write AHB buses, and accumulating a 32-bit MISR signatures of the data value transferred.

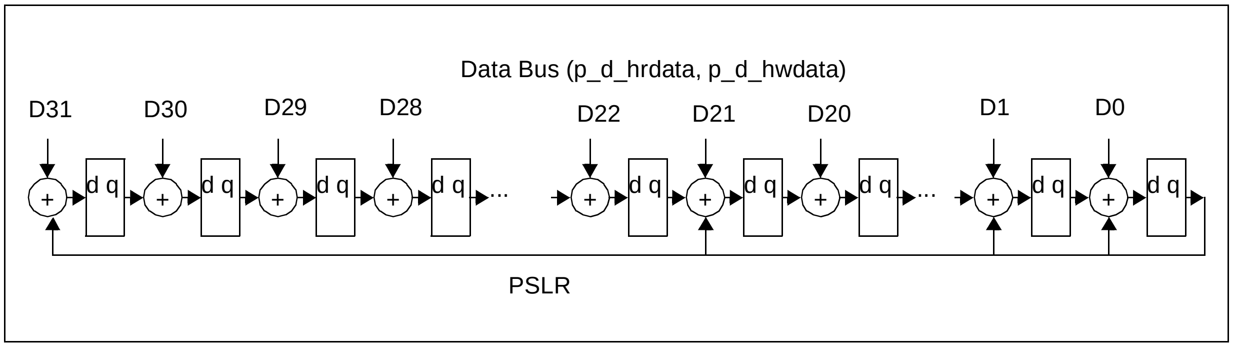
The primitive polynomial used is $P(X)=1+X^{10}+X^{30}+X^{31}+X^{32}$. Values are accumulated based on an initially programmed "seed" value, and are qualified based on active byte lanes of the data read and data write buses (*p_d_hrdata*[31:0], *p_d_hwdata*[31:0]) as indicated via the *p_d_hbstrb*[3:0] signals. Inactive byte lanes use a value of all zeros as input data to the MISRs. Refer to Table 7-11 for active byte lane information. If a transfer error occurs on any accumulated read data, the returned read data is ignored, a value of all zeros is used instead, and the error is logged. Errors occurring on data writes are not logged, since the data driven by the CPU is valid.

The unit may be independently enabled for read cycles and write cycles, allowing for flexible usage. Software may also control accumulation of software provided values via a pair of update registers. In addition, a counter is provided for software use to monitor the number of beats of data which have been compressed.

Updates are performed when the parallel signature registers are initialized, when a qualified 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



The Parallel Signature unit consists of six registers as described below. Access to these registers is privileged. No user-mode access is allowed.

NOTE

Proper access of the PSU registers requires that the **mfdcr** instruction which reads a PSU register be preceded by either an **mbar** or an **msync** instruction. To ensure that the effects of 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**).

9.7.1 Parallel Signature Control Register (PSCR)

The Parallel Signature Control Register (PSCR) controls operation of the Parallel Signature unit.

0	CNTEN	0	RDEN	WREN	INIT
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—272; Read/Write; Reset—0x0

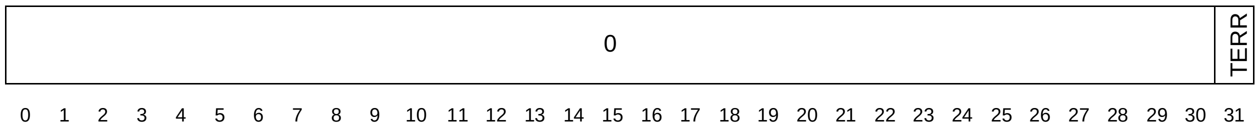
Figure 9-15. Parallel Signature Control Register (PSCR)

Table 9-13. 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 PSLR, and PSCTR registers to all '0's (0x00000000). This bit always reads as '0'.

9.7.2 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 9-16. Parallel Signature Status Register (PSSR)

Table 9-14. 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.

9.7.3 Parallel Signature Low Register (PSLR)

The Parallel Signature Low Register (PSLR) provides signature information for bits 31:0 of the AHB data read and data write buses. It may be written via a **mtdcr pslr, Rs** instruction (DCR register 275) to initialize a seed value prior to enabling signature accumulation. The PSCR[INIT] control bit may also be used to clear the PSLR. This register is unaffected by system reset, thus should be initialized by software prior to performing parallel signature operations.

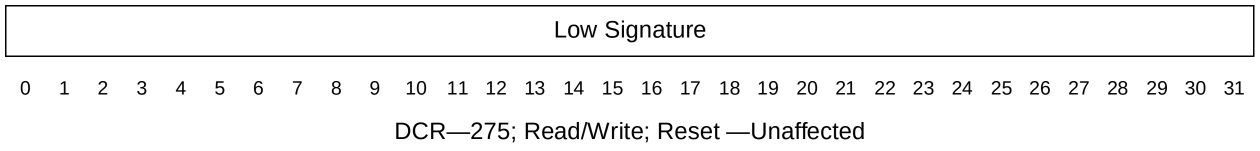


Figure 9-17. Parallel Signature Low Register (PSLR)

9.7.4 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.

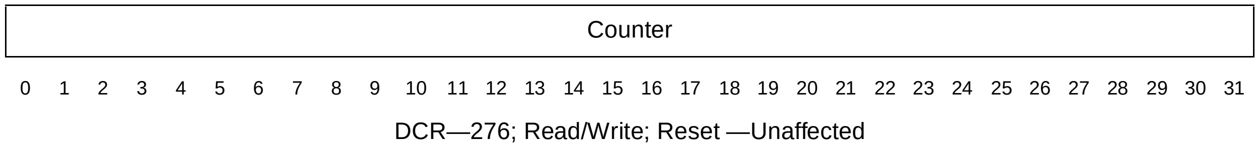


Figure 9-18. Parallel Signature Counter Register (PSCTR)

9.7.5 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.

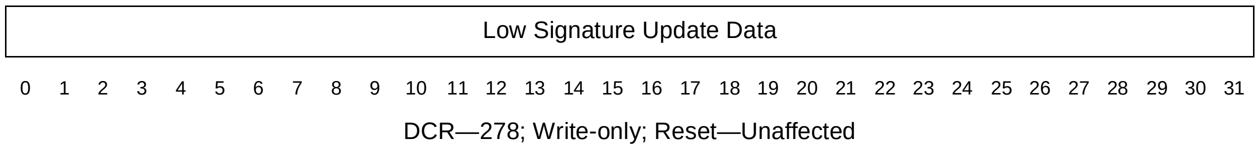


Figure 9-19. Parallel Signature Update Low Register (PSULR)

Appendix A
Register Summary

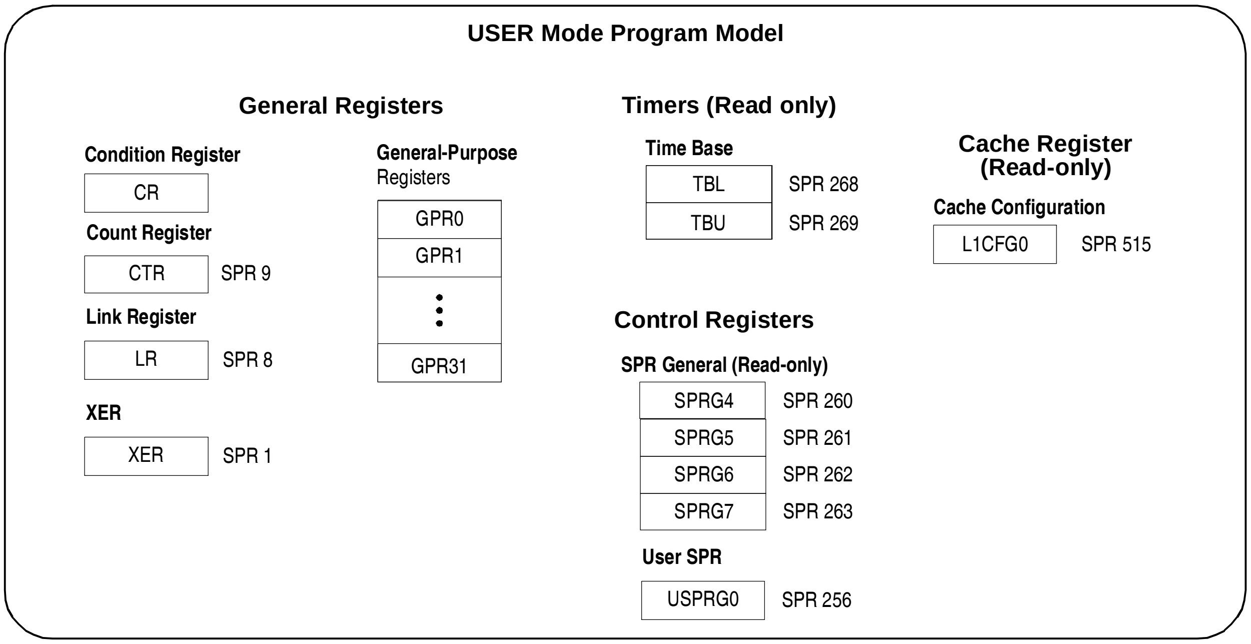


Figure A-1. e200 User Mode Registers

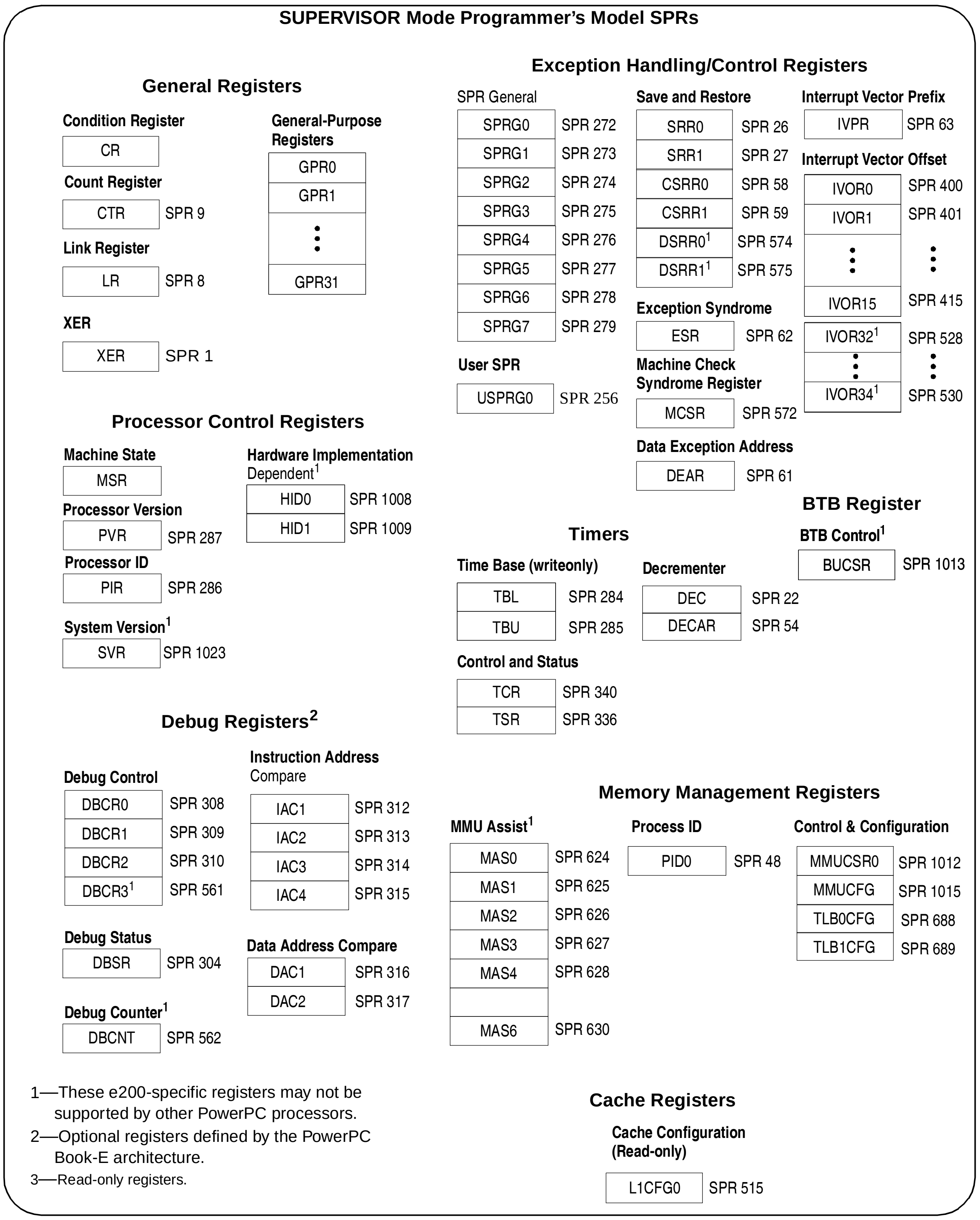


Figure A-2. e200 Supervisor Mode Registers

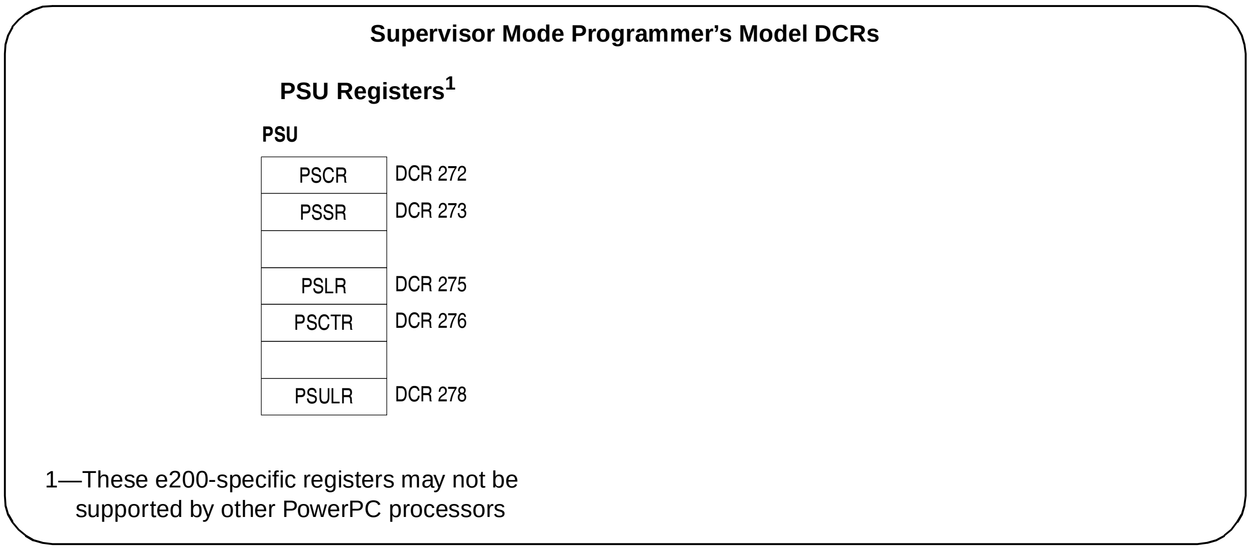

Supervisor Mode Programmer's Model DCRs
PSU Registers[1]
PSU
PSCR DCR 272
PSSR DCR 273
PSLR DCR 275
PSCTR DCR 276
PSULR DCR 278
1—These e200-specific registers may not be supported by other PowerPC processors

Figure A-3. e200 Supervisor Mode Device Control Registers (DCR)

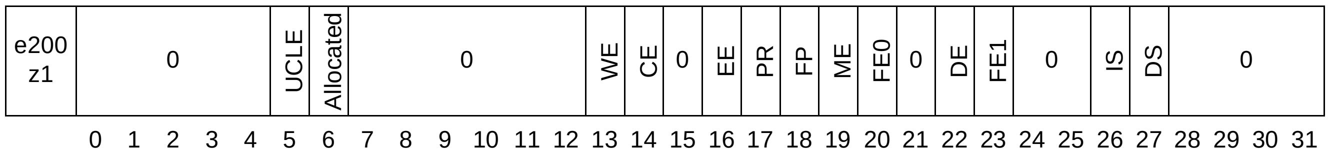


Figure A-4. Machine State Register (MSR)

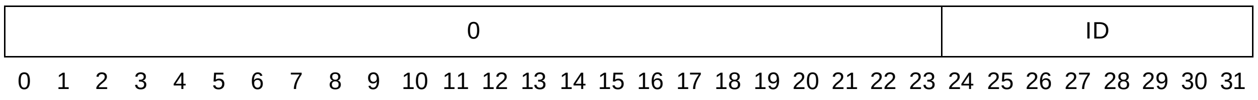


Figure A-5. Processor ID Register (PIR)

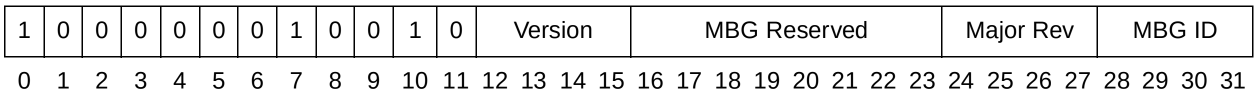


Figure A-6. Processor Version Register (PVR)

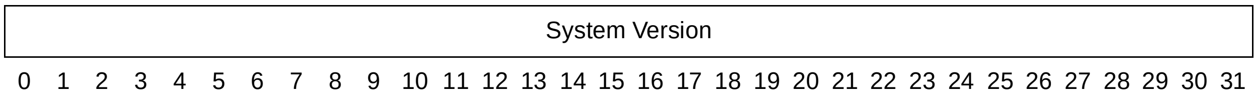


Figure A-7. System Version Register (SVR)

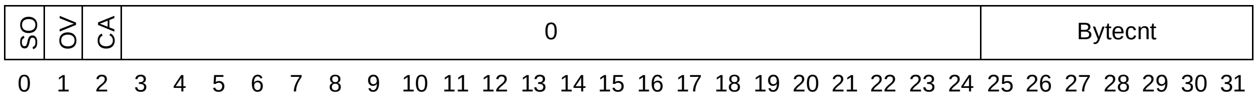


Figure A-8. Integer Exception Register (XER)

0–3	4	5	6	7	8	9	10	11	12	13	14	15	16–23	24	25	26	27–29	30	31
0	PIL	PPR	PTR	FP	ST	0	DLK	ILK	AP	PUO	BO	PIE	0	EFP	0	VLEMI	0	MIF	XTE

Figure A-9. Exception Syndrome Register (ESR)

	0	1	2	3	4	5–26	27	28	29	30–31
e200z1	MCP	0	CP_PERR	CPERR	EXCP_ERR	0	BUS_IRERR	BUS_DRERR	BUS_WRERR	0

Figure A-10. 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-11. Timer Control Register (TCR)

0	1	2–3	4	5	6–31
ENW	WIS	WRS	DIS	FIS	0

Figure A-12. Timer Status Register (TSR)

0	1–5	6–7	8	9	10	11–13	14	15	16	17	18	19	20	21	22	23	24–31
EMCP	0	BPRED	DOZE	NAP	SLEEP	0	ICR	NHR	0	TBEN	SEL TBCK	DCLREE	DCLRCE	CICLRDE	MCCLRDE	DAPUEN	0

Figure A-13. Hardware Implementation Dependent Register 0 (HID0)

0–15	16–23	24	25–31
0	SYSCTL	ATS	0

Figure A-14. Hardware Implementation Dependent Register 1 (HID1)

0–21	22	23–30	31
0	BBFI	0	BPEN

Figure A-15. Branch Unit Control and Status Register (BUCSR)

0
0–31

Figure A-16. Context Control Register (CTXCR)

0	Vector Offset	0
0–15	16–27	28–31

Figure A-17. e200 Interrupt Vector Offset Register (IVOR)

CNT1	CNT2
0–16	17–31

Figure A-18. DBCNT Register

EDM	IDM	RST	ICMP	BRT	IRPT	TRAP	IAC1	IAC2	IAC3	IAC4	DAC1	DAC2	RET	0	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	17–20	21	22	23	24	25	26	27–30	31

IAC1US	IAC1ER	IAC2US	IAC2ER	IAC12M	0	IAC3US	IAC3ER	IAC4US	IAC4ER	IAC34M	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-20. DBCR1 Register

DAC1US	DAC1ER	DAC2US	DAC2ER	DAC12M	DAC1LNK	DAC2LNK	0
0–1	2–3	4–5	6–7	8–9	10	11	12–31

Figure A-21. DBCR2 Register

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

Figure A-22. DBCR3 Register

IDE	UDE	MRR		ICMP	BRT	IRPT	TRAP	IAC1	IAC2	IAC3	IAC4	DAC1R	DAC1W	DAC2R	DAC2W	RET	0				DEVT1	DEVT2	DCNT1	DCNT2	CIRPT	CRET	VLES	0			CNT1TRG
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Figure A-23. DBSR Register

MCLK	ERR	CHKSTOP	RESET	HALT	STOP	DEBUG	WAIT	0	1
0	1	2	3	4	5	6	7	8	9

Figure A-24. OnCE Status Register

R/W	GO	EX	RS[0:6]						
0	1	2	3	4	5	6	7	8	9

Figure A-25. OnCE Command Register

0																DMDIS	0		DW	DI	DM	DG	DE	0					WKUP	FDB	DR
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-26. OnCE Control Register

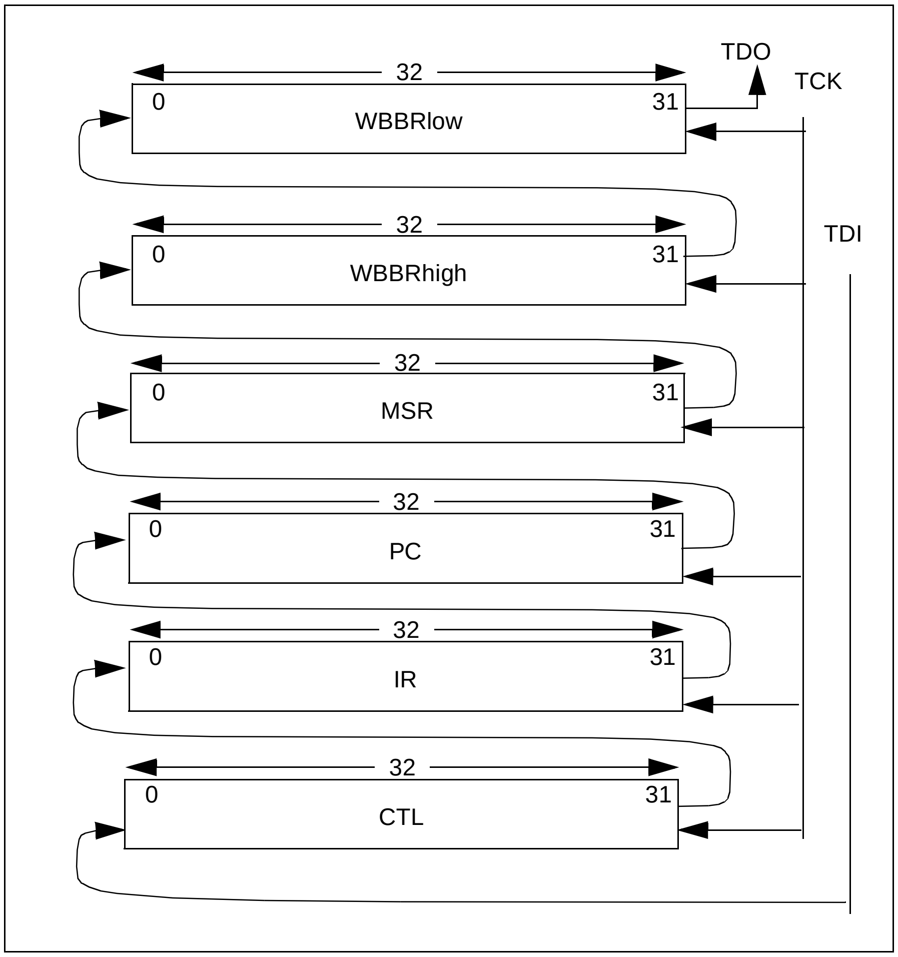


Figure A-27. CPU Scan Chain Register (CPUSCR)

Bits	Field
0–14	*
15	WAITING
16–19	PCOFST
20	PCINV
21	FFRA
22	IRSTAT0
23	IRSTAT1
24	IRSTAT2
25	IRSTAT3
26	IRSTAT4
27	IRSTAT5
28	IRSTAT6
29	IRSTAT7
30	IRSTAT8
31	IRSTAT9

Figure A-28. Control State Register (CTL)

Bits	Field
0–1	CARCH
2	CWPA
3	CFAHA
4	CFIWA
5–6	0
7–8	CBSIZE
9–10	CREPL
11	CLA
12	CPA
13–20	CNWAY
21–31	CSIZE

Figure A-29. L1 Cache Configuration Register 0 (L1CFG0)

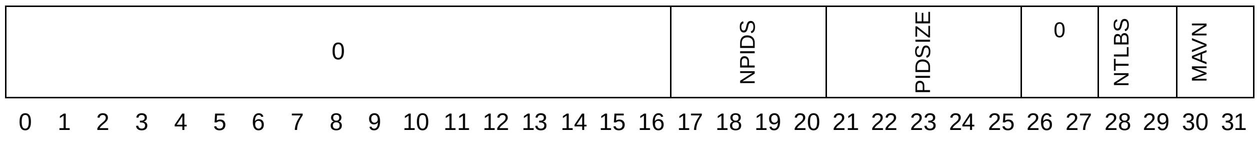


Figure A-30. MMU Configuration Register (MMUCFG)

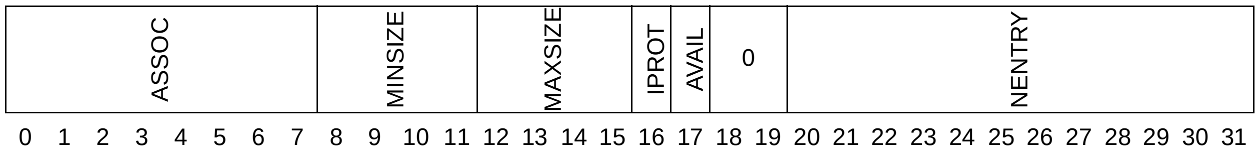


Figure A-31. TLB Configuration Registers (TLB0CFG, TLB1CFG)

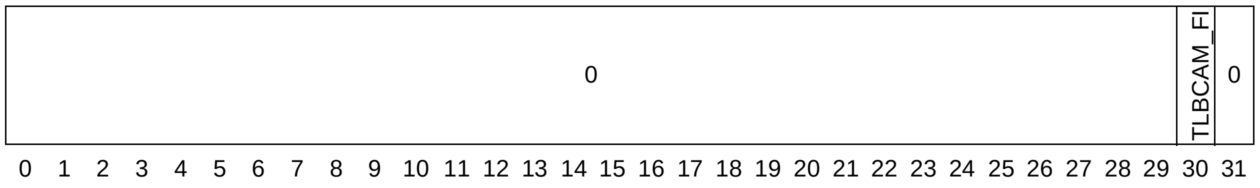


Figure A-32. MMU Control and Status Register 0 (MMUCSR0)

Register	0	1	2–3	4–7	8–12	13	14–15	16–18	19	20–21	22	23	24	25	26	27	28	29	30	31
MAS0	0		TLBSEL (01)	0		ESELCAM		0										NVCAM		
MAS1	VALID	IPROT	0		TID			0	TS	TSIZ			0							
MAS2	EPN									0					VLE	W	I	M	G	E
MAS3	RPN									0	U0	U1	U2	U3	UX	SX	UW	SW	UR	SR
MAS4	0		TLBSELD (01)	0			TIDSELD	0		TSIZED			0		VLED	WD	ID	MD	GD	ED
MAS6	0				SPID			0												SAS

Figure A-33. MMU Assist Registers Summary

0–25	26	27–28	29	30	31
0	CNTEN	0	RDEN	WREN	INIT

DCR—272; Read/Write; Reset—0x0

Figure A-34. Parallel Signature Control Register (PSCR)

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

DCR—273; Read/Write; Reset —Unaffected

Figure A-35. Parallel Signature Status Register (PSSR)

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-36. 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-37. Parallel Signature Counter Register (PSCTR)

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-38. Parallel Signature Update Low Register (PSULR)

Appendix B
Revision History

This appendix provides a list of the major differences between revisions of the *e200z1 Power Architecture™ Reference Manual*. This is the initial version of the manual so there currently are no differences.

Glossary

The glossary contains an alphabetical list of terms, phrases, and abbreviations used in this reference manual.

A

Architecture. A detailed specification of requirements for a processor or computer system. It does not specify details of how the processor or computer system must be implemented; instead it provides a template for a family of compatible *implementations*.

Atomic access. A bus access that attempts to be part of a read-write operation to the same address uninterrupted by any other access to that address (the term refers to the fact that the transactions are indivisible). The Power Architecture technology implements atomic accesses through the **lwarx**/**stwcx.** instruction pair.

Autobaud. The process of determining a serial data rate by timing the width of a single bit.

B

Beat. A single state on the bus interface that may extend across multiple bus cycles. A transaction can be composed of multiple address or data *beats*.

Big-Endian. A byte-ordering method in memory where the address *n* of a word corresponds to the *most significant byte*. In an addressed memory word, the bytes are ordered (left to right) 0, 1, 2, 3, with 0 being the *most significant byte*. See *Little-Endian*.

Boundedly undefined. A characteristic of certain operation results that are not rigidly prescribed by the Power Architecture technology. Boundedly-undefined results for a given operation may vary among implementations and between execution attempts in the same implementation.

Although the architecture does not prescribe the exact behavior for when results are allowed to be boundedly undefined, the results of executing instructions in contexts where results are allowed to be boundedly undefined are constrained to ones that could have been achieved by executing an arbitrary sequence of defined instructions, in valid form, starting in the state the machine was in before attempting to execute the given instruction.

Breakpoint. A programmable event that forces the core to take a breakpoint exception.

Burst. A multiple-beat data transfer whose total size is typically equal to a cache block.

Bus clock. Clock that causes the bus state transitions.

Bus master. The owner of the address or data bus; the device that initiates or requests the transaction.

C

Cache. High-speed memory containing recently accessed data or instructions (subset of main memory).

Cache block. A small region of contiguous memory that is copied from memory into a *cache*. The size of a cache block may vary among processors; the maximum block size is one *page*. In Power Architecture processors, *cache coherency* is maintained on a cache-block basis. Note that the term ‘cache block’ is often used interchangeably with ‘cache line.’

Cache coherency. An attribute wherein an accurate and common view of memory is provided to all devices that share the same memory system. Caches are coherent if a processor performing a read from its cache is supplied with data corresponding to the most recent value written to memory or to another processor’s cache.

Cache flush. An operation that removes from a cache any data from a specified address range. This operation ensures that any modified data within the specified address range is written back to main memory. This operation is generated typically by a Data Cache Block Flush (**dcbf**) instruction.

Caching-inhibited. A memory update policy in which the *cache* is bypassed and the load or store is performed to or from main memory.

Cast out. A *cache block* that must be written to memory when a cache miss causes a cache block to be replaced.

Changed bit. One of two *page history bits* found in each *page table entry* (PTE). The processor sets the changed bit if any store is performed into the *page*. See also *Page access history bits* and *Referenced bit*.

Clean. An operation that causes a cache block to be written to memory, if modified, and then left in a valid, unmodified state in the cache.

Clear. To cause a bit or bit field to register a value of zero. See also *Set*.

Completer. In PCI-X, a completer is the device addressed by a transaction (other than a split completion transaction). If a target terminates a transaction with a split response, the completer becomes the initiator of the subsequent split completion.

Context synchronization. An operation that ensures that all instructions in execution complete past the point where they can produce an *exception*, that all instructions in execution complete in the context in which they began execution, and that all subsequent instructions are *fetched* and executed in the new context. Context synchronization may result from executing specific instructions (such as **isync** or **rfi**) or when certain events occur (such as an exception).

Copy-back operation. A cache operation in which a cache line is copied back to memory to enforce cache coherency. Copy-back operations consist of snoop push-out operations and cache cast-out operations.

D

Direct-mapped cache. A cache in which each main memory address can appear in only one location within the cache; operates more quickly when the memory request is a cache hit.

Double data rate. Memory that allows data transfers at the start and end of a clock cycle. thereby doubling the data rate.

E

Effective address (EA). The 32-bit address specified for a load, store, or an instruction fetch. This address is then submitted to the MMU for translation to either a *physical memory* or an I/O address.

Exclusive state. MEI state (E) in which only one caching device contains data that is also in system memory.

F

Fetch. Retrieving instructions from either the cache or main memory and placing them into the instruction queue.

Flush. An operation that causes a cache block to be invalidated and the data, if modified, to be written to memory.

Frame-check sequence (FCS). Specifies the standard 32-bit cyclic redundancy check (CRC) obtained using the standard CCITT-CRC polynomial on all fields except the preamble, SFD, and CRC.

G

General-purpose register (GPR). Any of the 32 registers in the general-purpose register file. These registers provide the source operands and destination results for all integer data manipulation instructions. Integer load instructions move data from memory to GPRs and store instructions move data from GPRs to memory.

Gigabit media-independent interface (GMII) sublayer. Sublayer that provides a standard interface between the MAC layer and the physical layer for 1000-Mbps operation. It isolates the MAC layer and the physical layer, enabling the MAC layer to be used with various implementations of the physical layer.

Guarded. The guarded attribute pertains to out-of-order execution. When a page is designated as guarded, instructions and data cannot be accessed out-of-order.

H

Harvard architecture. An architectural model featuring separate caches and other memory management resources for instructions and data.

I

Illegal instructions. A class of instructions that are not implemented for a particular processor. These include instructions not defined by the architecture. In addition, for 32-bit implementations, instructions that are defined only for 64-bit implementations are considered to be illegal instructions. For 64-bit implementations, instructions that are defined only for 32-bit implementations are considered to be illegal instructions.

Implementation. A particular processor that conforms to the architecture, but may differ from other architecture-compliant implementations (for example, in design, feature set, and implementation of *optional* features).

Imprecise exception. A type of *synchronous exception* that is allowed not to adhere to the precise exception model (see *Precise exception*). The Power Architecture technology allows only floating-point exceptions to be handled imprecisely.

Inbound ATMU windows. Mappings that perform address translation from the external address space to the local address space, attach attributes and transaction types to the transaction, and map the transaction to its target interface.

In-order. An aspect of an operation that adheres to a sequential model. An operation is said to be performed *in-order* if, at the time that it is performed, it is known to be required by the sequential execution model.

Integer unit. An execution unit in the core responsible for executing integer instructions.

Instruction latency. The total number of clock cycles necessary to execute an instruction and make ready the results of that instruction.

K

Kill. An operation that causes a *cache block* to be invalidated without writing any modified data to memory.

L

L2 cache. Level-2 cache. See *Secondary cache*.

Latency. The number of clock cycles necessary to execute an instruction and make ready the results of that execution for a subsequent instruction.

Least significant bit (lsb). The bit of least value in an address, register, field, data element, or instruction encoding.

Least significant byte (LSB). The byte of least value in an address, register, data element, or instruction encoding.

Little-Endian. A byte-ordering method in memory where the address *n* of a word corresponds to the *least significant byte*. In an addressed memory word, the bytes are ordered (left to right) 3, 2, 1, 0, with 3 being the *most significant byte*. See *Big-Endian*.

Local access window. Mapping used to translate a region of memory to a particular target interface, such as the DDR SDRAM controller or the PCI controller. The local memory map is defined by a set of eight local access windows. The size of each window can be configured from 4 Kbytes to 2 Gbytes.

M

Media access control (MAC) sublayer. Sublayer that provides a logical connection between the MAC and its peer station. Its primary responsibility is to initialize, control, and manage the connection with the peer station.

Medium-dependent interface (MDI) sublayer. Sublayer that defines different connector types for different physical media and PMD devices.

Media-independent interface (MII) sublayer. Sublayer that provides a standard interface between the MAC layer and the physical layer for 10/100-Mbps operations. It isolates the MAC layer and the physical layer, enabling the MAC layer to be used with various implementations of the physical layer.

Memory access ordering. The specific order in which the processor performs load and store memory accesses and the order in which those accesses complete.

Memory coherency. An aspect of caching in which it is ensured that an accurate view of memory is provided to all devices that share system memory.

Memory consistency. Refers to agreement of levels of memory with respect to a single processor and system memory (for example, on-chip cache, secondary cache, and system memory).

Memory management unit (MMU). The functional unit that is capable of translating an effective (logical) *address* to a physical address, providing protection mechanisms, and defining caching methods.

Memory-mapped accesses. Accesses whose addresses use the page or block address translation mechanisms provided by the MMU and that occur externally with the bus protocol defined for memory.

Modified/exclusive/invalid (MEI). *Cache coherency* protocol used to manage caches on different devices that share a memory system. Note that the Power Architecture technology does not specify the implementation of an MEI protocol to ensure cache coherency.

Modified state. MEI state (M) in which one, and only one, caching device has the valid data for that address. The data at this address in external memory is not valid.

Most significant bit (msb). The highest-order bit in an address, registers, data element, or instruction encoding.

Most significant byte (MSB). The highest-order byte in an address, registers, data element, or instruction encoding.

N

NaN. An abbreviation for not a number; a symbolic entity encoded in floating-point format. There are two types of NaNs—signaling NaNs and quiet NaNs.

No-op. No-operation. A single-cycle operation that does not affect registers or generate bus activity.

O

OCeaN (on-chip network). Non-blocking crossbar switch fabric. Enables full duplex port connections at 128 Gb/s concurrent throughput and independent per port transaction queuing and flow control. Permits high bandwidth, high performance, as well as the execution of multiple data transactions.

Outbound ATMU windows. Mappings that perform address translations from local 32-bit address space to the address spaces of RapidIO or PCI/PCI-X RapidIO, which may be much larger than the local space. Outbound ATMU windows also map attributes such as transaction type or priority level.

P

Packet. A unit of binary data that can be routed through a network. Sometimes packet is used to refer to the frame plus the preamble and start frame delimiter (SFD).

Page. A region in memory. The OEA defines a page as a 4-Kbyte area of memory aligned on a 4-Kbyte boundary.

Page access history bits. The *changed* and *referenced* bits in the PTE keep track of the access history within the page. The referenced bit is set by the MMU whenever the page is accessed for a read or write operation. The changed bit is set when the page is stored into. See *Changed bit* and *Referenced bit*.

Page fault. A page fault is a condition that occurs when the processor attempts to access a memory location that does not reside within a *page* not currently resident in *physical memory*. A page fault exception condition occurs when a matching, valid *page table entry* (PTE[V] = 1) cannot be located.

Page table. A table in memory is comprised of *page table entries*, or PTEs. It is further organized into eight PTEs per PTEG (page table entry group). The number of PTEGs in the page table depends on the size of the page table (as specified in the SDR1 register).

Page table entry (PTE). Data structures containing information used to translate *effective address* to physical address on a 4-Kbyte page basis. A PTE consists of 8 bytes of information in a 32-bit processor and 16 bytes of information in a 64-bit processor.

Physical coding sublayer (PCS). Sublayer responsible for encoding and decoding data stream to and from the MAC sublayer. Medium (1000BASEX) 8B/10B coding is used for fiber. Medium (1000BASET) 8B1Q coding is used for unshielded twisted pair (UTP).

Physical medium attachment (PMA) sublayer. Sublayer responsible for serializing code groups into a bit stream suitable for serial bit-oriented physical devices (SerDes) and vice versa. Synchronization is also performed for proper data decoding in this sublayer. The PMA sits between the PCS and the PMD sublayers. For fiber medium (1000BASEX) the interface on the PMD side of the PMA is a one-bit 1250-MHz signal, while on the PMA PCS side, the interface is a ten-bit interface (TBI) at 125 MHz. The TBI is an alternative to the GMII interface. If the TBI is used, the gigabit Ethernet controller must be capable of performing the PCS function. For UTP medium, the PMD interface side of the PMA consists of four pair of 62.5-MHz PAM5 encoded signals, while the PCS side provides the 1250-Mbps input to an 8B1Q4 PCS.

Physical medium dependent (PMD) sublayer. Sublayer responsible for signal transmission. The typical PMD functionality includes amplifier, modulation, and wave shaping. Different PMD devices may support different media.

Physical memory. The actual memory that can be accessed through the system's memory bus.

Pipelining. A technique that breaks operations, such as instruction processing or bus transactions, into smaller distinct stages or tenures (respectively) so that a subsequent operation can begin before the previous one has completed.

Precise exceptions. A category of exception for which the pipeline can be stopped so instructions that preceded the faulting instruction can complete and subsequent instructions can be flushed and redispatched after exception handling has completed. See *Imprecise exceptions*.

Primary opcode. The most-significant 6 bits (bits 0–5) of the instruction encoding that identifies the type of instruction.

Program order. The order of instructions in an executing program. More specifically, this term is used to refer to the original order in which program instructions are fetched into the instruction queue from the cache.

Protection boundary. A boundary between *protection domains*.

Protection domain. A protection domain is a segment, a virtual page, a BAT area, or a range of unmapped effective addresses. It is defined only when the appropriate relocate bit in the MSR (IR or DR) is 1.

Q

Quad word. A group of 16 contiguous locations starting at an address divisible by 16.

Quiesce. To come to rest. The processor is said to quiesce when an exception is taken or a **sync** instruction is executed. The instruction stream is stopped at the decode stage and executing instructions are allowed to complete to create a controlled context for instructions that may be affected by out-of-order, parallel execution. See *Context synchronization*.

R

rA. The **r**A instruction field is used to specify a GPR to be used as a source or destination.

rB. The **r**B instruction field is used to specify a GPR to be used as a source.

rD. The **r**D instruction field is used to specify a GPR to be used as a destination.

rS. The **r**S instruction field is used to specify a GPR to be used as a source.

RapidIO. High-performance, packet-switched, interconnect architecture that provides reliability, increased bandwidth, and faster bus speeds in an intra-system interconnect. Designed to be compatible with integrated communications processors, host processors, and networking digital signal processors,

Reconciliation sublayer. Sublayer that maps the terminology and commands used in the MAC layer into electrical formats appropriate for the physical layer entities.

Record bit. Bit 31 (or the Rc bit) in the instruction encoding. When it is set, updates the condition register (CR) to reflect the result of the operation.

Reduced instruction set computing (RISC). An *architecture* characterized by fixed-length instructions with nonoverlapping functionality and by a separate set of load and store instructions that perform memory accesses.

Referenced bit. One of two *page history bits* found in each *page table entry*. The processor sets the *referenced bit* whenever the page is accessed for a read or write. See also *Page access history bits*.

Requester. In PCI-X, a requester is an initiator that first introduces a transaction into the PCI-X domain. If a transaction is terminated with a split response, the requester becomes the target of the subsequent split completion.

Reservation. The processor establishes a reservation on a *cache block* of memory space when it executes an **lwarx** instruction to read a memory semaphore into a GPR.

Reservation station. A buffer between the dispatch and execute stages that allows instructions to be dispatched even though the results of instructions on which the dispatched instruction may depend are not available.

S

Secondary cache. A cache memory that is typically larger and has a longer access time than the primary cache. A secondary cache may be shared by multiple devices. Also referred to as L2, or level-2, cache.

Sequence. In PCI-X, a sequence is one or more transactions associated with carrying out a single logical transfer by a requester. Each transaction in the same sequence carries the same unique sequence ID.

Set (*v*). To write a nonzero value to a bit or bit field; the opposite of *clear*. The term ‘set’ may also be used to generally describe the updating of a bit or bit field.

Set (*n*). A subdivision of a *cache*. Cacheable data can be stored in a given location in one of the sets, typically corresponding to its lower-order address bits. Because several memory locations can map to the same location, cached data is typically placed in the set whose *cache block* corresponding to that address was used least recently. See *Set associative*.

Set associative. Aspect of cache organization in which the cache space is divided into sections, called *sets*. The cache controller associates a particular main memory address with the contents of a particular set, or region, within the cache.

Slave. The device addressed by a master device. The slave is identified in the address tenure and is responsible for supplying or latching the requested data for the master during the data tenure.

Snooping. Monitoring addresses driven by a bus master to detect the need for coherency actions.

Snoop push. Response to a snooped transaction that hits a modified cache block. The cache block is written to memory and made available to the snooping device.

Stall. An occurrence when an instruction cannot proceed to the next stage.

Sticky bit. A bit that when *set* must be cleared explicitly.

Superscalar machine. A machine that can issue multiple instructions concurrently from a conventional linear instruction stream.

Supervisor mode. The privileged operation state of a processor. In supervisor mode, software, typically the operating system, can access all control registers and can access the supervisor memory space, among other privileged operations.

Synchronization. A process to ensure that operations occur strictly *in order*. See *Context synchronization*.

Synchronous exception. An *exception* that is generated by the execution of a particular instruction or instruction sequence. There are two types of synchronous exceptions, *precise* and *imprecise*.

System memory. The physical memory available to a processor.

T

Tenure. The period of bus mastership. There can be separate address bus tenures and data bus tenures.

Throughput. The measure of the number of instructions that are processed per clock cycle.

Time-division multiplex (TDM). A single serial channel used by several channels taking turns.

Transaction. A complete exchange between two bus devices. A transaction is typically comprised of an address tenure and one or more data tenures, which may overlap or occur separately from the address tenure. A transaction may be minimally comprised of an address tenure only.

Transfer termination. Signal that refers to both signals that acknowledge the transfer of individual beats (of both single-beat transfer and individual beats of a burst transfer) and to signals that mark the end of the tenure.

Translation lookaside buffer (TLB). A cache that holds recently-used *page table entries*.

U

User mode. The operating state of a processor used typically by application software. In user mode, software can access only certain control registers and can access only user memory space. No privileged operations can be performed. Also referred to as problem state.

V

Virtual address. An intermediate address used in the translation of an *effective address* to a physical address.

Virtual memory. The address space created using the memory management facilities of the processor. Program access to *virtual memory* is possible only when it coincides with *physical memory*.

W

Way. A location in the cache that holds a cache block, its tags, and status bits.

Word. A 32-bit data element.

Write-back. A cache memory update policy in which processor write cycles are directly written only to the cache. External memory is updated only indirectly, for example, when a modified cache block is *cast out* to make room for newer data.

Write-through. A cache memory update policy in which all processor write cycles are written to both the cache and memory.

Index